Multi-Area Thévenin Equivalent Based Power Flow Techniques

by

Alexander Guido De Maeseneer

B.A.Sc., The University of British Columbia

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies
(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

October 2010

Abstract

This thesis explores the possibility of applying the existing Multi-Area Thévenin Equivalent (MATE) algorithm to the power flow problem. Various theoretical considerations/difficulties of handling link connections in power flow are discussed. The current equation power flow program is examined in hopes of aiding link decoupling by taking advantage of the current equation's inherent symmetrical links. However, implementation and testing of the current equation program indicated contrary results to recently published material on current equation programs.

In an attempt to make MATE viable for power flow, one of MATE's bottlenecks was examined, the link matrix. It was found that the problem could be alleviated by using a multi-level approach. This approach would allow link computation to be distributed across the subsystems and levels. An existing multi-level MATE algorithm has already been proposed but was implemented for only two levels. This thesis proposes a massively parallel algorithm for a general number of levels. The distribution of the link matrix allows for mass parallelization of the system matrix into very small subsystems. A flop analysis of the proposed multi-level MATE algorithm reveals that the majority of the computation is spent performing independent small matrix multiplication operations.

Upon inspection of the strengths of the proposed multi-level MATE algorithm, it appears that the algorithm would benefit from a parallel computing platform such as modern GPUs. Today's GPUs contain hundreds to thousands of scalar processors providing approximately an order of magnitude in computational power over multi-core CPUs. This has garnered the GPU much attention in many scientific disciplines. To test the feasibility of the MATE's algorithm on the GPU, the algorithm's most common operation, small matrix multiplication, was implemented. The test case was arranged to simulate the conditions of a 15,000 node system being factorized. The routine is meant to serve as the algorithm's BLAS since linear algebra libraries on the GPU are not meant to handle very small matrices. The routine was found to successfully achieve a decent portion of the GPU's peak flops.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ALU	Arithmetic Logic Unit
API	Application Programming Interface
BBDF	Block Bordered Diagonal Form
BLAS	Basic Linear Algebra Subprograms
CG	Conjugate Gradient
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
DDM	Domain Decomposition Methods
EMTP	ElectroMagnetic Transients Program
FACTS	Flexible AC Transmission Systems
FP	Floating Point
FLOPS	Floating Point Operations per Second
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
GPGPU	General Purpose computing on Graphics Processing Unit
GS	Gauss-Seidel
ISA	Instruction Set Architecture
LAPACK	Linear Algebra PACKage
LTC	Load Tap Changing
MAD	Multiply-Add

MATE	Multi-Area Thévenin Equivalent
MNA	Modified Nodal Analysis
NR	Newton-Raphson
OVNI	Object Virtual Network Integrator
PF	Power Flow
PTX	Parallel Thread Execution
SCI	Scalable Coherent Interface
SDK	Software Development Kit
SGEMM	Single Precision General Matrix-to-Matrix Multiplication
SFU	Special Functions Unit
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
SP	Streaming/Scalar Processor
TCUL	Tap Changing Under Load
VDHN	Very DisHonest Newton-Raphson
WECC	Western Electricity Coordinating Council

Acknowledgments

The creation of this thesis has been brought about by the help of many individuals. First and foremost I would like to give my sincerest gratitude to Dr. Luis Linares for being a source of inspiration and providing me the opportunity and freedom to carry out the research found in this thesis. I would also like to thank Dr. José Marti for graciously acting as co-supervisor. Dr. Hermann Dommel for his insight on current equation power flow algorithms. I'd also like to thank my thesis advisory committee for participating in my thesis defense. Special thanks to Dr. Marcelo Tomim whose help and guidance with sparse matrix packages greatly furthered the research in this thesis. Thanks to Matthew Hsu for his assistance with Linux and CUDA. And finally I would like to thank everyone in the power lab for their kindness and support.

Chapter 1

Introduction

Very early power systems where designed, built and operated by monopolies such as government bodies or public utilities. Electrical networks were often found in separate and isolated regions of generators and loads. The system operator's main goal was primarily placed on system reliability. However with the deregulation of the power market in the recent decades, emphasis began to shift from reliability to a more retail/market-centric view. As demand for power grew, electrical networks have also become more interconnected allowing disturbances to propagate over farther distances resulting in a more vulnerable system. [27, 40, 42]

There are various software tools used to help ensure continuous operation during a disturbance to the system. Disturbances may be a loss of a transmission line, transformer, generator or major load. Determining an appropriate response to return the system into a new "safe" equilibrium is known as a contingency. The validity of a contingency is evaluated based on how the system reaches steady state (i.e. transient analysis) and how the system behaves in steady state. Analysis of the system's transient behaviour is known as dynamic security assessment (DSA). DSA programs typically focus primarily on the transient stability assessment (TSA) program which concerns itself with the issue of maintaining synchronous operation of the AC generators. Analysis of the pre- and post-fault system conditions is known as static security assessment (SSA). It is the power flow (aka load flow) program that simulates the system in steady state to provide the necessary data for analysis. [27]

In an effort to speed up security assessment software, the MATE (Multi-Area Thévenin Equivalent) algorithm was applied to the TSA program in [56]. MATE's algorithm would tear the power network into subsystems. Each subsystem could then compute their Thévenin equivalent values independently from the other. A reduced system can then be formed by interconnecting all the Thévenin equivalents which can then be used to determine the linking currents between subsystems. The solution to the link currents are then distributed to all the subsystems allowing for parallel computation of the nodal voltages. A commodity cluster of off-the-shelf computers were used to realize this implementation.

One of the main goals of this thesis is to further extend MATE's application into the realm of SSA through the power flow program. The objective of applying MATE to power flow is to achieve a speedup in execution time. This can be accomplished through MATE's ability to parallelize the power flow algorithm which can allow for concurrent execution over more than one processor. This can be quite

advantageous, especially given the current trends in microprocessor design.

Over the last several years, there has been minimal performance improvements made for single processors despite the continued improvements in transistor densities as predicted by Moore's Law. This can be attributed to various limitations in the current microprocessor design that are beginning to be reached. For instance, further gains from superscalar design are now providing severe diminishing returns when attempting to extract further parallelism from sequential code. Another limiting factor is the memory speed being much slower than the actual CPU speed. Typically cache and out-of-order execution helps remedy this discrepancy however the effects of diminishing returns is becoming significant. Also, the cost of powering transistors is reaching the limits of what is economically feasible. In all these cases, a large cost is being expended for only a small gain. Thus the general trend in the last few years is to use the additional transistors to add on more cores rather than to further increase the performance of a single core. Processors have evolved from single-core to dual-core, quad-core, hex-core and soon octo-core. Thus parallel algorithms such as MATE have the potential to benefit a lot from current and future computing architectures. [37, 6]

The rest of this chapter will provide a literature review, objectives & motivation, and conclude with an overview of the thesis organization. The majority of literature review is found in this chapter, however other specialized chapters, such as the GPU chapter, will provide a further survey on relevant literature at the start of the chapter. The literature review found here will first start with power flow algorithms and then move towards parallel algorithms.

1.1 The Power Flow Problem

A power flow (aka load flow) program is concerned with solving the nodal voltages, and in effect solving the flow of power between nodes, in an electrical network in steady state. The power flow program plays a critical role in planning and designing of future power networks. It also provides pre- and post-fault static conditions during contingency analysis. The power flow problem models an electrical network by a set of nodal equations. Typically each node (aka bus) is represented by their power equation as shown in equation 1.1 where N_k represents the set of buses adjacent to bus k as well as bus k itself. These formulae represent the net power at each bus as function of nodal voltages. Notice that since the net current at bus k also contains a V_k term, each nodal equation contains a term where V_k is squared. Hence the system of equations that model a power network in steady state are quadratic.

$$\begin{aligned} S_k &= V_k I_k^* \\ S_k &= V_k \left(\sum_{m \in N_k} Y_{km} V_m \right)^* \end{aligned} \tag{1.1}$$

The apparent power S_k can be split into its real and imaginary components. When power is specified at a bus, it refers to the net power at that bus as shown in equation 1.2. Depending on the type of bus, these quantities may or may not be fixed (or specified/scheduled). The bulk of the computation in a load flow is determining the unknown voltages in the network from the system of nodal power equations.

$$\begin{aligned} P_k &= P_k^{gen} - P_k^{load} \\ Q_k &= Q_k^{gen} - Q_k^{load} \end{aligned} \tag{1.2}$$

There are generally three types of buses which will either consume power or generate power. At each bus there are four quantities: P_k, Q_k, $|V_k|$ and δ_k. Two of the four quantities will be specified and two will be unknown. The specifics of which quantities are known/unknown depends on the bus type.

Load buses, will specify the real and reactive power that is being consumed. This data is typically known from historical record, load forecast or measurement. Since both the real and reactive power are specified, these buses are referred to as PQ buses. The unknown quantities are the voltage magnitude and angle.

Voltage controlled buses (aka PV buses), are generator nodes where the voltage magnitude as well as the real power can be set by adjusting the prime mover and generator excitation, respectively. Thus both P_k and $|V_k|$ are specified where δ_k and Q_k are unknown.

At a **slack bus** (aka reference bus or swing bus) both the $|V_k|$ and δ_k are specified and P_k and Q_k are unknown. The purpose of a slack bus can be described as follows. Consider a system made up of only PQ and PV buses. Since both types of buses specify real power, the total losses (I^2R losses) in the system can be determined by the equation shown below.

$$P_{losses} = P_{gen} - P_{load} \tag{1.3}$$

However this implies that the total I^2R losses in the system are known prior to the solution of the system. Thus the need for a slack bus to make up for the difference between P_{gen} and $\left(P_{load} + I^2R\right)$ after the system voltages and currents are solved. In addition, it is sometimes more realistic to distribute this power across more than one slack bus [19].

Since a general power system in steady state can be modeled by thousands of quadratic equations, an analytical solution is not feasible. Thus numerical methods such Gauss-Seidel or Newton's Method are employed. These methods begin with an initial guess for all the unknowns in the system, then proceed in an iterative manner until a sufficiently accurate solution is obtained. Prior to the start of these iterative methods, an admittance (or impedance) matrix must be formed and an initial guess must be made.

When solving the power flow problem a system admittance matrix or impedance matrix is needed to specify how the power network is interconnected. The admittance matrix is by far the most widely used due to the admittance matrix being very sparse whereas the impedance matrix is completely dense. Hence Sparsity Techniques can be applied to the admittance matrix which allows for a drastic reduction in both computation time and storage. This performance improvement is achieved by only performing computation on the non-zero elements while avoiding all the zero elements. Thus when Sparsity Techniques began being implemented in power system simulations in the mid 1960s, Y-matrix methods became the most prevalent technique.

Finally, to start an iterative method, an initial guess must be made of all the nodal voltages in the system. Typically a "flat start" is used as a guess which is is when nodal voltages are set to the slack bus voltage (except of course for the specified voltage magnitude on PV buses) which is typically $1.0\angle 0$.

Alternatively, sometimes the results of a previous simulation are used instead of the flat start. From here, the Gauss-Seidel method or the Newton-Raphson Method can be used to solve the power flow problem. [26, 22]

1.1.1 The Gauss-Seidel Method

The Gauss-Seidel method begins by equating the scheduled power with the calculated power at every bus in the system excluding the slack buses. This is shown in equation 1.4 where N_k represents the set of buses adjacent to bus k as well as bus k itself but excluding slack buses. The voltages and admittances are phasors which can be represented in polar coordinates or more commonly in rectangular coordinates.

$$P_k^{sch} + jQ_k^{sch} = V_k \left(\sum_{m \in N_k} Y_{km} V_m \right)^* \tag{1.4}$$

From the nodal power equation, V_k can be extracted out of the summation and then reordered as shown below.

$$V_k = \frac{1}{Y_{kk}} \left[\frac{P_k^{sch} - jQ_k^{sch}}{V_k^*} - \sum_{m \neq k} Y_{km} V_m \right] \tag{1.5}$$

This reordered formula will serve as the iterative equation which is the essence of the Gauss-Seidel algorithm. As shown in equation 1.6, the voltage at bus k for the current iteration $(i+1)$ can be determined from voltages from the previous iteration i and also from voltages that have already been computed for the current iteration $(i+1)$. Note that the summation variable m for the second term of equation 1.6 is set to start at $m = 2$ because it assumes there is only one slack bus and is numbered as bus 1.

$$V_k^{(i+1)} = \frac{1}{Y_{kk}} \left[\frac{P_k^{sch} - jQ_k^{sch}}{\left(V_k^{(i)}\right)^*} - \sum_{m=2}^{k-1} Y_{km} V_m^{(i+1)} - \sum_{m=k+1}^{N} Y_{km} V_m^{(i)} \right] \tag{1.6}$$

Thus the algorithm for the Gauss-Seidel method is to first begin with an initial guess, typically a flat start. Then iterate with equation 1.6 until the change in voltages are less than a specified minimum. Once all the voltages are solved, the flow of power between buses can be determined.

1.1.1.1 Acceleration factor

It has been found empirically that the number of iterations needed to converge can be substantially reduced if an acceleration factor is used. At each iteration, the Gauss-Seidel method takes relatively small steps towards the desired root. By using an acceleration factor, the steps are extrapolated, hence larger steps are taken. Equation 1.7 describes the techniques where α is the acceleration factor. The acceleration factor is typically set between $1 < \alpha < 2$ because if the acceleration factor is set too large, overshoot can occur resulting in the program to diverge. The actual value of the acceleration factor is

determined from years of experience.

$$V_{acc}^{(i+1)} = V_{acc}^{(i)} + \alpha \left(V_{acc}^{(i+1)} - V_{acc}^{(i)} \right) \tag{1.7}$$

1.1.1.2 Handling PV Buses

Since PV buses do not specify Q^{sch}, it is calculated as shown below for use in equation 1.6. However, from a practical viewpoint, a generator is designed to produce reactive power within some maximum and minimum limit. Thus if the calculated reactive power is beyond either limit, it is simply set to the violated limit.

$$Q_k = -\left\{ V_k^* \sum_{m \in N_k} Y_{km} V_m \right\} \tag{1.8}$$

Also, the voltage magnitude at a PV bus is held constant at a scheduled value $|V^{sch}|$. Thus it is necessary to ensure that this condition is satisfied mathematically by applying the correction below to all PV buses.

$$V_k = |V_k^{sch}| \frac{V_k}{|V_k|} \tag{1.9}$$

1.1.1.3 Practical Usage

The Gauss-Seidel algorithm is relatively simple and used to be the most popular method used in the early days of digital computers. However, there are many practical situations which cannot be solved by the Gauss-Seidel method, but can be solved by Newton's method [53]. In addition to this, the Gauss-Seidel algorithm requires a lot of iterations to converge and the overall computational speed of the algorithm cannot compete with the Newton-Raphson method. Nevertheless, the Gauss-Seidel method is still often used to start the Newton-Raphson algorithm by running one or two iterations prior to the start of NR. In certain situations, a flat start is not a sufficiently accurate to serve as an initial guess for the NR algorithm. However, the GS algorithm is capable of tolerating a poor initial guess making it the ideal algorithm for this purpose. [26, 22]

1.1.2 The Newton-Raphson Method

The Newton Raphson Method is the main algorithm used in today's power flow programs. Shortly after the advent of Sparsity Techniques in the mid 1960s, the NR method became widely used due to its ability to converge quickly while requiring relatively low computation and memory storage. The full details are given in [53] which was the first published NR power flow implementation that used sparsity techniques.

1.1.2.1 Newton-Raphson Algorithm

Newton's method for both the general scalar and multi-dimensional cases can be described as follows. Consider a quadratic function $f(x)$ of which we would like to find its roots. Thus we seek a value x such

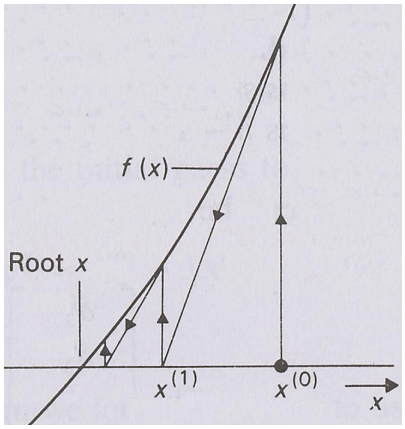


Figure 1.1: Newton's Method
(modified from: O.I. Elgerd. Electric energy systems theory. McGraw-Hill New York, 1971.)

that $f(x) = 0$.

To apply Newton's method, $f(x)$ can be linearized around some initial guess $x^{(0)}$ that is sufficiently close to the desired root. This can be done by approximating $f(x)$ by the first two terms of its Taylor series shown below.

$$f(x^{(0)}) + f'(x^{(0)})\,(x - x^{(0)}) = 0 \tag{1.10}$$

Then the root of the approximated tangent line can be given by equation 1.11.

$$x^{(1)} = x^{(0)} - \frac{f(x)}{(df/dx)^{(0)}} \tag{1.11}$$

If the initial point chosen was sufficiently close to the solution, then root of the linearized function $f(x)$ will provide a more accurate approximation of the actual root of $f(x)$ as shown in figure 1.1. The procedure is then repeated as many times as necessary until a desired accuracy is achieved. Thus in general for an iteration i, a closer approximation can be achieved with equations 1.12.

$$x^{(i+1)} = x^{(i)} - \frac{f(x)}{(df/dx)^{(i)}} \tag{1.12}$$

The method can also be expanded for a system of n nonlinear equations with n unknowns. Each equation is then approximated by its tangent hyperplane by expanding each equation into their Taylor series around the initial guess as shown below.

$$\begin{bmatrix} f_1(\mathbf{x}^{(0)}) \\ \vdots \\ f_n(\mathbf{x}^{(0)}) \end{bmatrix} + \begin{bmatrix} \left(\frac{\partial f_1}{\partial x_1}\right)^{(0)} & \cdots & \left(\frac{\partial f_1}{\partial x_n}\right)^{(0)} \\ \vdots & & \vdots \\ \left(\frac{\partial f_n}{\partial x_1}\right)^{(0)} & \cdots & \left(\frac{\partial f_n}{\partial x_n}\right)^{(0)} \end{bmatrix} \begin{bmatrix} (x_1 - x_1^0) \\ \vdots \\ (x_n - x_n^0) \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \tag{1.13}$$

The derivative term from the scalar case is now a matrix of partial differentials which is referred to as a Jacobian matrix. The matrix equation can then be reordered, as was done in the scalar case, where

the Jacobian matrix is inverted as shown below.

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} - \left[\mathbf{J}^{(0)}\right]^{-1} \mathbf{f}(\mathbf{x}^{(0)}) \tag{1.14}$$

Thus in general the iterative formula would be that of equation 1.15. This formula is repeated until a prespecified accuracy is reached.

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} - \left[\mathbf{J}^{(i)}\right]^{-1} \mathbf{f}(\mathbf{x}^{(i)}) \tag{1.15}$$

Alternatively, the procedure could also be split into two steps as shown in the two equations shown below. This is the formulation used in a typical NR power flow program.

$$\triangle\ \mathbf{x} = \left(\mathbf{x}^{(i+1)} - \mathbf{x}^{(i)}\right) = -\left[\mathbf{J}^{(i)}\right]^{-1} \mathbf{f}(\mathbf{x}^{(i)}) \tag{1.16}$$

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} + \triangle\ \mathbf{x} \tag{1.17}$$

1.1.2.2 Newton-Raphson method applied to Power Flow

The derivation to the NR power flow algorithm begins again with the nodal power equations. These equations can be divided into their real and imaginary components yielding two equations at a given bus as shown in equation 1.18. It should also be noted that the algorithm can so be derived using nodal current equations instead of power equations which is discussed later in section 3.2.

$$\begin{aligned} P_k^{calc} &= Re\left\{V_k \left(\sum_{m\in N_k} Y_{km} V_m\right)^*\right\} \\ Q_k^{calc} &= Im\left\{V_k \left(\sum_{m\in N_k} Y_{km} V_m\right)^*\right\} \end{aligned} \tag{1.18}$$

At each bus there can be a *scheduled* real and reactive power. In addition, the power at each bus can also be calculated as a function of nodal voltages as written in equation 1.18. Either way, both quantities should be equal and thus equation 1.19 should hold true.

$$\begin{aligned} \triangle\ P_k &= P_k^{sch} - P_k^{calc} = 0 \\ \triangle\ Q_k &= Q_k^{sch} - Q_k^{calc} = 0 \end{aligned} \tag{1.19}$$

However, since the NR algorithm begins with an initial guess, the calculated values use approximated nodal voltages. Thus during the solution procedure, the difference between the scheduled and calculated power is not zero. Hence equation 1.19 is referred to as mismatch equations. The NR algorithm will solve for the roots of the mismatch equations to determine the nodal voltages within the system.

Since the voltages and admittances from the power equations are phasor quantities, they can be represented in either polar or rectangular coordinates. Polar form is the most widely used and will also be used in the derivation in this section. However, there are advantages to using the rectangular form

when the power flow algorithm is derived based on the current equation as oppose to the power equation (see section 3.2).

Thus to solve the power flow problem, Newton's method can be applied to the mismatch equations using polar form. This results in a linearized system that can be conveniently written in equation 1.20. The system matrix is a Jacobian where its partial differential elements are described in equation 1.21.

$$\begin{bmatrix} \triangle P \\ \triangle Q \end{bmatrix} = \begin{bmatrix} H & N \\ M & L \end{bmatrix} \begin{bmatrix} \triangle \delta \\ \triangle |V| \end{bmatrix} \tag{1.20}$$

$$H_{km} = -\frac{\delta P_k^{calc}}{\partial \delta_m} \qquad N_{km} = -\frac{\partial P_k^{calc}}{\partial |V_m|} \qquad M_{km} = -\frac{\partial Q_k^{calc}}{\partial \delta_m} \qquad L = -\frac{\partial Q_k^{calc}}{\partial |V_m|} \tag{1.21}$$

As described earlier in the multi-dimensional NR algorithm, the unknown voltage corrections $\triangle\ \delta$ and $\triangle\ |V|$ are first solved for from the linear system of equation 1.21. The second step of the NR algorithm is to update the voltages for the next iteration $(i+1)$ as shown below. [26]

$$\begin{aligned} \delta^{(i+1)} &= \delta^{(i)} + \triangle\ \delta \\ |V|^{(i+1)} &= |V|^{(i)} + \triangle\ |V| \end{aligned} \tag{1.22}$$

Thus the algorithm for the NR power flow program first begins with an initial guess, such as a flat start. Then algorithm simply repeats the following two steps until the specified accuracy is met:

1. Solve linearized system in equation 1.20
2. Update voltages with equation 1.22

Handling PV nodes

From the definition of a PV bus, the Q^{sch} is not specified thus the reactive power mismatch equation cannot be specified (see equation 1.19). Thus, unlike PQ buses which are modeled by two mismatch equations, the PV bus is only represented by one equation. Fortunately, $|V|$ is specified at PV buses, thus only one equation is needed to solve for the one unknown which is $\triangle\ \delta$.

As mentioned in the GS algorithm, generators are designed to produce reactive power within some maximum and minimum limit. Thus if a limit is violated, the PV bus is then modeled as a PQ bus. Where the Q^{sch} is set to the violated limit and the criteria for constant $|V|$ is relaxed. Conversely, PQ nodes can be regulated between a range of voltages. If a PQ bus exceeds one of the operational limits, reactive-power support may be switched in. Thus the PQ node would then be modeled as a PV node where $|V|$ is set to the violated limit. Switching operation from PQ to PV or vice versa is said to be done at the end of the second iteration [53].

Transformers

Various types of transformer can be modeled in a power flow program. For instance, the following is a list of transformers that can be represented in the IEEE common data format:

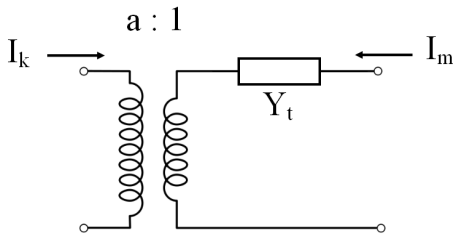


Figure 1.2: Transformer Model in IEEE Common Data Format

- Fixed tap
- Variable tap for nodal voltage control (TCUL, LTC)
- Variable tap for reactive power flow control
- Variable phase angle for real power flow control (phase shifter)

Using the IEEE common data format, the first inputted terminal is the side with the non-unity tap ratio a and the second terminal is the side with the modeled impedance Y_t (see figure 1.2). Following this convention, a general transformer model can be derived and integrated into the power flow algorithm. The tap value a will be considered as a complex quantity, thus making the following derivation applicable to both tap changing transformers and phase-shifting transformers. [48]

Consider the apparent power at each of the terminal nodes which can be calculated as shown in equation 1.23. A relationship between the currents I_k and I_m can then be extracted as shown in equation 1.24.

$$S_k = V_k I_k^* \qquad\qquad S_m = \frac{1}{a} V_k I_m^* \tag{1.23}$$

$$I_k = -\frac{1}{a^*} I_m \tag{1.24}$$

Also, equation 1.25 can be written where the current I_m is determined from the voltage drop across the transformer's admittance Y_t. Then substituting equation 1.25 into equation 1.24 gives the formula for the other current I_k (equation 1.26). The two equations for I_k and I_m can finally be reordered in matrix form as shown in equation 1.27 where the admittance matrix of the transformer is shown.

$$I_m = \left(V_m - \frac{1}{a} V_k \right) Y_t = -\frac{1}{a} Y_t V_k + Y_t V_m \tag{1.25}$$

$$I_k = \frac{1}{|a|^2} Y_t V_k - \frac{1}{a^*} Y_t V_m \tag{1.26}$$

$$\begin{bmatrix} Y_t/|a|^2 & -Y_t/a^* \\ -Y_t/a & Y_t \end{bmatrix} \begin{bmatrix} V_k \\ V_m \end{bmatrix} = \begin{bmatrix} I_k \\ I_m \end{bmatrix} \tag{1.27}$$

Thus to append the transformer model onto a general power system admittance matrix between buses k and m, one would simply add the transformers admittance terms to the proper elements in the system matrix (see equation 1.28). This derivation can now be used to describe the inclusion of the fixed tap, variable tap, and phase-shifting transformers.

$$Y_{modified} = \begin{bmatrix} Y_{11} & & & \cdots & & & Y_{1N} \\ & \ddots & & & & & \\ & & \left(Y_{km} + Y_t/|a|^2\right) & \cdots & \left(Y_{km} - Y_t/a^*\right) & & \\ \vdots & & \vdots & & \vdots & & \vdots \\ & & \left(Y_{km} - Y_t/a\right) & \cdots & \left(Y_{km} + Y_t\right) & & \\ & & & & & \ddots & \\ Y_{N1} & & & \cdots & & & Y_{NN} \end{bmatrix} \tag{1.28}$$

Fixed transformers are the simplest model. They are included during the assembly of the system's admittance matrix as shown in equation 1.28. Its admittance terms remain static throughout the entire simulation and no other changes are made to the power flow algorithm.

There are two kinds of **variable tap changing transformers**, voltage control and reactive power control. *Voltage control* can attempt to regulate a voltage at a bus either connected directly to the transformer or a bus elsewhere in system (aka a remote bus). Thus at the regulated bus k, $|V_k|$ is set to a constant value and is no longer an unknown (and can be removed as a state variable). Instead, the tap ratio a_{km} is now the unknown and can replace $|V_k|$ as the state variable at that node. The system's admittance matrix is also modified as in equation 1.28, however it is no longer static since the tap ratio a_{km} is now variable. Thus the tap ratio a_{km} is now present in the mismatch equations. Hence, when the mismatch equations are linearized with respect to the new state variable a_{km}, the Jacobian matrix is altered. Only the nodes connected to the transformer will contain a_{km} in their mismatch equations and thus only those nodes will contain non-zero partial differentials when linearized with respect to a_{km}. For instance, if bus k is connected to the transformer, the mismatch equation for the reactive power will consist of the following partial differential elements as shown in equation 1.29 (where N_k represents the set of buses adjacent to bus k and M_k represents the set of PQ buses adjacent to bus k).

$$\triangle\, Q_k = \underbrace{\left[-\frac{\partial Q_k^{calc}}{\partial \delta_k} + -\frac{\partial Q_k^{calc}}{\partial \mathbf{a_{km}}}\right]}_{diagonal\,block} + \underbrace{\left[\sum_{m \in N_k}\left(-\frac{\partial Q_k^{calc}}{\partial \delta_m}\right) + \sum_{m \in M_k}\left(-\frac{\partial Q_k^{calc}}{\partial |V_m|}\right)\right]}_{off-diagonal\,blocks} \tag{1.29}$$

However for the case when bus k is a remote node, the diagonal element would be zero. This is because bus k's mismatch equation will not contain any a_{km} terms and thus it will also not contain any non-zero terms when the mismatch equation is linearized with respect to a_{km}. Hence when factorizing the Jacobian matrix, special ordering of the rows/columns must take place to avoid the zero diagonal

element.

Since the tap ratio a_{km} is now a state variable, it also gets updated at the end of an iteration with all the other state variables. Thus once the linearized system is formed, a_{km} is also updated with a tap ratio correction as shown in equation 1.30.

$$a_{km}^{(i+1)} = a_{km}^{(i)} + \triangle\ a_{km} \tag{1.30}$$

A variable tap changing transformer can also control the amount of *reactive power flow* between two nodes k and m. However now, the desired variable to be regulated (Q_{km}) is not an existing state variable as was the case with $|V|$. Since this new unknown is not replacing an existing state variable, a new equation needs to be added. Thus a branch mismatch equation in terms of reactive power can be created in a similar fashion as the nodal mismatch equations (see equation 1.31). This equation is grouped together with the rest of the system's mismatch equations then linearized as was done in step 1 of the NR algorithm. Step 2 of an NR iteration is then done where the state variables are updated as shown in equations 1.22 and 1.30.

$$\triangle\ Q_{km} = Q_{km}^{reg} - Q_{km}^{calc} \tag{1.31}$$

The **phase-shifting transformer** has a similar model as the tap changing transformer that controls reactive power. It is similar in that the real power flow from a branch is not an existing state variable and thus an additional branch mismatch equation is necessary (see equation 1.32).

$$\triangle\ P_{km} = P_{km}^{reg} - P_{km}^{reg} \tag{1.32}$$

However it differs in the fact that now the tap variable a_{km} now has unity magnitude and a non-zero phase angle. Thus the new state variable becomes the phase angle of the transformer ϕ_{km}. Aside from the difference in the state variable, the approach is the same as the tap changing transformer controlling reactive power. Thus step 1 of the NR algorithm remains the same and step 2 of the NR algorithm will include equation 1.33 to update the transformer's phase angle at the end of each iteration.

$$\phi_{km}^{(i+1)} = \phi_{km}^{(i)} + \triangle\ \phi_{km} \tag{1.33}$$

Detailed derivations are provided in the appendix of [47]. It should be noted that the derivation for the tap changing voltage control transformer uses a representation where nodes k and m are reversed from than that of figure 1.2. Also the tap is modeled on the opposite terminal of figure 1.2 and thus their tap ratio is effectively $t = \frac{1}{a}$.

A few other implementation details can mentioned about the transformer model. For instance, the transformers with an active tap or phase angle typically remain static until the end of the second ([53]) or third ([47]) iteration to allow for more accurate system values before applying transformer control. When the transformer taps or phase angle are active, they are allowed to act as continuous values. It is only near the end of the simulation where the final tap or phase angle is rounded to the nearest tap/phase increment. If the tap/phase quantity exceeds its maximum or minimum value, it is simply set to the

violated limit. Afterwards, the constraint variable is checked every iteration to see if the limit is still necessary. [47]

1.2 System Partitioning and Parallel Algorithms

This section attempts to review concepts in parallel algorithms and partitioning techniques in various applications which will be later used in MATE. This section begins with a very brief discussion of Diakoptics. The history of Domain Decomposition Methods (DDM) as well as its direct method techniques, as oppose to the more common iterative methods, are then examined. Finally the MATE algorithm and implementation details are discussed and compared to previous established concepts from the perspective of power systems.

1.2.1 Diakoptics

In 1953, Gabriel Kron published the first paper on Diakoptics [34]. Diakoptics is a method of tearing a system into pieces, obtaining independent partial solutions of each subsystem, then combining the partial solutions to obtain the final result. This approach is the basis of parallel algorithms such as MATE.

Shortly after the inception of Diakoptics, load flow algorithms based on the techniques of Diakoptics came about. The first one can be traced back to 1963 in [10] that used a Z matrix power flow algorithm, which were common in the early 1960s as sparsity techniques haven't yet been discovered [51]. However shortly after the advent of sparsity techniques, it wasn't long until sparsity techniques dominated the power systems arena leaving little room for Diakoptics to flourish.

The original purpose of Diakoptics was not to improve computational speed but rather to gain the ability to solve electrical circuits that were too large for the computers at the time to handle. Although Kron's ideas were at first not well received by the scientific community, the concepts of Diakoptics have later gone on to influence today's parallel computing community. It is believed that Kron's work helped lay the grounds for modern direct *Domain Decomposition Methods*, which will be discussed in the next section. [35]

1.2.2 Domain Decomposition Methods

This section will give a brief history of DDM. The rest of the section will focus on direct methods and will begin by discussing two types of partitioning in DDM. These partitioning techniques are then compared with tearing techniques. This is followed by a discussion of the Schur complement and an example of its application is given for a vertex-based partitioning example.

1.2.2.1 History

Domain Decomposition Methods (DDM) are traditionally a set of techniques performed on discretized partial differential equations using a "divide-and-conquer" paradigm. Just as in Diakoptics, a large problem is decomposed into smaller problems, each of which are computationally easier to solve than the original problem, and most of all these sub-problems can solved in parallel. Historically, DDM has

its origins from the work of H. A. Schwarz (1843-1921) which has later been developed into a wide array of iterative techniques named after Schwarz (e.g. Schwarz alternating/additive/multiplicative methods as well as Schwarz Machinery) [36, 50]. Another area that forms the core of more modern DDM comes from substructuring techniques which were originally created by mechanical engineers for the finite element analysis of complex structures in the 1960s [36]. With the advent of massively parallel computers in the 1980s, DDM became a prime candidate as the framework to parallel algorithms. Since then, DDM has become one of the most successful collection of methods in numerical analysis for parallelization in many scientific applications such as fluid dynamics, structural mechanics, biomechanics, geophysics, plasma physics, radiation transport, electricity and magnetism, and flows in porous media to name a few [9].

The collection of domain decomposition techniques can be classified into various categories. For instance, these techniques can be sub-divided into direct/iterative methods or methods specifically designed for overlapping/non-overlapping domains. Most of the research in DDM focuses on iterative methods, however advancements are still being made on the direct methods. This trend is most evident from the amount of papers on iterative techniques published in the International Conference of Domain Decomposition[1] volumes which are held every ~1.5 years. Much of the research in iterative techniques are invested in reducing the number of iterations. Thus preconditioners are a main source of interest and are typically the most complex operation in domain decomposition [9]. Despite the popularity of iterative methods in DDM, the emphasis will be placed on direct methods since they are the main concern of this thesis.

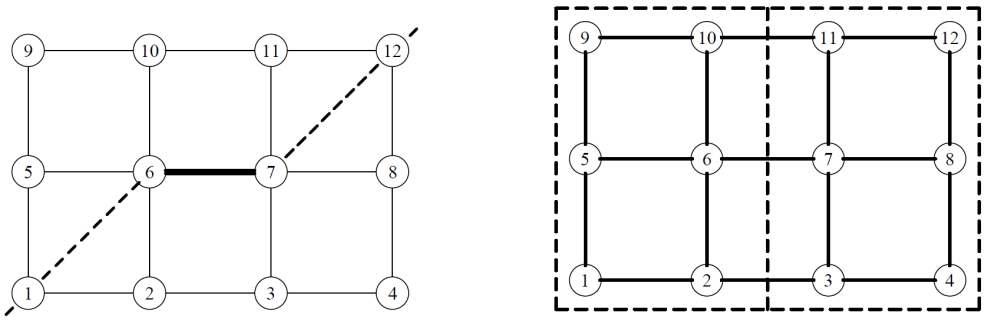
1.2.2.2 Partitioning

One of the first divide-and-conquer ideas in structural analysis partitioned a system for use in a direct solver. There are several ways to go about partitioning a system in DDM. The simplest techniques are edge-based[2] partitioning and vertex-based partitioning. Figure 1.3 (a) and (b) depict a domain being split into two subdomains by both types of partitioning. The type of partitioning indicates the quantity (i.e. vertices or edges) that cannot be split between two subdomains. Thus if the partitioning is edge-based, then edges cannot straddle between two subdomains. Similarly if the partitioning is vertex-based, than vertices cannot be shared between two or more subdomains. [50]

Examples of edge-based and vertex-based partitioning are also shown in figure 1.4 where a domain is divided into three subdomains. In the case of edge-based partitioning, nodes numbered 34 to 40 are chosen as a global interface. By ordering the nodes of each subsystem together and numbering the interface nodes last, the system matrix is turned into block bordered diagonal form (BBDF). In the case of vertex-based partitioning, each subdomain contains its own set of local interface nodes. Thus the interface nodes are numbered last within the scope of their subdomains, as oppose to edge-base where

[1] A list of all conferences can be found at www.ddm.org

[2] In graph theory, nodes and branches are referred to as vertices and edges respectively.



(a) Edge-Based Partitioning (b) Vertex-Based Partitioning

Figure 1.3: Types of Partitioning in DDM

(modified from: Y. Saad. Iterative methods for sparse linear systems. Society for Industrial Mathematics, 2003.)

the interface nodes are numbered last within the scope of the entire domain. Since all the interfaces are local, there is no global interface connecting all the subdomains together. Hence this non-BBDF matrix will yield a different solution procedure than the BBDF matrix. It is also important to note that vertex-based partitioning required almost twice as many interface nodes than edge-based partitioning.

A comparison of these types of partitioning can be made with tearing techniques. Tearing techniques involve removing a small set of branches or nodes from a graph such that the graph is disconnected into parts. Figure 1.5 illustrates both branch tearing and node tearing. In the case of branch tearing, additional equations are introduced to represent the set of branches cut. Branch tearing is also the same technique used in Diakoptics. For node tearing (figure 1.5.b), subsystems are decoupled by removing a set of border nodes from one subsystem. Border nodes are a set of nodes in a subsystem that have branches connecting to them to other subsystems. In figure 1.5.b, two border nodes from subsystem G_2 are removed and placed into a new linking subsystem G_3 which yields a BBDF matrix. Notice that node tearing is essentially the same as the partitioning techniques in DDM. Vertex-based partitioning would be a slight variant, where border nodes from both subsystems are used in a separate linking subsystems.

1.2.2.3 Direct Solution and the Schur Complement

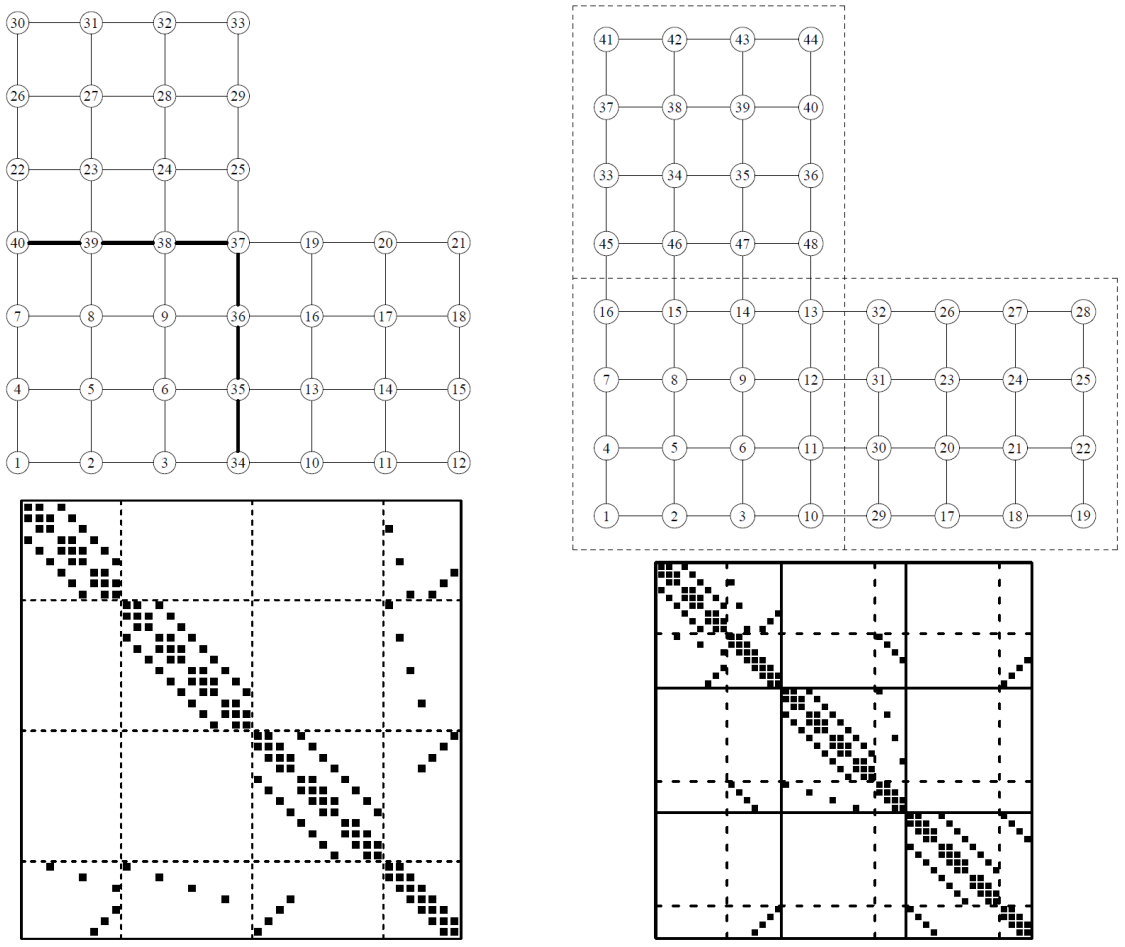
Once the system has been partitioned into subdomains where the matrix is in BBDF, the system can be solved by block Gaussian elimination. Consider the following system where a matrix is divided into submatrices B, E, F, and C.

$$\begin{bmatrix} B & E \\ F & C \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix} \tag{1.34}$$

The first block row can be solved for x in terms of y as shown below.

$$x = B^{-1}\left(f - E\,y\right) \tag{1.35}$$

Upon substituting this expression for x into the second block row yields

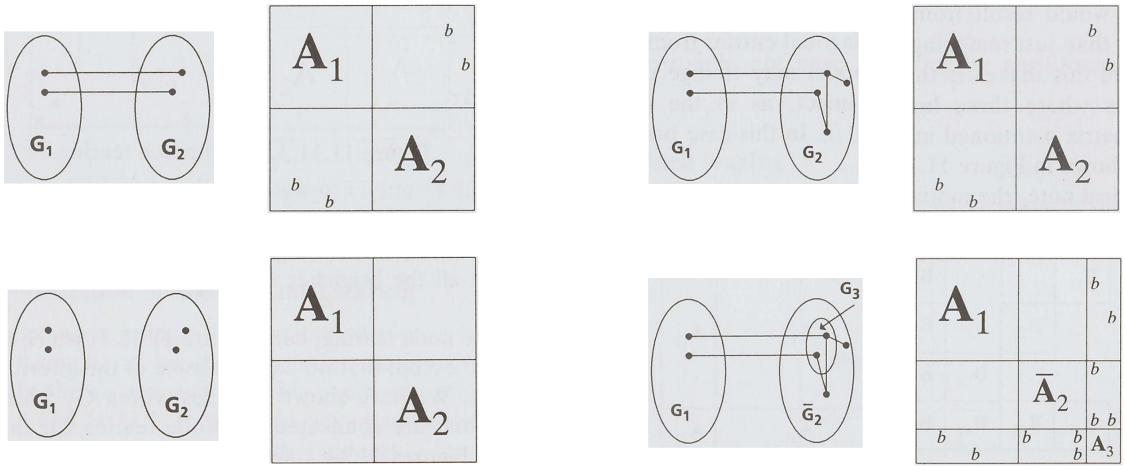


(a) Edge-Based

(b) Vertex-Based

Figure 1.4: Examples of Partitioning Types

(modified from: Y. Saad. Iterative methods for sparse linear systems. Society for Industrial Mathematics, 2003.)



(a) Branch Tearing

(b) Node Tearing

Figure 1.5: Tearing Techniques

(extracted from: I.S. Duff, A.M. Erisman, and J.K. Reid. Direct methods for sparse matrices. Oxford University Press, USA, 1989.)

$$\left(C - F\,B^{-1}E\right) y = g - F\,B^{-1}f \tag{1.36}$$

Where the matrix

$$S = C - F\,B^{-1}E \tag{1.37}$$

is called the *Schur complement.* If the Schur complement can be formed, the solution to the interface variables y can be obtained. Then once the interface values are solved, the subdomain variables x can be obtained to achieve the total solution to the system. [50]

This technique can easily be extrapolated to systems of two or more subdomains (as shown in equation 1.38). For a domain with s subdomains, equation 1.39 provides the general Schur complement. The Schur complement is named after Issai Schur (1875-1941) where the notation was first used in 1968 in a partitioned (two-way block) matrix. However earlier implicit manifestations of the Schur complement was first published in 1812 by Pierre Simon Laplace. [65]

$$\begin{bmatrix} B_1 & 0 & E_1 \\ 0 & B_2 & E_2 \\ F_1 & F_2 & C \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ y \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ g \end{bmatrix} \tag{1.38}$$

$$S = C - \sum_{i=1}^{s} C_i - F_i\,B_i^{-1}\,E_i \tag{1.39}$$

The general procedure using this approach is listed below. Notice that the subdomain computation for steps 1) and 3) are independent which allows for parallel processing.

1. Form the Schur complement & the RHS of equation 1.36

2. Solve for the interface variables y

3. Back-substitute y to obtain subdomain variables x

Practical implementations of this method will factorize the B_i matrices but will not calculate their inverses explicitly . Instead linear systems are created that solve for intermediate quantities E' and f' via factorization then forward/backward substitution (see equation 1.40) . These intermediate quantities are then substituted into the Schur complement as well as the RHS of equation 1.36 which is shown in equations 1.41 and 1.42 ([50] section 13.2.1). The same technique is also done in the large scale implementation of MATE [55].

$$B\,E' = E \qquad B\,f' = f \tag{1.40}$$

$$S = C - F\,E' \tag{1.41}$$

$$S\,y = g - F\,f' \tag{1.42}$$

1.2.2.4 Vertex-Based Solution

In vertex-based partitioning, the system matrix is not in BBDF (see figure 1.4.b) which results in a slightly different procedure. Notice that instead of a global block border, each diagonal block contains its own local block border. Hence each subdomain (diagonal block) is of the form shown below.

$$\begin{bmatrix} B_i & E_i \\ F_i & C_i \end{bmatrix} \tag{1.43}$$

Thus each subdomain contains both interior nodes as well as interface nodes. Then consider a subdomain's block row shown in equation 1.44. The E_{ij} terms reflect the connections between the local interfaces of subdomains in the off-diagonal blocks as seen in figure 1.4.b. Thus N_i is the set of local interface variables that subdomain i is connected to.

$$\begin{aligned} B_i x_i + E_i y_i &= f_i \\ F_i x_i + C_i y_i + \sum_{j \in N_i} E_{ij} y_j &= g_i \end{aligned} \tag{1.44}$$

Then just as was done in the solution to edge-based partitioning, an x_i is extracted from the first equation ($x_i = B_i^{-1}\,(f_i - E_i\,y_i)$) and is substituted into the the second equation to give:

$$S_i\,y_i + \sum_{j \in N_i} E_{ij} y_j = g_i - F_i\,B_i^{-1} f_i \tag{1.45}$$

Notice the interface equation is only a function of interface variables y_i. By collecting these interface equations from all the subdomains a *global* Schur complement can be assembled. For example, the global Schur complement for figure 1.4.b is shown below.

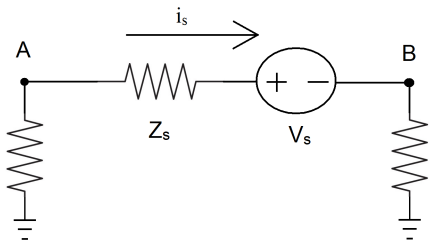


Figure 1.6: MNA Example Circuit

$$S = \begin{bmatrix} S_1 & E_{12} & E_{13} \\ E_{21} & S_2 & 0 \\ E_{31} & 0 & S_3 \end{bmatrix} \tag{1.46}$$

An interesting result of the solution to vertex-based partitioning is that the Schur complement is not completely dense in general as was the case with edge-based partitioning. The off-diagonal blocks E_{ij} only exist if there is coupling between subdomains i and j Thus a system with many weakly coupled subdomains would result in a much more sparse Schur complement. However this sparsity comes at a cost of having a much larger Schur complement matrix since vertex-based partitioning requires almost twice as many interface nodes than edge-based partitioning. [50]

1.2.3 Multi-Area Thévenin Equivalent

Multi-Area Thévenin Equivalent (MATE) is yet another divide-and-conquer technique specifically designed in the area of power systems. As described in [41], MATE achieves parallel computation by utilizing concepts from Diakoptics in the form of branch tearing. Modified nodal analysis (MNA) then assists with the representation of the torn branches. Once the system is partitioned, block Gaussian elimination (as described in section 1.2.2.3) is used to solve the system. This section discusses parallels with the direct solution approach in DDM but from an electrical systems viewpoint.

To demonstrate MNA's usage in MATE, consider the two node system in figure 1.6. Then suppose the branch between node A and B is torn. Using MNA, the system of equations will appear as shown in equation 1.47 where the branch equation is placed at the bottom of the matrix. This third equation is used to compute the current flowing through the linking branch. Then due to the removal of the connecting branch a $+1$ and -1 appear in the i_s column of the admittance matrix to include the effects of current being injected into the system and current being drawn from the system respectively. It is also important to note that the branch is *physically* being removed, thus the branch's admittance is also being removed from the self-admittance of Y_a and Y_B. This is exactly what MATE does to partition a power system.

$$\left[\begin{array}{cc|c} Y_A & 0 & +1 \\ 0 & Y_B & -1 \\ \hline +1 & -1 & Z_s \end{array}\right] \begin{bmatrix} v_A \\ v_B \\ i_s \end{bmatrix} = \begin{bmatrix} i_A \\ i_B \\ V_s \end{bmatrix} \tag{1.47}$$

Since branch tearing introduces an additional equation for every branch that is torn, an ideal power

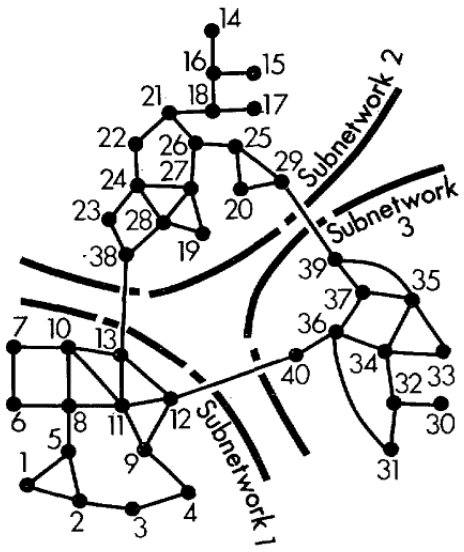


Figure 1.7: Weakly Interconnected Subsystems

(extracted from: W. Tinney and W. Meyer. Solution of large sparse systems by ordered triangular factorization. IEEE Transactions on Automatic Control, 18(4):333–346. © 1973 IEEE)

system would be one with weakly interconnected subsystems as shown in figure 1.7. In this case, only three additional branch equation would be introduced to the system matrix. The system can be represented as shown in equation , where the p terms are sparse matrices which represent the set of torn branch connections with $+1$'s and -1's, as described in MNA above, for each subsystem. Since the p serve to inject the currents of the torn branches into each subsystem, they are referred to as injection matrices.

$$\left[\begin{array}{ccc|c} Y_1 & & & p_1 \\ & Y_2 & & p_2 \\ & & Y_3 & p_3 \\ \hline p_1^t & p_2^t & p_3^t & Z \end{array}\right] \left[\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ i_\alpha \end{array}\right] = \left[\begin{array}{c} h_1 \\ h_2 \\ h_3 \\ 0 \end{array}\right] \tag{1.48}$$

With the system matrix in BBDF, block Gaussian elimination can be performed in the same manner as was done in section 1.2.2.3. A Schur complement is formed and the interface quantities, in this case branch currents, are solved. The interface quantities are then feedback to each of the subsystems which can then be solved independently of each other.

In contrast to general DDMs, the submatrices formed in MATE during the solution procedure also possess a physical meaning. As described in [55], the injection matrix p_i maps subsystem nodes to link branches. Thus equation 1.49 is used to extract the Thévenin equivalent impedance from the perspective of the link branches of subsystem i. Also, the internal voltages of a subsystem when disconnected from the rest of the system can be determined by $e_i = Y_i^{-1} h_i$. The Thévenin equivalent voltages from the perspective of the link branches can then be determined with equation 1.50, which selects the appropriate voltages from e_i.

$$Z_i^{th} = p_i^t \, Y_i^{-1} \, p_i \tag{1.49}$$

$$e_i^{th} = p_i^t \, e_i \tag{1.50}$$

Then each subsystem can be represented by their Thévenin equivalents to yield a much smaller system which can then be solved to obtain the link currents in the system. Intuitively, to obtain the current in a simple series circuit, one would simply sum the impedances and voltages. The current would then be obtained by $I = V_{eq}/Z_{eq}$. Analogously, the Thévenin equivalents are also summed as shown in equation 1.51. With these quantities, the link currents can be obtained by solving the link system in equation 1.52. Notice that mathematically the procedure is the same as forming the Schur complement then solving for the interface variables. Once the link currents are solved, they are fed back to each subsystem where their voltages can be solved independently of each other.

$$Z_\alpha = Z_l + \sum_{\forall i} Z_i^{th} \qquad\qquad e_\alpha = \sum_{\forall i} e_i^{th} \tag{1.51}$$

$$Z_\alpha i_\alpha = e_\alpha \tag{1.52}$$

There are several advantages associated with this method. The independence of the subsystems allows for different techniques such as using different step sizes to reduce overall computation. The decoupling of the subsystems even allows for the possibility of using different integration rules. Also, since MATE uses the MNA branch equations, switches can be modeled to prevent topological changes to the system which avoids re-triangularization of subsystems [41]. For applications such as HVDC converter, significant speedups have been reported [3].

MATE was later redesigned in [56] to tackle Transient Stability Assessment (TSA) simulations of actual large scale power systems. This implementation used a cluster of CPUs and was the only practical implementation of MATE for general large scale power systems. The algorithm incorporated several well-known and widely-used software packages. METIS was used to partition the system, SuperLU (coupled with GotoBLAS) was used to factorize the subsystem matrices and LAPACK was used for dense matrix operations. All communications between processors used MPI over an SCI network.

MATE's algorithm was reformulated where a series of mapping schemes were used to ensure that the Thévenin equivalent quantities were with respect to the border nodes instead of the link branches. The mapping schemes allow for better handling of cases where a border node is connected to more than one link branch. In these situations, a reduction in computation and data transfer can be achieved as explained in section 2.3 of [55]. A detailed description and procedure for obtaining the Thévenin equivalents of subsystems was also given. Coincidentally, the mathematics behind the procedure happen to be equivalent with that of the practical implementations of block Gaussian elimination in DDM (described in section 1.2.2.3). This newer version of MATE obtained speedups of approximately 7 times using 14 CPUs over a conventional sparse solver on a single CPU for a ~15,000 node WECC system [56].

1.3 Motivation and Objectives

Trends in computing are clearly moving towards parallel architectures, thus it is only natural to look into parallel algorithms for power systems. Literature on parallelizing power flow algorithms can already be found for architectures such as CPU clusters, transputers, FPGAs and even GPUs. Node tearing (described in section 1.2.2.2) appears to be the most common partitioning method used for power flow. However, multi-level techniques (described at the start of chapter 4) appear to provide superior scaling across multiple processors. In fact, MATE already has its own multi-level algorithm as described in [5] which appears to play a vital role in parallelization. It will be seen that by furthering the existing multi-level MATE algorithm, the use of massively parallel computing platforms (such as GPUs) appear to become viable for power system simulations.

In the last several years, graphics processing units (GPUs) have been beginning to cross over into general purpose computation on GPUs (GPGPU). GPUs focus on creating much "simpler" processing cores than the superscalar cores of CPUs. This allows for modern GPUs to consist of hundreds to thousands of scalar processors. GPUs are known for their raw computational power which is typically an order magnitude larger than a CPU (see figure 1.8). Hence many substantial speedup claims from various simulation programs have been reported across the scientific community. GPUs have even been making their presence known in supercomputers. As of June 2010, two of the top 10 most powerful[3] supercomputers in the world contain GPUs. The Nebulae is ranked at #2 and is composed of NVIDIA Tesla C2050 GPUs and Intel processors. And the Tianhe-1 is ranked at #7 which consists of ATI's Radeon HD 4870 and Intel processors. [44]

The main objective of this thesis is to explore the possibility of expanding MATE to the power flow problem successfully. First an appropriate type of power flow program must be chosen to serve as a base for MATE. Also, given the recent developments in the area of GPUs, it would be advantageous if MATE's algorithm could be adapted to suit the GPU architecture. This would then provide the possibility of not only accelerating power flow simulations but also other power system simulations such as the transient stability program.

1.4 Thesis Organization

This thesis contains four research chapters. The first research chapter begins with Chapter 2 that discusses the issue of partitioning in power flow. It covers how the link equation which originally represented the flow of current in EMTP can be represented in power flow. This chapter also makes a distinction between two types of branch tearing and suggests the possibility of node tearing. It also addresses the challenges associated with tearing certain types nodes in power flow, such as two PQ nodes.

Chapter 3 acknowledges the added difficulty of tearing in power flow than with EMTP by exploring

[3] As ranked by http://www.top500.org/

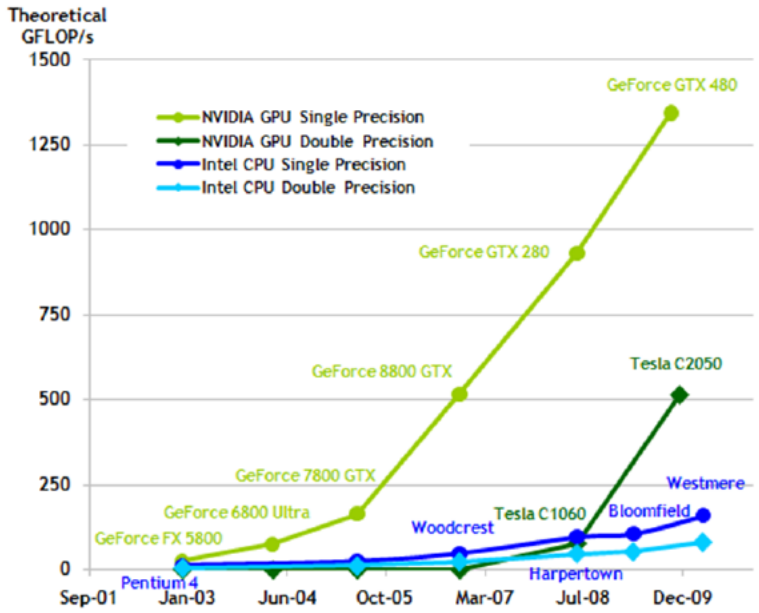


Figure 1.8: GPU vs. CPU Theoretical Flops
(modified from: NVIDIA Corporation. NVIDIA CUDA Programming Guide, 3.1.1 edition, July 2010.)

ways to optimize the overall algorithm. The main focus of this chapter is to verify the viability of using the current equation power flow program as the base program for MATE. The current equation program appears promising due to its symmetrical branches and reported performance compared to a conventional power flow program. CPU implementations (both using SuperLU) of a conventional power flow program as well as a current equation power flow program were implemented and benchmarked against each other. Vague implementation details from existing papers are also clarified.

Chapter 4 attempts to reformulate the MATE algorithm to a multi-level approach. The existing multi-level MATE algorithm is furthered to extend beyond two levels. A detailed level three example is provided. Finally, a simplistic model is made so that a flops analysis can be performed to determine the distribution of the flops.

With the creation of the proposed algorithm, Chapter 5 looks into the feasibility of utilizing the GPU as a computing platform. This chapter will attempt to create a kind of GPU based BLAS for the multi-level MATE algorithm. The chapter begins with an overview of GPU hardware and software. Then a small matrix representation is proposed which can serve as the starting point for all computation done on the GPU using the multi-level MATE algorithm. From the flops analysis of the previous chapter, it was determined that small matrix multiplication was the most common operation performed in the MATE algorithm. Thus a small matrix multiplication routine was designed, implemented and benchmarked to determine the viability of the GPU as a computing platform.

Concluding remarks as well as suggested future work is detailed in the final chapter.

Chapter 2

MATE in Power Flow

The MATE algorithm is well documented and has been implemented in many ways ([38], [29], [4], [55]) and even generalized in [49] for analyzing infrastructure dependencies. In all cases, a system matrix is partitioned into subsystems which can then be computed independently in their respective simulation program. The goal is to apply this idea of partitioning a system matrix into subsystems to the power flow algorithm.

However there has already been much research done in the area of power flow where the concept of partitioning a system matrix is applied. The most popular method to partition a matrix into subsystems is by use of *node tearing* (discussed in section 1.2.2 and in the next paragraph) which converts the system matrix into BBDF where block elimination or more elaborate elimination algorithms are used to solve the system. These methods have been implemented for CPU clusters ([57], [57], [32], and [64]), transputers [13], and even FPGAs [58]. Single level approaches (i.e. using only one link matrix or Schur complement) gave similar results as MATE [55] where speedup began to saturate around 16 processors [32]. The multi-level or nested decomposition approaches report superior results in terms of scaling to processors compared to the single level approach (particularly in [64]). Multilevel techniques are discussed more thoroughly in chapter 4.

This chapter will set up the foundation for the MATE to be applied to power flow. The idea to accomplish this transition is to remove small *pieces* from the system matrix that will disconnect the system into parts. Tearing techniques can accomplish this by either removing a small number of branches or nodes. Hence there are intuitively two categories of tearing, branch tearing and node tearing ([21] section 11.11 and 11.12). In node tearing, once a node is removed, all branches connected to that node is also removed (further discussion is found in section 1.2.2). Branch tearing, however, can be further sub-divided into two more categories by considering branch tearing from a physical system viewpoint or just a generic matrix viewpoint. If a physical system viewpoint is taken, such as the admittance matrix of a power system, the removal of a branch will also affect the self admittance of the buses that are connected to that branch. Hence not only are pairs of off-diagonal entries being removed, but the diagonal entries are also being modified. This is the type of branch tearing employed by Diakoptics and MATE. A generalized version of physical branch tearing for use in power flow is derived in section 2.1. The second section of this chapter discusses the generic matrix oriented branch tearing. The third

section describes advantages and disadvantages of both branch tearing and node tearing techniques being applied to the Jacobian matrix from a power flow program.

2.1 Generalized MATE Link Equations

In MATE, the link equation comes naturally from modified nodal analysis as the branch current. However the link equation can also be generalized for a generic system matrix. This generalization will then pave the way to an efficient approach for unsymmetrical links. MATE as well as the generalized technique are both branch tearing techniques from a physical system point of view, thus they both modify the diagonal entries of the system matrix. As it will be shown, the most essential portion of these derivations is creating equivalent diagonal terms that are to be substituted into the system matrix.

2.1.1 Symmetrical Links

Consider the system shown in matrix form and equation form below. The system matrix is symmetrical just like the admittance matrix of an EMTP program.

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \tag{2.1}$$

$$\begin{aligned} a \cdot x_1 + b \cdot x_2 &= y_1 \\ b \cdot x_1 + c \cdot x_2 &= y_2 \end{aligned} \tag{2.2}$$

Suppose this was the admittance matrix of a two node system. To decouple the two nodes using MATE, the admittance of the branch connecting the two nodes would be removed. The removal of that branch would require removing that branch's mutual and self admittance (i.e. modifying both the off-diagonal and diagonal entries). From a purely mathematical perspective, this can be accomplished by replacing the diagonal entries a and c with the equivalent values shown in equation 2.3.

$$a = (a + b) - b \qquad c \ = (c + b) - b \tag{2.3}$$

Upon substituting equations 2.3 into the system equations 2.2, the two equations shown below are obtained. Notice that these two equations resemble that of MATE decoupling two nodes. It contains a modified diagonal as well as linking "current" term. Notice how the current term is analogous to $Y \cdot (v_1 - v_2)$. Furthermore, the linking current term is positive in the first row and negative in the second row, which is analogous to the linking current leaving one node and entering another.

$$\begin{aligned} (a + b)\, x_1 \overbrace{-bx_1 + bx_2}^{i_\alpha} &= y_1 \\ (c + b)\, x_2 \underbrace{+bx_1 - bx_2}_{-i_\alpha} &= y_2 \end{aligned} \tag{2.4}$$

From the above equation, the link equation can be extracted (see equation 2.5) and the system can then be represented with equation 2.6 or in matrix form in equation 2.7 in which decoupling is achieved.

$$i_\alpha = -bx_1 + bx_2 \qquad or \qquad x_1 - x_2 + \tfrac{1}{b} i_\alpha = 0 \tag{2.5}$$

$$\begin{aligned} (a+b)\, x_1 + i_\alpha &= y_1 \\ (c+b)\, x_2 - i_\alpha &= y_2 \end{aligned} \tag{2.6}$$

$$\begin{bmatrix} (a+b) & 0 & +1 \\ 0 & (c+b) & -1 \\ +1 & -1 & \frac{1}{b} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ i_\alpha \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ 0 \end{bmatrix} \tag{2.7}$$

Interestingly, instead of adding to the diagonals as shown in equation 2.3 one could also subtract from the diagonals as shown in equation 2.8.

$$a = (a-b) + b \qquad c \ = (c-b) + b \tag{2.8}$$

Just as before, equation 2.8 can be substituted into the system equation giving equation 2.9. However this version has much less in common with a real system as now the current terms in rows one and two are both positive, which would be analogous to current leaving both nodes. Extracting the link equation and grouping all the system equations yields the system in matrix form shown in equation 2.11. As a result of the modification on the diagonals, this variation leads to purely positive pointers shown by all the $+1$'s .

$$\begin{aligned} (a-b)\, x_1 \overbrace{+bx_1 + bx_2}^{i_\alpha} &= y_1 \\ (c-b)\, x_2 \underbrace{+bx_1 + bx_2}_{i_\alpha} &= y_2 \end{aligned} \tag{2.9}$$

$$i_\alpha = bx_1 + bx_2 \qquad or \qquad x_1 + x_2 - \tfrac{1}{b} i_\alpha = 0 \tag{2.10}$$

$$\begin{bmatrix} (a-b) & 0 & +1 \\ 0 & (c-b) & +1 \\ +1 & +1 & -\frac{1}{b} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ i_\alpha \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ 0 \end{bmatrix} \tag{2.11}$$

2.1.2 Unsymmetrical Links

Consider the following system shown in both matrix and equation form below. The system matrix is unsymmetrical, just like the Jacobian matrix in a power flow program.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \tag{2.12}$$

$$\begin{aligned} a \cdot x_1 + b \cdot x_2 &= y_1 \\ c \cdot x_1 + d \cdot x_2 &= y_2 \end{aligned} \tag{2.13}$$

An approach similar to the symmetric link equation is taken. The diagonals are again modified only now they are modified by the off-diagonal entry that is located directly below/above them as shown in equation 2.14.

$$a = (a - c) + c \qquad d \ = (d - b) + b \tag{2.14}$$

The modified diagonal values are then substituted back into system equations. By rearranging the terms, a **common linking term** can be extracted despite the system matrix being unsymmetrical which is shown below.

$$\begin{aligned} (a-c)\, x_1 \overbrace{+cx_1 + bx_2}^{i_\alpha} &= y_1 \\ (d-b)\, x_2 \underbrace{+cx_1 + bx_2}_{i_\alpha} &= y_2 \end{aligned} \tag{2.15}$$

$$i_\alpha = cx_1 + bx_2 \qquad or \qquad x_1 + x_2 - \tfrac{1}{b} i_\alpha = 0 \tag{2.16}$$

Placing these equations in matrix form results in the desired decoupling shown in equation 2.7. Although unsymmetrical decoupling of a branch was achieved with only one link equation, the link equation row now contains the floating point value $\frac{b}{c}$ instead of the simple $+1$. This can be avoided by having two separate link terms however it will of course result in twice the number link equations to decouple one unsymmetrical branch as shown in equation 2.18.

$$\begin{bmatrix} (a-c) & 0 & +1 \\ 0 & (d-b) & +1 \\ +1 & +\frac{b}{c} & -\frac{1}{c} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ i_\alpha \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ 0 \end{bmatrix} \tag{2.17}$$

$$\begin{bmatrix} (a-c) & & +1 & +1 \\ & (d-b) & +1 & +1 \\ +1 & & -\frac{1}{c} & \\ & +1 & & -\frac{1}{b} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ i_{\alpha 1} \\ i_{\alpha 2} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ 0 \\ 0 \end{bmatrix} \tag{2.18}$$

2.2 Matrix Oriented Branch Tearing

This type of branch tearing technique differs from the physical system point of view, since it does not modify the diagonal entries of the system matrix. This technique is particularly advantageous for nodes with multiple connections and will be exemplified in the next section. This section will focus on the mechanics of the technique in which the goal will be to zero two off-diagonal elements of a general system matrix.

Consider the following system with an unsymmetrical matrix shown below. Suppose the goal is to remove the elements a_{12} and a_{13}. The first row multiplied out is shown in equation 2.20.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \tag{2.19}$$

The technique's approach is to simply assign the desired entries to be decoupled to a link quantity. Thus the link equation can be extracted shown in equation 2.21. The desired decoupling is then obtain in equation 2.22.

$$a_{11}x_1 + \underbrace{a_{12}x_2 + a_{13}x_3}_{i_\alpha} = y_1 \tag{2.20}$$

$$i_\alpha = a_{12}x_2 + a_{13}x_3 \qquad or \qquad a_{12}x_2 + a_{13}x_3 - i_\alpha = 0 \tag{2.21}$$

$$\begin{bmatrix} a_{11} & 0 & 0 & 1 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & a_{12} & a_{13} & -1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ i_\alpha \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ 0 \end{bmatrix} \tag{2.22}$$

2.3 MATE in Power Flow Theory

This section will explore the possibility of applying MATE to the power flow problem, as well as its relation to the original EMTP MATE. In both applications, groups of nodes will form subsystems. In the ideal situation, the groups of nodes within the subsystems are densely connected while the subsystem to subsystem connections are weakly connected. To decouple a system requires the cutting of the subsystem to subsystem branches. It is the *quantity* that is flowing through these branches, the linking quantity, that will provide greater insight on the application of MATE in power flow. This section concludes with a discussion on various techniques, either branch tearing or node tearing, that can be employed to achieve partitioning in the power flow algorithm.

The linking quantity that connects partitioned systems in an EMTP MATE implementation can be determined by considering the system equation (see equation 2.23). Notice that the multiplication of the admittance matrix with the voltage vector results in a sum of *current terms* at each node. At a given node, each of these current terms corresponds to a branch current coming in from other nodes. Thus the act of cutting a branch is to remove its corresponding current term which is done by modifying the admittance matrix. Therefore the branch quantity is indeed current, however in general, it can be easily identified by inspecting the RHS of the system equation.

$$[Y]\,[V] = [I] \tag{2.23}$$

Using the conclusions drawn from the above EMTP discussion, the power flow linking quantity can be directly identified from the RHS of the linearized system equation (see equation 2.24); which happens to be real and imaginary mismatch power. This can be explained from the fact that a power flow program

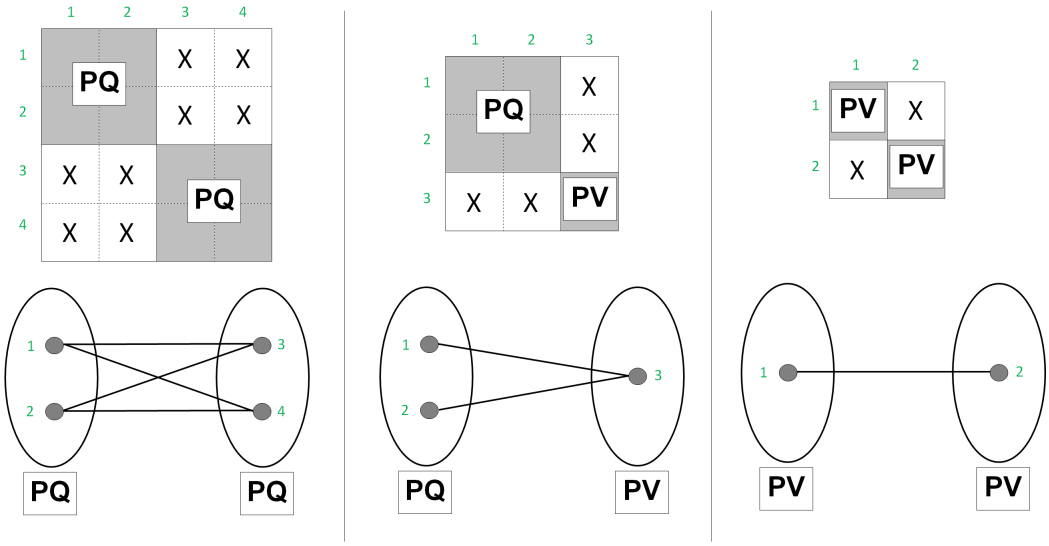


Figure 2.1: Power Flow Branch Types

starts with an initial voltage guess. Hence there will be errors between the calculated power (using the estimated voltages) and the scheduled power at a given node which is called mismatch power. Similar to the net current at each node being a sum of currents flowing through branches from neighbouring nodes, the net mismatch power at a node is the sum of the error in power flowing through branches coming from neighbouring nodes. Thus to cut a branch, would be to remove a flow of the power error from the two connecting nodes. Therefore the link quantity in power flow is the mismatch of real and imaginary power flowing through a particular branch. Notice that in power flow, the linking quantity is more of a theoretical value rather than a physical value, as is the case with EMTP.

$$[J]\,[\triangle\ V] = [\triangle\ S] \tag{2.24}$$

As explained in section 1.1 there are three types of nodes in the power flow algorithm: PQ node, PV node and slack node. Since the slack node is not present in the system's Jacobian matrix, it is left out of the discussion. The majority of a power system is made up of PQ nodes which are described by two equations whereas the remainder of the system is made up of PV nodes which take up only one equation. The difference in the number of equations for PQ and PV buses leads to three distinctly different branch connections: PQ-to-PQ, PQ-to-PV and PV-to-PV. Now consider a system matrix composed of only two nodes (either PQ or PV) that are connected together but are to be decoupled into two separate subsystems. Figure 2.1 depicts such a situation with the three different branch types. The figure also provides a graph of the system matrix based on only the non-zeros connection pattern (i.e. ignoring the fact that a PQ node is modeled by two equations). The nodes and branches in these graphs will be referred to as vertices and edges, respectively, to prevent confusion with the nodes and branches of an actual power system. The decoupling of these three types of connections can be done using either node tearing or branch tearing.

In the case of *node tearing*, the method is to simply re-number one of the two nodes border nodes to the bottom of the matrix as described in section 1.2.2. For instance, the case of a PQ-PQ connection, one

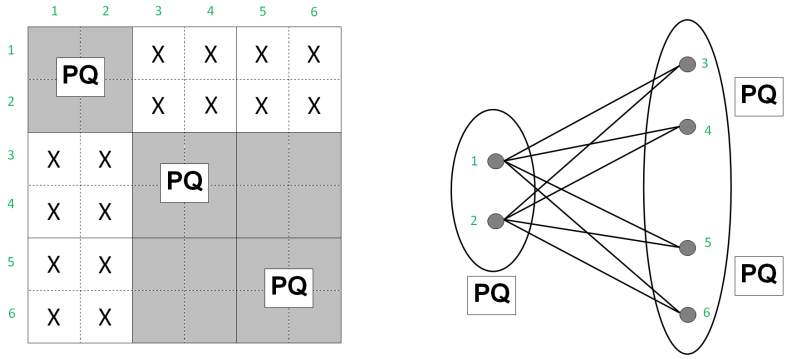


Figure 2.2: PQ-PQ-PQ Branch

of the two nodes would have its two equations moved to the bottom of the system matrix which would act as the link equations. For a PQ-PV connection, it would be advantageous to choose the PV node's single equation to be moved to the bottom, since it results in a smaller link matrix. Thus the PQ-PV and PV-PV connections are preferred to be decoupled over the PQ-PQ connection.

In the case of branch tearing, there are two choices: physical branch tearing or matrix oriented branch tearing. Certain connections will favour one method over the other as will be explained below.

Physical branch tearing (i.e. the generalized link equations) can be visualized by referring to figure 2.1 and remembering the distinction made with the vertices and edges versus the nodes and branches of a power system. The unsymmetrical version (described in section 2.1.2) can be used here since the Jacobian matrix of a conventional power flow algorithm is unsymmetrical. Thus when using the unsymmetrical link version, notice that decoupling just one PQ-PQ branch is extremely computationally expensive since the connection contains four edges (see figure 2.1), and hence four equations are necessary. Furthermore, the PQ-PV connection requires two link equations and the PV-PV connection only requires only one link equation. Therefore, when partitioning a system, decoupling PV-PV branches are significantly more favorable than decoupling PQ-PQ branch.

Matrix oriented branch tearing, described in section 2.2, can also be applied to decouple connections. This method is best used for nodes with multiple connections. For instance consider the situation shown in figure 2.2 where one PQ node is connected two PQ nodes from another subsystem. Since this method requires one link equation for every row containing entries that are to be decoupled, the system in figure 2.2 will require 6 link equations. Whereas the generalized link equations requires one link equation per edge, hence requiring a total of 8 link equations. Furthermore, the higher the number of connection a node has, the greater the discrepancy between the two types of branch tearing. However the best type of decoupling may come from node tearing. The two equations that represent the PQ node with the most connections is placed at the bottom of the system matrix. These two equations serve as the link equations. Notice this version only contains two link equations whereas the branch tearing methods required 6 to 8 link equations.

It should also be noted that matrix oriented branch tearing does perform worst for decoupling nodes with only one branch. For instance a PQ-PV requires three link equations with matrix oriented branch

tearing as oppose to two link equations with the physical branch tearing. Fortunately, the two branch tearing techniques can be used simultaneously since matrix oriented branch tearing does not modify the diagonal entries.

2.4 Summary

Node tearing as well as two different branch tearing techniques were presented in this chapter for the application of partitioning a power flow algorithm's Jacobian. Perhaps the most distinctive characteristic of the Jacobian, are the PQ nodes which take up two equations. This fact can significantly increase the number of link equations necessary to decouple nodes when using branch tearing techniques. However node tearing techniques do not suffer from the same problem as it only reorders the system equations to achieve the desired partition. The main difference of branch tearing is that the block border column is made up of ones and zeros (i.e. pointer matrices), whereas node tearing has floating point values. This results in an increase in matrix multiplication during the formation of the link matrix. However the significant increase in link equations from branch tearing would indicate node tearing to be the technique of choice for partitioning in power flow. A fully implemented version would be necessary to obtain a clear conclusion. However the projected performance of such an implementation would most likely be worst than what has already done with MATE in [56] due to the PQ nodes. The next chapter will explore some possible optimizations for the power flow algorithm with the objective of providing a better power flow platform for MATE.

Chapter 3

Power Flow Optimizations for MATE

This chapter attempts to provide various power flow algorithms with the intention of providing speedups to the MATE algorithm. Two separate programs (in addition to the base conventional program) have been implemented and performance improvements are measured against a conventional Newton-Raphson power flow program.

3.1 General Implementation Details

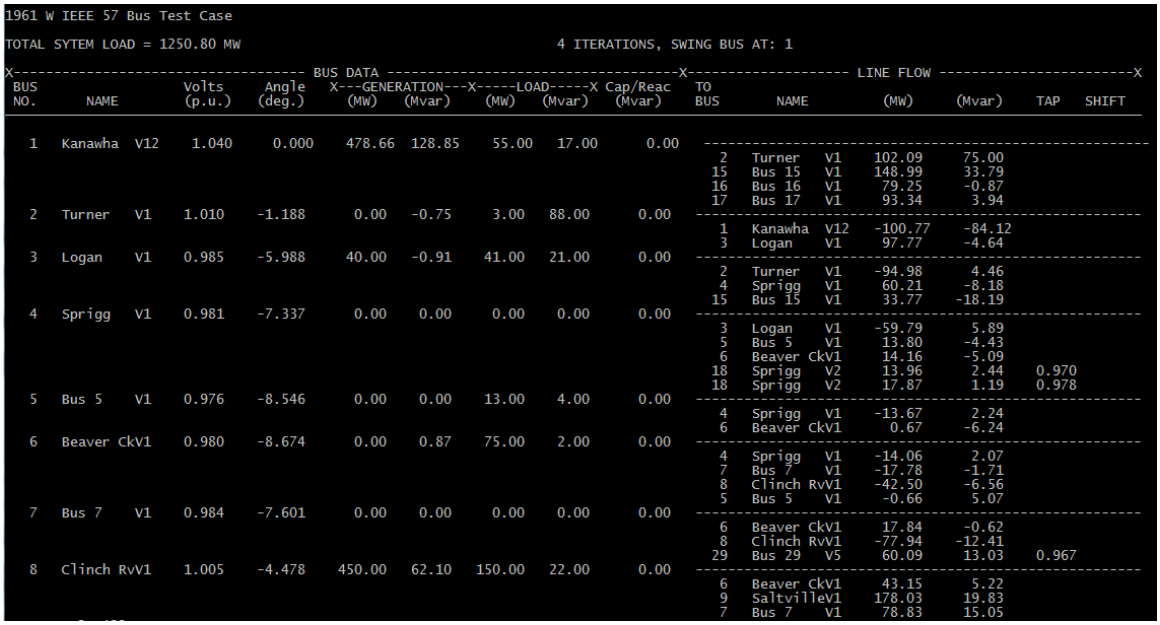
To compare the performance of each of the optimizations described in this chapter, a conventional Newton-Raphson power flow program has been implemented and will serve as the base program. Details on test cases, sparse matrix solver and execution times are provided in this section.

3.1.1 Test Cases

All implemented programs used power flow test cases provided from the University of Washington [45]. The Common Data Format (described in [48]) was used by all programs. The IEEE 118 and IEEE 300 buy systems will be used as a means of determining performance improvements. The IEEE 118 bus test system represents a portion of the American Electric Power System (in the Midwestern US) as of December, 1962 [45]. The IEEE 300 bus test case was created by the IEEE Test Systems Task Force in 1993 [45]. These two test cases are summarized in table 3.1.

Table 3.1: Test Case Summary

	IEEE 118	IEEE 300
# of branches	186	411
branches to nodes ratio	1.58	1.37
% of PQ nodes	54%	77%
% of PV nodes	45%	23%
Jacobian Size	181x181	530x530
# of non-zeros in Jacobian	1051	3736
Jacobian Matrix Sparsity	96.8%	98.7%



```
1961 W IEEE 57 Bus Test Case

TOTAL SYTEM LOAD = 1250.80 MW                                4 ITERATIONS, SWING BUS AT: 1

X---------------------------------- BUS DATA ----------------------------------X-------------------- LINE FLOW ----------------------X
 BUS                 Volts    Angle   X---GENERATION---X-----LOAD-----X Cap/Reac   TO
 NO.      NAME       (p.u.)   (deg.)    (MW)    (Mvar)    (MW)    (Mvar)   (Mvar)  BUS      NAME         (MW)     (Mvar)    TAP    SHIFT
-------------------------------------------------------------------------------------------------------------------------------------
   1   Kanawha  V12   1.040     0.000   478.66   128.85    55.00    17.00     0.00    2   Turner   V1    102.09     75.00
                                                                                    15   Bus 15   V1    148.99     33.79
                                                                                    16   Bus 16   V1     79.25     -0.87
                                                                                    17   Bus 17   V1     93.34      3.94
   2   Turner   V1    1.010    -1.188     0.00    -0.75     3.00    88.00     0.00    1   Kanawha  V12  -100.77    -84.12
                                                                                     3   Logan    V1     97.77     -4.64
   3   Logan    V1    0.985    -5.988    40.00    -0.91    41.00    21.00     0.00    2   Turner   V1    -94.98      4.46
                                                                                     4   Sprigg   V1     60.21     -8.18
                                                                                    15   Bus 15   V1     33.77    -18.19
   4   Sprigg   V1    0.981    -7.337     0.00     0.00     0.00     0.00     0.00    3   Logan    V1    -59.79      5.89
                                                                                     5   Bus 5    V1     13.80     -4.43
                                                                                     6   Beaver CkV1     14.16     -5.09
                                                                                    18   Sprigg   V2     13.96      2.44    0.970
                                                                                    18   Sprigg   V2     17.87      1.19    0.978
   5   Bus 5    V1    0.976    -8.546     0.00     0.00    13.00     4.00     0.00    4   Sprigg   V1    -13.67      2.24
                                                                                     6   Beaver CkV1      0.67     -6.24
   6   Beaver CkV1    0.980    -8.674     0.00     0.87    75.00     2.00     0.00    4   Sprigg   V1    -14.06      2.07
                                                                                     7   Bus 7    V1    -17.78     -1.71
                                                                                     8   Clinch RvV1    -42.50     -6.56
                                                                                     5   Bus 5    V1     -0.66      5.07
   7   Bus 7    V1    0.984    -7.601     0.00     0.00     0.00     0.00     0.00    6   Beaver CkV1     17.84     -0.62
                                                                                     8   Clinch RvV1    -77.94    -12.41
                                                                                    29   Bus 29   V5     60.09     13.03    0.967
   8   Clinch RvV1    1.005    -4.478   450.00    62.10   150.00    22.00     0.00    6   Beaver CkV1     43.15      5.22
                                                                                     9   SaltvilleV1    178.03     19.83
                                                                                     7   Bus 7    V1     78.83     15.05
```

Figure 3.1: Sample Generated Power Flow Output

3.1.2 Sparse Solver

The sparse matrix solver chosen for factorization of the Jacobian matrix is SuperLU. SuperLU is a direct sparse matrix solver optimized for unsymmetrical matrices. It uses a left-looking supernodal approach to the LU decomposition algorithm described in [18]. All programs in this chapter use this solver.

The factorization in the power flow program is described as follows. The first iteration performs ordering, symbolic factorization, numerical factorization and backward/forward substitution. All subsequent iterations reuse the ordering pattern and elimination tree that came from symbolic factorization thus leaving only the numerical factorization and backward/forward substitution to be performed. Thus the first iteration will take substantially more time to complete than the other iterations.

3.1.3 Execution Times

To verify the correct output of the program, the output of the Gauss-Seidel and current equation power flow programs have been checked against the results of the base power equation PF program for correctness. All simulations in this chapter begin with a flat start. A sample output of the base program is provided in figure 3.1.

All simulations were performed in Linux using the GNU g++ compiler on an Intel Core 2 Duo E7400 @ 2.8GHz with 4 GB of RAM. To simplify the comparison between load flows, control equations are turned off and PV buses were not allowed to change to PQ if its generator VAR limits are exceeded. A summary of the number of iterations required to converge to various specified mismatch tolerances are provided in table 3.2 and execution times, averaged over many iterations, are in table 3.3. A more detailed analysis can be found in appendices A and B. All performance improvements in this chapter will be with respect to these benchmarks.

Table 3.2: Number of Iterations Required to Reach Convergence

Mismatch Tolerance	Iterations for 118	Iterations for 300
1e-3	3	4
1e-5	3	4
1e-6	4	5

Table 3.3: Execution Times (ms) for a 0.001 Mismatch Tolerance

Iteration	118 Bus	300 Bus
1	0.415	1.261
2	0.226	0.781
3	0.225	0.780
4	-	0.781
Total	**0.68**	**3.60**

3.2 Current Equation Power Flow Program

This method uses the current equation instead of the regular power equation to derive the PF algorithm. A recent improvement to the PV bus representation has led to the re-surfacing of the algorithm [16]. Later in [24], the method was improved, however gave a few contradictory results to the original paper. This section will investigate several possible implementations of the current equation algorithm as well providing a proposed version. Execution times are then compared against a conventional power flow program described in section 3.1.

3.2.1 Notation

The following notation will be used when deriving the current equation algorithm:

Variables	Description
$e_k + j \cdot f_k$	Complex voltage at bus k (rectangular)
$V_k \angle \delta$	Complex voltage at bus k (polar)
$\Delta P_k + j \cdot \Delta Q_k$	Complex mismatch power at bus k
$P_k^{sch} + j \cdot Q_k^{sch}$	Net complex scheduled power at bus k
$P_k^{calc} + j \cdot Q_k^{calc}$	Net complex scheduled power at bus k
$\Delta I_k^R + j \cdot \Delta I_k^M$	Complex mismatch current at bus k
$G_{km} + j \cdot B_{km}$	Admittance of branch k-m

3.2.2 Literature Review

Load flow programs can be derived from either[1] using the power equation ($S = VI^*$) or the current equation ($I = \frac{S^*}{V^*}$) with the voltage represented in either rectangular or polar coordinates. Upon linearizing the two types of equations (using the conventional polar coordinates for the voltage in the power equation and rectangular coordinates for the voltage in the current equation) we would obtain the following:

$$\begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} = \begin{bmatrix} \quad J \quad \end{bmatrix} \cdot \begin{bmatrix} \Delta V \\ \Delta \delta \end{bmatrix} \qquad \begin{bmatrix} \Delta I^R \\ \Delta I^M \end{bmatrix} = \begin{bmatrix} \quad Y^* \quad \end{bmatrix} \cdot \begin{bmatrix} \Delta e \\ \Delta f \end{bmatrix} \tag{3.1}$$

Notice that the current equation version contains real and imaginary current mismatches instead of power mismatches.

The benefit of using the current equation (using rectangular coordinates for voltage) comes from the result of linearizing the system of equations (i.e. applying Newton's method) where most of the off-diagonal entries in the Jacobian are simply equal to bus admittances. Thus the majority of the Jacobian is constant. However the problem with the current equation load flow (using rectangular coordinates for voltage) was the fact that it required knowledge of the scheduled/final reactive power at all buses in the system during the solving process. This isn't a problem for the PQ node (since both P and Q are specified) however for a PV node, the Q^{sch} is determined after the voltages in the system have been solved for.

The current equation algorithm was originally proposed by [20] in 1970. The PV bus was represented using one power mismatch equation (ΔP) where the voltage was represented in rectangular form. Despite the exclusion of the Q mismatch equation, it still performed reasonably well for difficult lightly loaded systems [51].

More recently in 1999 a paper was published ([16]) proposing a new PV bus representation which contained both P and Q mismatch values along with a third equation relating the real and imaginary voltage to the voltage magnitude (shown in equation 3.2). This third equation is necessary for there to be enough equations to solve for the newly introduced Q mismatch variables at PV buses.

$$V_k^2 = e_k^2 + f_k^2 \tag{3.2}$$

Upon linearizing the above equation with respect to e_k and f_k we obtain the following:

$$\Delta V_k = 0 = \frac{e_k}{V_k}\Delta e_k + \frac{f_k}{V_k}\Delta f_k \tag{3.3}$$

Since we're dealing with PV buses, V_k is specified and thus constant. The authors used this fact to simplify equation 3.3 by setting the mismatch in voltage magnitude (ΔV_k) equal to zero.

A few years later, it was realized in [24] that setting ΔV_k to zero was only valid after convergence

[1] A voltage equation power flow is also possible however it would utilize the impedance matrix (which is dense) as oppose to the sparse admittance matrix

had been reached. But since the original formulation assumed a zero ΔV_k *during the solution procedure*, this approximation led to a slower convergence trajectory. This is both explained in words and shown in table 1 of [24], which was in contrast to the original paper ([16]) whose results claimed to have identical convergence to the conventional power equation power flow algorithm. The revision made to the third equation of the PV bus representation was to calculate the voltage magnitude mismatch using equation 3.4 instead of setting it to zero.

$$\Delta V_k = V_k^{spec} - V_k^{calc} \qquad where \qquad V_k^{calc} = \sqrt{e_k^2 + f_k^2} \tag{3.4}$$

This correction was then tested on a balanced three-phase version on the IEEE 14 bus system at loading factors ranging from 0.5 to 4.1 p.u. of the base case. It was also mentioned that the previous version did not have identical convergence to the conventional program, however with the new improvement they are able to achieve exact convergence with the conventional version. This method has also been modified for control adjustments/FACTS [59, 17], then later modified for distribution systems [25] (balanced three phase [23] and four conductor [46] load flow).

3.2.3 Expected Speed Up

By taking a closer look at the anatomy of the conventional power flow program we can determine an upper bound to the potential performance improvement of the current equation version empirically.

The current equation algorithm has the potential to provide a speed up in two ways: (1) much of the Jacobian does not need to be updated, and (2) the use of rectangular coordinates avoids the need of sine and cosine functions[2]. Thus the speedup cannot be more than if the execution times of those two operations were neglected (assuming identical convergence trajectory).

To gain a better understanding of this upper bound, the implemented conventional load flow program (detailed in section 3.1) was profiled for 118 bus system and the 300 bus system. There are two different types of iterations: the first iteration and then all subsequent iterations. The difference between them is the first iteration requires both ordering and symbolic factorization, whereas the subsequent iterations do not. The results are shown in tables 3.4 and 3.5, the detailed results are found in appendix B.

From these results we see factorization process takes between 75% to 88% of the execution time. This value is corroborated by [31] which claims ~85% of a power flow program is spent performing factorization. Furthermore looking at time taken to compute both the Jacobian and sine & cosine tasks, they only contribute between 7% to 14% of the execution time which equates to the upper limit of the improved current equation algorithm.

It should also be noted that although the majority current equation's Jacobian matrix do not require computation, the diagonals do require updating have formulas that are quite long (example of a diagonal entry in equation 3.5, the rest are found in the appendix of [16]). These saving are further reduced when

[2]PV bus calculations **do** require the use of transcendental functions (square root and arc-tangent) but are shown to be linearized in [16]

Table 3.4: Conventional Program Profiling for 118 Bus

Task	First Iteration		Subsequent Iterations	
	Time (us)	Percentage	Time (us)	Percentage
ΔP & ΔQ	23.7	5.7%	23.7	10%
Jacobian	24.0	5.8%	24.0	11%
SuperLU	**360**	**86%**	**171**	**75%**
Voltage Update	1.52	0.4%	1.55	0.6%
Sine & Cosine	5.35	1.3%	6.12	2.7%

Table 3.5: Conventional Program Profiling for 300 Bus

Task	First Iteration		Subsequent Iterations	
	Time (us)	Percentage	Time (us)	Percentage
ΔP & ΔQ	56.4	4.5%	56.8	7.3%
Jacobian	80.8	6.4%	80.8	10%
SuperLU	**1107**	**88%**	**625**	**80%**
Voltage Update	3.33	0.3%	3.32	0.4%
Sine & Cosine	14.1	1.1%	14.7	1.9%

the additional factorization time of the larger Jacobian matrix is taken into account. The larger Jacobian matrix is due to the additional equations of the newly developed PV bus representation.

$$B_{kk}^{'} = B_{kk} - \frac{\left(e_k^2 - f_k^2\right) \cdot Q_k^{sch} - 2 \cdot e_k \cdot f_k \cdot P_k^{sch}}{\left(e_k^2 + f_k^2\right)^2} \tag{3.5}$$

In contrast to the results of the analysis presented here, [16] claimed an average speed up of 20% (later claiming 30% in [23]) when compared to an industry grade power flow program. They also noted that they used the same factorization code as the production grade program. However, this speed up was cited for the original formulation, which was before the improvement in [24] and thus may not be reliable. Since then, the current equation algorithm was modified for distribution systems and was never compared to the conventional power flow after the improvement to the PV bus representation was made.

3.2.4 Implementations

The current equation algorithm was implemented as outlined in [16] however using the improvements mentioned in [24]. This was implemented in two different ways, the expanded form and the condensed form. The difference between them is just the algebraic representation of the third PV bus equation.

3.2.4.1 Expanded Form

This version was first used in the paper with the improved formulation [24]. This form simply retains all 3 linearized equations to represent a PV bus, however moves the third row and column (shown in equation 3.6) to end of the system's Jacobian matrix which results in the equation 3.7. In this equation, Z and X represent the contribution of the third row and column terms of all PV buses in the system.

$$\begin{bmatrix} \frac{f_k \cdot \Delta P_k}{e_k^2+f_k^2} \\ \frac{e_k \cdot \Delta P_k}{e_k^2+f_k^2} \\ \Delta V_k \end{bmatrix} = \begin{bmatrix} B'_{kk} & G'_{kk} & \frac{e_k}{e_k^2+f_k^2} \\ G''_{kk} & B''_{kk} & -\frac{f_k}{e_k^2+f_k^2} \\ \frac{e_k}{V_k} & \frac{f_k}{V_k} & 0 \end{bmatrix} \cdot \begin{bmatrix} \Delta e_k \\ \Delta f_k \\ \Delta Q_k \end{bmatrix} \tag{3.6}$$

$$\begin{bmatrix} \triangle\, I_m \\ \triangle\, I_r \\ \triangle\, V \end{bmatrix} = \left[\begin{array}{c|c} J & X \\ \hline Z & \end{array}\right] \begin{bmatrix} \triangle\, e \\ \triangle\, f \\ \triangle\, Q \end{bmatrix} \tag{3.7}$$

It was confirmed by the author (Dr. Paulo Garcia), in a conversation at the UBC power lab in January of 2009, that the expanded form was indeed the way it was implemented in their code[3]. Although the Z and X terms increased the size of the Jacobian, Dr. Garcia made references to how the sparsity of the matrix was increased and argued it would be faster than a condensed formulation which is explained in the next section. The end result is a system matrix with 2 equations for PQ buses and **3 equations for PV buses**.

3.2.4.2 Condensed Form

This form was used in the original derivation ([16]) of the current equation however it has not been derived for the improved version ([24]) since it was believed to be slower than the expanded form. This section will derive the condensed form using the improved version for the first time.

Starting with the 3 linearized PV equations at bus 'k' we have:

$$\begin{bmatrix} \frac{f_k \cdot \Delta P_k}{e_k^2+f_k^2} \\ \frac{e_k \cdot \Delta P_k}{e_k^2+f_k^2} \\ \Delta V_k \end{bmatrix} = \begin{bmatrix} B'_{kk} & G'_{kk} & \frac{e_k}{e_k^2+f_k^2} \\ G''_{kk} & B''_{kk} & -\frac{f_k}{e_k^2+f_k^2} \\ \frac{e_k}{V_k} & \frac{f_k}{V_k} & 0 \end{bmatrix} \cdot \begin{bmatrix} \Delta e_k \\ \Delta f_k \\ \Delta Q_k \end{bmatrix} \tag{3.8}$$

Since the third row is only dependent on two state variables, we can eliminate one of those two state variables by solving for Δe_k using the third equation of 3.8 shown below:

$$\Delta e_k = \frac{V_k}{e_k}\Delta V_k - \frac{f_k}{e_k}\Delta f_k \tag{3.9}$$

With this formula we can multiply the first column of equation 3.8 to obtain equation 3.10 which has the same diagonal Jacobian matrix as [16] however has an additional term on the LHS to reflect the improvement version of [24].

$$\begin{bmatrix} \frac{f_k \cdot \Delta P_k}{e_k^2+f_k^2} - \left(B'_{kk} \cdot \frac{V_k}{e_k}\Delta V_k\right) \\ \frac{e_k \cdot \Delta P_k}{e_k^2+f_k^2} - \left(G''_{kk} \cdot \frac{V_k}{e_k}\Delta V_k\right) \end{bmatrix} = \begin{bmatrix} \left(G'_{kk} - \frac{B'_{kk} \cdot f_k}{e_k}\right) & \frac{e_k}{e_k^2+f_k^2} \\ \left(B''_{kk} - \frac{G''_{kk} \cdot f_k}{e_k}\right) & -\frac{f_k}{e_k^2+f_k^2} \end{bmatrix} \cdot \begin{bmatrix} \Delta e_k \\ \Delta f_k \end{bmatrix} \tag{3.10}$$

[3]Their paper implemented a three phase load flow, thus it is not exactly a direct comparison

Table 3.6: Comparisons of Execution Times (Tolerance of 1e-3)

Jacobian Details	Conventional PF		Expanded		Condensed	
System	118	300	118	300	118	300
First Iteration (ms)	0.42	1.26	0.73	1.61	0.61	1.63
Subsequent Iterations (ms)	0.23	0.78	0.52	1.17	0.44	1.15
# of iterations to converge	**3**	**4**	**9**	**7**	**9**	**7**
Total Time (ms)*	0.68	3.60	4.91	8.64	4.09	8.55

*Extrapolated from first and subsequent iteration timings

However, the elimination of Δe_k does not just affect bus 'k', it will also affect all buses connected to bus 'k' (i.e. all rows in the Jacobian that contain a Δe_k term). Thus in general, the LHS formulas for a bus 'k' affected by all PV buses 'm' are shown below for both the PQ bus equations 3.11 and the PV bus in equations 3.12.

$$\begin{aligned} LHS^M &= \Delta I_k^M - \sum_{m \in PV} B_{km} \cdot \tfrac{V_m}{e_m} \Delta V_m \\ LHS^R &= \Delta I_k^R - \sum_{m \in PV} G_{km} \cdot \tfrac{V_m}{e_m} \Delta V_m \end{aligned} \tag{3.11}$$

$$\begin{aligned} LHS^M &= \tfrac{f_k \cdot \Delta P_k}{e_k^2 + f_k^2} - \left(B'_{kk} \cdot \tfrac{V_k}{e_k} \Delta V_k \right) - \sum_{m \in PV} B_{km} \cdot \tfrac{V_m}{e_m} \Delta V_m \\ LHS^R &= \tfrac{e_k \cdot \Delta P_k}{e_k^2 + f_k^2} - \left(G''_{kk} \cdot \tfrac{V_k}{e_k} \Delta V_k \right) - \sum_{m \in PV} G_{km} \cdot \tfrac{V_m}{e_m} \Delta V_m \end{aligned} \tag{3.12}$$

The end result is a Jacobian matrix with 2 equations for PQ nodes and 2 equations for PV nodes.

3.2.4.3 Results

Programs were implemented as described in [16, 24] and also in section 3.2.5.1.

Timing results are shown in table 3.6 which indicate very poor performance due primarily due to the large number of iterations. An explanation for the large number of iterations is found in section 3.2.5.1. It should also be noted that the 118 bus system does particularly poorly because it has a large number of PV nodes (45% were PV buses). Both formulations appear to have similar speeds, where the condensed version performing slightly faster than the expanded (20% for 118 and 1.1% for 300).

Supposing that the current equation PF converged in the same number of iterations as the conventional PF as mentioned in ([24]), each iteration still takes longer to finish than the conventional PF program. Further analysis of an iteration's tasks are shown in tables 3.7 and 3.8. As expected the time to build the Jacobian matrix is reduced and the cosine and sine calculations are no longer present. However there is a drastic increase in factorization time due to the increase in the size of the Jacobian caused by the PV bus representation.

There is a significant difference where the condensed and expanded forms spend their computational time. Tables 3.7 and 3.8 show that percentage wise, the condensed form spends far less time performing factorization then the expanded form. Indicating that if we had a very fast solver, the expanded form would benefit the most from solver.

Table 3.7: Expanded Form Profiling of 300 Bus

Task	First Iteration		Subsequent Iterations	
	Time (us)	Percentage	Time (us)	Percentage
ΔI	61.9	3.8%	61.7	5.3%
Jacobian	45.0	2.8%	48.2	4.1%
SuperLU	**1498**	**93%**	**1056**	**90%**
Voltage Update	3.47	0.22%	3.57	0.30%

Table 3.8: Condensed Form Profiling of 300 Bus

Task	First Iteration		Subsequent Iterations	
	Time (us)	Percentage	Time (us)	Percentage
ΔI	304	19%	307	27%
Jacobian	50.5	3.1%	53.7	4.7%
SuperLU	**1268**	**78%**	**789**	**68%**
Voltage Update	3.86	0.3%	3.86	0.3%

The timing results are further validated once we analyze the characteristics of the Jacobian matrix. The data summarized in table 3.9 is very important for understanding the poor performance. It outlines an increase in the number of non-zeros from 12% to 101%, which results in an increase in factorization from 17.6% to 178%. Thus the main culprit of the slow performance comes the larger Jacobian matrix (which was a result of the additional PV bus equations).

3.2.5 Proposed Algorithm

This section proposes a possible modification to the PV bus representation however it is possible that this was how current equation algorithm was implemented in [24]. However these implementation details where never explicitly specified.

The method takes the standard approach in applying Newton's method, however it chooses Q^{sch}

Table 3.9: Factorization Analysis & Comparisons

Jacobian Details	Conventional PF		Expanded		Condensed	
System	118	300	118	300	118	300
Jacobian Size	181x181	530x530	287x287	667x667	234x234	598x598
# of non-zeros	1051	3736	2117	4800	1486	4192
#of required updates	n/a	n/a	680	1468	834	1464
Jacobian Sparsity	96.8%	98.7%	97.4%	98.9%	97.3%	98.8%
% incr. non-zeros*	n/a	n/a	101%	**28.5%**	41%	**12%**
% incr. in fact. 1†	n/a	n/a	89%	35%	35%	17.6%
% incr. in fact. 2 ‡	n/a	n/a	178%	**70%**	46%	**25%**

* Increase with respect to conventional power equation PF

† Increase with respect to conventional PF factorization time for the first iteration

‡ Increase with respect to conventional PF factorization time for subsequent iterations

as the state variable instead of ΔQ. At a PV bus, we have 3 unknowns (e, f, Q^{sch}) thus we need 3 equations. Just as before, there are 3 non-linear equations for each PV bus:

$$\begin{aligned} \Delta I_k^M &= I_k^{M(sch)} - I_k^{M(calc)} \\ \Delta I_k^R &= I_k^{R(sch)} - I_k^{R(calc)} \\ \Delta V_k &= V_k^{sch} - V_k^{cacl} \end{aligned} \tag{3.13}$$

The proposed modification of this thesis uses a standard Newton's method approach to solve the system of non-linear equations. This section will first describe the original PV bus derivation followed by the proposed representation. The section concludes with a comparison of the previous and proposed representations and benchmarks against the conventional PF program.

3.2.5.1 Original PV Bus Derivation

In the original derivation, it appears that the 3 PV bus equations were linearized with respect to **only 2 state variables**, which were the real and imaginary voltage (e_k & f_k). Thus upon linearization we obtain the following:

$$\begin{bmatrix} \Delta I_k^M \\ \Delta I_k^R \\ \Delta V_k \end{bmatrix} = \begin{bmatrix} B_{kk}^{'} & G_{kk}^{'} \\ G_{kk}^{''} & B_{kk}^{''} \\ \frac{e_k}{V_k} & \frac{f_k}{V_k} \end{bmatrix} \cdot \begin{bmatrix} \Delta e_k \\ \Delta f_k \end{bmatrix} \tag{3.14}$$

Where $B_{kk}^{'}$ and other elements in the matrix are the resulting derivatives of the 3 equations (full derivation is covered in [16] and [24]). It can also be shown that ΔI_k^M and ΔI_k^R can be expanded as follows:

$$\begin{bmatrix} \frac{f_k \cdot \Delta P_k - e_k \cdot \Delta Q_k}{e_k^2 + f_k^2} \\ \frac{e_k \cdot \Delta P_k + f_k \cdot \Delta Q_k}{e_k^2 + f_k^2} \\ \Delta V_k \end{bmatrix} = \begin{bmatrix} B_{kk}^{'} & G_{kk}^{'} \\ G_{kk}^{''} & B_{kk}^{''} \\ \frac{e_k}{V_k} & \frac{f_k}{V_k} \end{bmatrix} \cdot \begin{bmatrix} \Delta e_k \\ \Delta f_k \end{bmatrix} \tag{3.15}$$

Since ΔQ_k is not available at PV buses the authors of [16] proposed to introduce ΔQ_k as a state variable and move the ΔQ_k quantities in the LHS vector (of equation 3.15) into the Jacobian as shown below:

$$\begin{bmatrix} \frac{f_k \cdot \Delta P_k}{e_k^2 + f_k^2} \\ \frac{e_k \cdot \Delta P_k}{e_k^2 + f_k^2} \\ \Delta V_k \end{bmatrix} = \begin{bmatrix} B_{kk}^{'} & G_{kk}^{'} & \frac{e_k}{e_k^2 + f_k^2} \\ G_{kk}^{''} & B_{kk}^{''} & -\frac{f_k}{e_k^2 + f_k^2} \\ \frac{e_k}{V_k} & \frac{f_k}{V_k} & 0 \end{bmatrix} \cdot \begin{bmatrix} \Delta e_k \\ \Delta f_k \\ \Delta Q_k \end{bmatrix} \tag{3.16}$$

It is this operation that is believed to be reason for the slow convergence observed in section 3.2.4.3 since it deviates from Newton's method.

After solving equation 3.16, the state variables can be updated by the correction terms as follows:

$$\begin{aligned} e_k^{new} &= e_k + \Delta e_k \\ f_k^{new} &= f_k + \Delta f_k \\ \Delta Q_k^{new} &= \Delta Q_k \end{aligned} \tag{3.17}$$

The ΔQ_k update was not specified in the paper, however the implemented version in this thesis *assumed* ΔQ_k to be updated as shown in equation 3.17. The ΔQ_k value is necessary at PV buses to determine Q_k^{sch} for the diagonal term of the Jacobian. Later it was discussed by the authors of [16] via email that they never updated ΔQ_k during the iterative process. Further details were not revealed but may explain the strange results in convergence found in section 3.2.4.3.

It should also be noted that the improved version ([24]) recognized that the third column of the Jacobian in equation 3.16 were the partial derivatives of the current equation mismatches with respect to *reactive power*. However [24] never specified whether they were referring to Q_k^{sch} or Q_k^{calc} or ΔQ_k. It is also unknown how the ΔQ_k update was handled. Thus it appears that [24] used an unknown third state variable to linearize the PV bus equations. Finally, there was also no explanation why the LHS was full at all buses instead of the partial formulas shown in equation 3.16 for PV buses. It was assumed that this was used to simplify notation, as the paper referenced the original algorithm [16] which used the partial mismatches shown in equation 3.16.

3.2.5.2 Proposed PV Bus Derivation

The proposed/clarified[4] version is to simply apply Newton method to all the PV bus equations. Since there are 3 non-linear equations with 3 unknowns, one can just linearize all 3 equations with respect to all 3 state variables. However we can already determine the Q^{calc} portion of ΔQ_k, the real unknown is Q_k^{sch}, thus we make it the third state variable.

$$\Delta Q_k = Q_k^{sch} - Q_k^{calc} \tag{3.18}$$

Then it is just a matter of linearizing all three equations with respect to e_k , f_k and Q_k^{sch}. Upon doing so, one would obtain the following linearized system:

$$\begin{bmatrix} \Delta I_k^M \\ \Delta I_k^R \\ \Delta V_k \end{bmatrix} = \begin{bmatrix} B'_{kk} & G'_{kk} & \frac{e_k}{V_k^2} \\ G''_{kk} & B''_{kk} & -\frac{f_k}{V_k^2} \\ \frac{e_k}{V_k} & \frac{f_k}{V_k} & 0 \end{bmatrix} \cdot \begin{bmatrix} \Delta e_k \\ \Delta f_k \\ \Delta Q_k^{sch} \end{bmatrix} \tag{3.19}$$

Coincidentally, the change in the state variable still results in the same Jacobian matrix as before. However, the LHS has changed since there is now the regular full current mismatch equations (much like the power mismatches) whereas the previous formulation only had partial current mismatches. Lastly, it is important to take note that ΔQ_k^{sch} is not the same as ΔQ_k. ΔQ_k^{sch} is the *scheduled reactive power* correction whereas $\Delta Q_k = Q_k^{sch} - Q_k^{calc}$.

[4] Bearing in mind the comments made at the end of section 3.2.5.1, this may just be a clarification

Table 3.10: Execution Times for 118 Bus System Using Condensed Current Equation PF

Tolerance	Iterations		Time (ms)		Speedup
	conventional	current eqn	conventional	current eqn	
1e-3	3	3	0.87	1.48	-1.7x
1e-5	4	4	0.87	1.92	-2.2x
1e-6	4	4	1.09	1.91	-1.8x

Table 3.11: Execution Times for 300 Bus System Using Condensed Current Equation PF

Tolerance	Iterations		Time (ms)		Speedup
	conventional	current eqn	conventional	current eqn	
1e-3	4	4	3.60	5.09	-1.4x
1e-5	4	5	3.60	6.24	-1.7x
1e-6	5	5	4.39	6.24	-1.4x

After solving equation 3.19, we obtain correction terms which are used to update the state variables as shown below:

$$\begin{aligned} e_k^{new} &= e_k + \Delta e_k \\ f_k^{new} &= f_k + \Delta f_k \\ Q_k^{sch} &= Q_k^{sch} + \Delta Q_k^{sch} \end{aligned} \tag{3.20}$$

It was found that this implemented version had a better convergence characteristic which is described in the next section.

3.2.6 Results

Convergence is much better and we are able to achieve the same convergence as the conventional PF program (see in tables 3.10 and 3.11) for a precision of 1e-3. However using a mismatch tolerance of 1e-5, we find that the current equation PF takes one iteration longer. Thus the convergence between the conventional and current equation PF programs is still not exactly equivalent.

The speedups in tables 3.10 and 3.11 are negative, indicating the current equation took longer to execute than the conventional program. Since each iteration takes longer to solve, the only way for it to be faster is if it took less iterations.

It is possible that the improved version in [24] implemented their program as similar to the version presented here however the specific details are not given in their paper. Even if this was the case, the results of the 118 bus and 300 bus test systems do not corroborate any speedup, much less the 20% speedup found in [16]. The main reason for the poor performance was found to be the larger Jacobian matrix of the current equation, which leads to longer factorization times (see table 3.9).

The current equation algorithm was originally proposed by Dr. Hermann Dommel in 1970 [20]. In a discussion with Dr. Dommel in March 2010, Dr. Dommel thought it was doubtful that a current equation PF program could out perform a conventional PF program by 20% to 30% in as stated in [16]

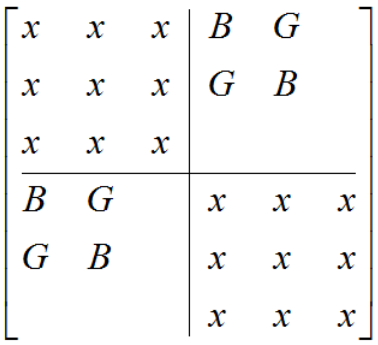


Figure 3.2: Two PV Buses Using 3x3 Blocks

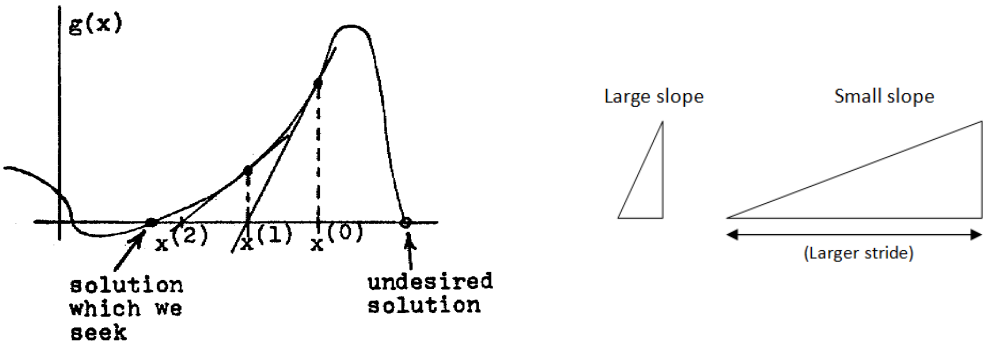
and [23]. He agreed that the majority (~85%) of the PF program is spent during the factorization phase which means only a small gain could be obtained from the formation of the Jacobian. Furthermore, having to represent PV buses with three equations as oppose to one would lead to a noticeable increase in non-zeros which would increase the factorization time. Once the additional equations to represent the PV buses are taken, the 20% speedup would seem very unlikely.

3.2.7 Summary

[16, 24] presented a novel approach to representing PV buses using three nodal equations. It was shown in section 3.2.6 that it is possible to get near identical convergence with the conventional power equation load flow program. However the effect of using 3 equations (instead of the usual one equation) to represent a PV bus leads to a larger Jacobian matrix which leads to a significant increase in factorization time. The results from the iteration profiling in section 3.2.4.3 indicates that the savings from the reduction in computation time of building the Jacobian matrix is not justified by increase in factorization time. Especially once one considers that factorization time accounts of ~85% of an iteration whereas creating a Jacobian accounts for only ~10% of an iteration.

The expanded form may present a more suitable host for the MATE algorithm since it maintains symmetry in the off-diagonal blocks and it spends most of its time performing factorization. It would be best to adopt 3x3 PV blocks rather than placing the third rows and columns of the PV bus diagonal at the ends of the of Jacobian matrix. This will ensure the PV buses do not interfere with the MATE operations. An example is given in figure which depicting the non-zero structure of 2 PV buses in a Jacobian. Notice that symmetry is maintained in the off-diagonals hence reducing link computation. However, due to the poor performance of current equation algorithm, it would seem that the current equation is best avoided.

It should be noted that the current equation algorithm would provide no obvious benefits when using node tearing as oppose to branch tearing. The reason being is that the border quantities are assumed to be non-zero, thus using the unsymmetrical link formula detailed in section 2.1.2 would have the same performance as symmetric link formula. In this case there would be no reason to use the current equation over the conventional PF program.



(a) Newton's Method

(b) Slope Sizes

Figure 3.3: Constant Jacobian Concept

(fig. (a) extracted from: H.W. Dommel. Course notes. "EECE 459 - Computer Applications in Power Systems", 1995.)

3.3 Constant Jacobian

This method calculates the Jacobian matrix once, then is kept constant for all subsequent iterations. This is a standard technique used to accelerate Newton's methods [53].

3.3.1 Theory

The basic theory behind using a constant Jacobian can be explained as follows. Newton's Method is a series of linear approximations used to determine the roots of some non-linear function. The first linearization (or slope calculation) takes place at some initial point, a guess, reasonably close to the desired root. This slope is is then used to calculate the next successive point closer to root until solution meets some desired tolerance. However at each new point, the function is re-linearized. In general, the first slope calculated will be the largest due to the distance from the root and the curvature of the non-linear function. Thus as we approach the root, the slope becomes smaller and thus takes larger strides (proportional to the height of the function) are taken towards the root. See figure 3.3which better illustrates the concept.

When using a constant Jacobian, the slope calculated at the first iteration is re-used for all subsequent iterations. As a result, we lose the quadratic convergence of the standard Newton's method (thus it will incur additional iterations), but we gain the benefit of not having to update the Jacobian matrix after the first iteration. Though the smaller strides do increase the number of iterations, they also help reduce overshooting the root thus lessening wasted steps. In practice, it is found that this trade-off worth making.

This method is also used in many production grade transient stability programs under the name Very Dishonest Newton Raphson (VDHN). The Jacobian matrix is a result of the linearization of differential equations to model the dynamics of the system (i.e. generators, loads, controllers). The Jacobian matrix is normally kept constant and only updated if there is a significant change in the system or if the number of iterations exceed a pre-determined value. [12]

Some power flow algorithms, such as the fast decoupled power flow method or a dc load flow [52], already have constant Jacobian matrices. The natural constant Jacobian is a result of a series

Table 3.12: Iteration Times (ms)

Iteration Type	118 Bus	300 Bus
First Iteration	0.415	1.268
Jacobian Update	0.228	0.781
Constant Jacobian	0.0463	0.125

of approximations based on the assumption of a properly operated power system. However if the assumption of normal conditions is violated (such as a system being heavily loaded) the algorithm may not converge at all.

3.3.2 Results

Results in this section also experiments with an iteration that updates the Jacobian at a certain point in time. Altogether, three different types of iterations can be identified: first iteration, Jacobian update and constant Jacobian. The first iteration performs ordering, symbolic/numerical factorization and backward/forward substitution. The Jacobian update iteration requires numerical factorization and backward/forward substitution. Finally the constant Jacobian iteration only needs to perform backward/forward substitution since the LU factors remain constant. Timing for each type of iteration is given in table 3.12. Notice that the constant Jacobian iteration is roughly an order of magnitude less than the first iteration.

The reduction in execution time for the constant Jacobian iteration comes from the fact that the Jacobian does not need to be updated and that backward/forward substitution is all that is needed. However, it is the reduction in factorization that plays the biggest role. For the 300 bus system, it was found that 88% of the reduction in time was due to the savings from factorization, and only 12% came from not having to update the Jacobian.

Simulation results using a constant Jacobian are shown in tables 3.13 and 3.14. The first column in both tables indicates which iteration the Jacobian update occurred. The speedup is measured against the conventional power flow program detailed in section 3.1. There is a big difference in speedup between the 118 and 300 bus systems. This is largely due to the number of iterations required to solve each system when using the conventional algorithm. The 118 bus requires 3 iterations and the 300 bus requires 4 iterations. The additional iteration accounts for ~20% more time to complete and thus we observe a reasonable speed improvement for the 300 bus.

The 300 bus system was also tested at a higher precision tolerance (1e-5) with a one Jacobian update on the fourth iteration. A speedup of 43% was recorded further validating that the constant Jacobian method works best for systems that naturally require many iterations to converge.

3.3.3 Summary

Using a constant Jacobian method can provide speedups of up ~43%, however simulations that normally converge within a few iterations (two or three) can end up having the reverse effect and actually slow down the overall simulation.

Table 3.13: Execution Times for 118 Bus System (Tolerance of 1e-3)

Update @ Iteration:	# of iterations to converge	Execution Time (ms)	Speedup
no update	6	0.647	5%
2	3	0.689	-1%
3	3	0.689	-1%
4	4	0.736	-8%

Table 3.14: Execution Times for 300 Bus System (Tolerance of 1e-3)

Update @ Iteration:	# of iterations to converge	Execution Time (ms)	Speedup
no update	11	2.518	30%
2	7	2.674	26%
3	5	2.424	33%
4	5	2.424	33%
5	5	2.424	33%
6	6	2.549	29%
7	7	2.674	26%

When implementing the constant Jacobian method on a single CPU, MATE can provide some additional benefits that Sparsity Techniques alone cannot. When dealing with automatic adjustment of transformers controlling either a nodal voltage or a power flow within some certain predefined limit, control equations will need to be substituted or added in if the limits are violated [47]. Thus the Jacobian matrix will need to be updated and will incur numerical factorization and potentially symbolic factorization. In the case of Sparsity Techniques, if there are a small amount of tap changing transformers it is possible to significantly mitigate the factorization time by placing those transformer equations at the bottom of the matrix [54]. However when thethe VAR limits from the PV nodes are include, the number equations at the bottom grow too large for the technique to work properly. Thus MATE is able provide a performance increase over Sparsity Techniques since it only needs to update the subsystems with the necessary control equations. It should be noted however, that a multi-CPU implementation may not receive any additional benefits since the subsystems that do not require updating would have to wait for the subsystems that are updating.

Lastly, inclusion of control equations will favour the constant Jacobian method since the control equations raise the number of iterations required to reach convergence. Thus it is expected that this method would provide a greater improvement in performance when used with a single CPU MATE implementation.

3.4 Chapter Summary

In this chapter, various types of power flow programs were implemented and benchmarked where the goal was to find the best program to serve as a host to MATE. A conventional power flow program was implemented as the base case for speedups. Two versions of the current equation program were

implemented. The current equation program was quite appealing to the MATE algorithm due to its Jacobian matrix giving rise to symmetrical links. However the speedups reported in various published papers were not achieved when implemented. The analysis and benchmarking done in this chapter gives strong evidence against the reported speedup. The main argument being that the additional PV bus equations increases the size and non-zeros of the Jacobian matrix. The time saved from having a near constant Jacobian is not justified by the increased factorization time. Thus the current equation power flow program is not deeded a viable option for MATE. However the constant Jacobian implementation was able to achieve reasonable speed up, as expected.

Chapter 4

Proposed Multi-level Algorithms

As was found in the large-scale implementation of MATE ([56]), the single level approach would scale well only up to around 14 processors. Going above this saturation point causes the link matrix to become too large. A multi-level approach mitigates this problem by allowing for the link matrix to be split up and computed in parallel. Thus a massively parallel multi-level MATE algorithm will be proposed in this chapter. This will be a direct extension of the existing multi-level MATE algorithm found in [5] where the same base formulas are used but furthered for an arbitrary number of levels.

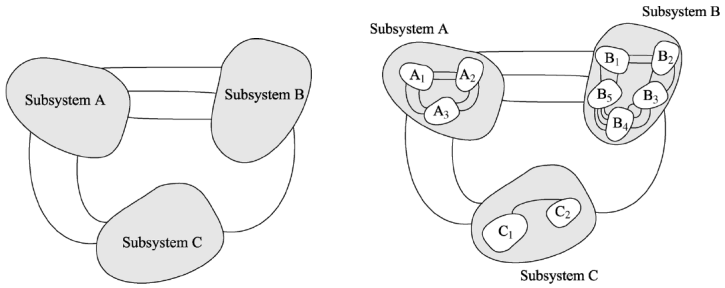
This chapter begins with a short review of literature on multi-level techniques to solving power system simulations. A basic description of the proposed algorithm is provided followed by an example of a level three decomposition. From the level three example, it becomes easier to visualize a general algorithm. A model for a sparse matrix is then described which is then used for an analysis of the distribution of floating point operations in both a branch tearing and node tearing version of the multi-level MATE algorithm.

4.1 Literature Review

The idea of partitioning a subsystem into successive levels can be traced back as far as 1973 where Happ describes a multi-level algorithm for Diakoptics [28]. Since then the concept of using multi-level techniques for MATE and BBDF matrices have naturally been extended (first in [38] called "Extended MATE" and later in [5] as Multi-level MATE). The main benefit of such an algorithm is its ability to distribute the link matrix and thus enable further parallelism.

Multi-level MATE

As described in [5], multi-level MATE was originally designed for inclusion of controller equations and nonlinearities in EMTP. One of the motivations for introducing a second level was the desire to partition a system into natural subsystems that could have their own time steps (i.e. latency techniques). It was recognized that increasing the number of link branches between subsystems slowed down the execution speed of the program and hence the need for the second level.



(a) One Level Decomposition (b) Two Level Decomposition

Figure 4.1: System Decomposition

(extracted from: M.L. Armstrong. Multilevel MATE Algorithm for the Simulation of Power System Transients with the OVNI Simulator. PhD thesis, The University of Bristish Columbia, 2006.)

$$\begin{bmatrix} \begin{bmatrix} A & p_A \\ p_A^t & -z_A \end{bmatrix} & 0 & 0 & \begin{bmatrix} p \\ 0 \end{bmatrix} \\ 0 & \begin{bmatrix} B & q_B \\ q_B^t & -z_B \end{bmatrix} & 0 & \begin{bmatrix} q \\ 0 \end{bmatrix} \\ 0 & 0 & \begin{bmatrix} C & r_C \\ r_C^t & -z_C \end{bmatrix} & \begin{bmatrix} r \\ 0 \end{bmatrix} \\ \begin{bmatrix} p^t & 0 \end{bmatrix} & \begin{bmatrix} q^t & 0 \end{bmatrix} & \begin{bmatrix} r^t & 0 \end{bmatrix} & -z \end{bmatrix} \begin{bmatrix} \begin{bmatrix} v_A \\ i_{A\_sublink} \end{bmatrix} \\ \begin{bmatrix} v_B \\ i_{B\_sublink} \end{bmatrix} \\ \begin{bmatrix} v_C \\ i_{C\_sublink} \end{bmatrix} \\ i_\alpha \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} h_A \\ -V_{A\_sublink} \end{bmatrix} \\ \begin{bmatrix} h_B \\ -V_{B\_sublink} \end{bmatrix} \\ \begin{bmatrix} h_C \\ -V_{C\_sublink} \end{bmatrix} \\ -V_\alpha \end{bmatrix}$$

(a) Starting Matrix Form

$$\begin{bmatrix} 1 & 0 & 0 & a_{MTE} \\ 0 & 1 & 0 & b_{MTE} \\ 0 & 0 & 1 & c_{MTE} \\ 0 & 0 & 0 & z_{\alpha\_MTE} \end{bmatrix} \times \begin{bmatrix} v_A \\ v_B \\ v_C \\ i_\alpha \end{bmatrix} = \begin{bmatrix} e_{A\_MTE} \\ e_{B\_MTE} \\ e_{C\_MTE} \\ e_{\alpha\_MTE} \end{bmatrix}$$

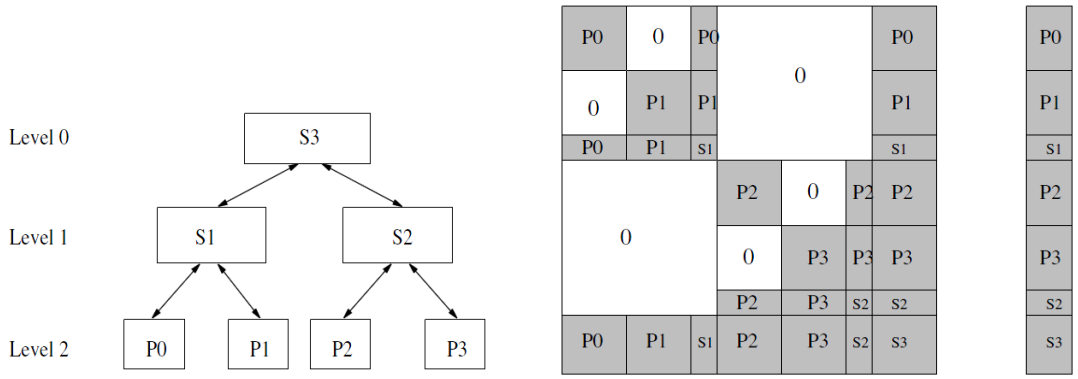
(b) Final Matrix Form

Figure 4.2: Multi-Level MATE's Matrices

(extracted from: M.L. Armstrong. Multilevel MATE Algorithm for the Simulation of Power System Transients with the OVNI Simulator. PhD thesis, The University of Bristish Columbia, 2006.)

The following example was given in [4] where a level two decomposition was performed. Figure 4.1 illustrates the chosen partitioning performed in both first and second level. The corresponding system matrix is shown in figure 4.2 (a) where each diagonal block represents another MATE system. Following a series of elegant manipulation, the sublink branches are able to be "hidden" from the first level and thus results the matrix equation shown in figure 4.2 (b). After collapsing the level two data into level one, this final matrix equation resembles that of the original MATE formulation as shown in equation 4.1.

$$\begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & Z_\alpha \end{bmatrix} \begin{bmatrix} v_A \\ v_B \\ i_\alpha \end{bmatrix} = \begin{bmatrix} e_A \\ e_B \\ e_\alpha \end{bmatrix} \tag{4.1}$$



(a) Separator Tree (b) System Matrix Structure

Figure 4.3: Multi-Level Technique

(extracted from: F. Tu and A.J. Flueck. A message-passing distributed-memory Newton-GMRES parallel power flow algorithm. Power Engineering Society Winter Meeting, 1:211–216. © 2002 IEEE)

Other Multi-level Programs

There have also been other multi-level approaches that have used in the area of power flow. For instance in [57], implemented an algorithm for a general number of levels to be run on a CPU cluster. Figure 4.3 depicts a level two decomposition. The partitioning of the matrix was performed by METIS. Both a direct and iterative matrix solvers where used and it was the iterative solver that was found to provide the superior scaling. The iterative solver showed a speed up of 12.6 times over 16 processors (i.e 79% efficiency per processor). However there was no mention of how the subsystem matrices where factorized in their direct solver other than the authors stating they implemented it. There was also no mention of any optimized linear algebra libraries being used.

Another multi-level implementation described in [62, 64, 63] takes a different approach. It first begins by using epsilon decomposition to partition the system matrix onto multiple processors. The partitioning used creates block borders that are initially much larger than the subsystem matrices themselves. The elements of the sparse matrix are represented by linked lists. Then factorization occurs through a complex algorithm (described in [62]) which iteratively reduces the size of the block border (similar to level elimination). This particular algorithm was reported to have achieved a speedup of 11.2 times over 14 processors (i.e. 80% scaling efficiency per processor). Unfortunately the algorithm was never benchmarked against any well known sparse solvers such SuperLU, MUMPS or UMFPACK.

4.2 General Approach

The main idea is to always divide a subsystem into roughly two equal pieces at each level. By never dividing by more than two, the minimum number links are decoupled at every level. This provides the greatest spread of link matrix computation. Because the division is always by two, the following formula could be derived that describes the relationship between the number of levels L and the desired size of the subsystems.

Table 4.1: Notation of Multi-Level Quantities

Quantity	Description
A, B	Original subsystem matrix
h_A, h_B	Original RHS of subsystem
p, q	Injection/pointer matrices
a, b	Partial Thévenin equivalent matrix
e_A, e_B	Thévenin equivalent voltages
z_{km}	Impedance matrix
z_α	Link matrix
e_α	Link voltage
i_α	Link current

$$\frac{Size.of.System}{2^L} = Size.of.Sub.System \tag{4.2}$$

For example, suppose it is desirable to decompose a system of 2000 nodes into subsystems of size ~8x8. using equation 4.2, it would determine that 8 levels are necessary to achieve the desired subsystem size. A level three example shown in figure 4.5 depicts the composition of the system matrix.

4.3 Notation

Table 4.1 lists the quantities used in the multi-level algorithm. Although the physical meaning of many of the quantities does not apply in case of power flow, much of the original MATE notation is retained for convenience of explanation. Also the word "subsystem" will typically be used to refer to bottom-most level subsystems, otherwise the term "subnetwork" will be used for any arbitrary level.

Numerical subscripts attached to any of the quantities listed in table 4.1 provides further information about which level it belongs to and the current state that quantity is in. The first numerical subscript indicates which level it belongs to (i.e. A_4 is a level 4 subsystem matrix). If a second numerical subscript is present, the second value represents the level it was just modified by (i.e. b_{52} is a level 5 quantity and has just been modified by level 2).

One exception exists to both subscripts at the very start of the program where the value of a subscript can exceed the total number of levels by one. This is merely to indicate initial factorization of each subsystem. An example is given in section 4.4.1.

4.4 Level 3 Multi-level MATE Example

To provide a more concrete idea of the multi-level MATE algorithm, a level three decomposition is provided in this section. This approach will then be generalized for more than three levels in the following section. The algorithm will first start with a partitioned system, then iteratively eliminate each level following the same formulas as described in the existing multi-level MATE algorithm [5]. In this example a system of an arbitrary size is assumed.

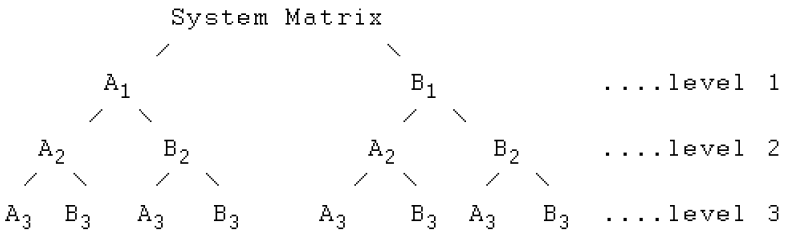


Figure 4.4: Level 3 Hierarchy

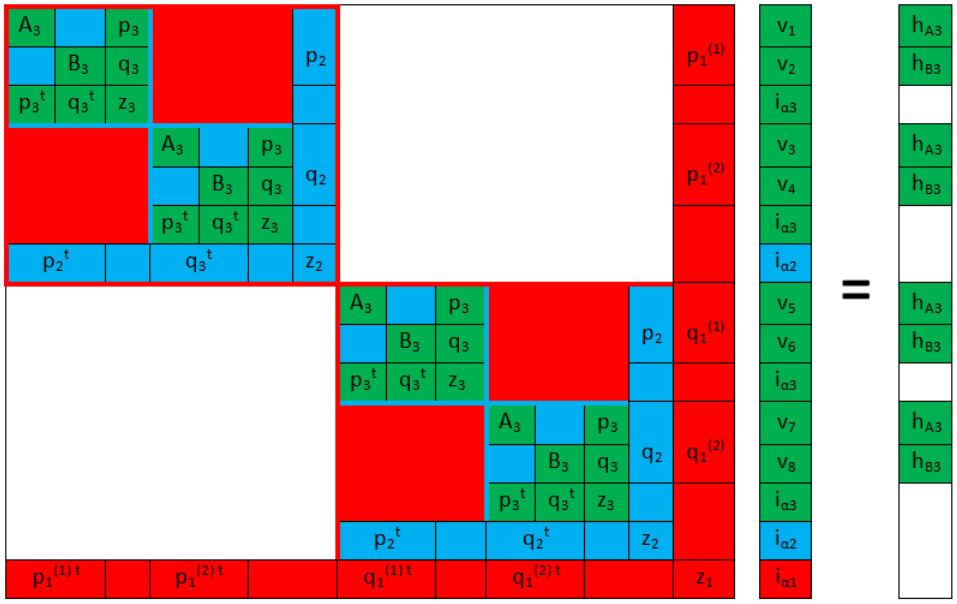


Figure 4.5: Initial Multi-Level Structure

The algorithm begins by partitioning the system into pieces of two as was described in section 4.2. Since three levels are being used, equation 4.2 states that the system matrix will be partitioned into 8 subsystems ($2^3 = 8$). Figure 4.4 depicts the colour coded system matrix where green indicates level three quantities, blue represents level two and red represents level one.

The solution procedure is to perform level elimination starting with the bottom-most level, in this case level three. The goal is to eliminate all level three block borders by transferring their information into the block borders of the remaining levels, thereby eliminating level three. The same process is then done with level two. Then once the system is in the level one state, it proceeds by solving the system as a regular single level MATE implementation as shown in equation 4.1. Conceptually, it can be pictured as ascending the hierarchical pyramid shown in figure 4.4 until level one is reached.

4.4.1 Initial Subsystem Factorization

Prior to the start of level elimination, all the subsystems are first factorized. This step can be seen as inverting each of the subsystem matrices then multiplying through their respective block rows. As a result of having multiple block borders, additional partial Thévenin equivalent matrices fill in the

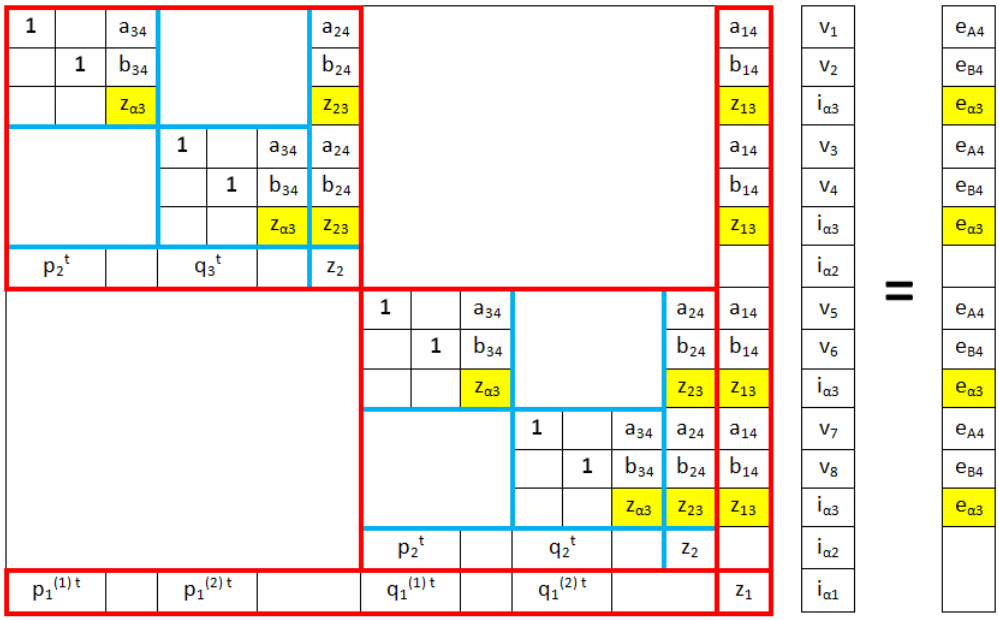


Figure 4.6: After Level 3 Triangularization

injection matrices. The corresponding quantities are shown below.

$$\begin{array}{llll} a_{34} = A_3^{-1} p_3 & a_{24} = A_3^{-1} p_2^{(1)} & a_{14} = A_3^{-1} p_1^{(1)} & e_{A4} = A_3^{-1} h_{A3} \\ b_{34} = B_3^{-1} q_3 & b_{24} = B_3^{-1} p_2^{(2)} & b_{14} = B_3^{-1} p_1^{(2)} & e_{B4} = B_3^{-1} h_{B3} \end{array} \tag{4.3}$$

4.4.2 Elimination of Level 3

The start of the elimination of level three will be defined to begin by performing block Gaussian elimination on the level three block border row. This corresponds to the original MATE operations where a link matrix and link voltage is calculated, except here it is done for level three quantities.

$$z_{\alpha 3} = p_3^t \cdot a_{34} + q_3^t \cdot b_{34} + z_3 \tag{4.4}$$

$$e_{\alpha 3} = p_3^t \cdot e_{A4} + q_3^t \cdot e_{B4} \tag{4.5}$$

As a by-product of the level three link matrix formation, additional impedance matrices are also formed from the block elimination and are given below.

$$\begin{array}{l} z_{23} = p_3^t \cdot a_{24} + q_3^t \cdot b_{24} \\ z_{13} = p_3^t \cdot a_{14} + q_3^t \cdot b_{14} \end{array} \tag{4.6}$$

After performing all these operations, the system matrix will look like that of figure 4.6 where all the updated quantities are highlighted in yellow.

The actual level elimination is done by collapsing the level three quantities into the quantities of the remaining levels. This is accomplished by clever algebraic manipulations of the subsystem formulae as

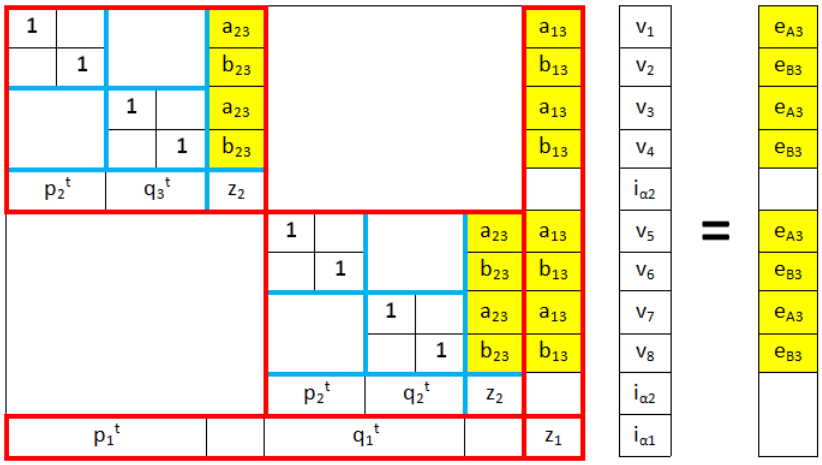


Figure 4.7: After Level 3 Elimination

was done in [5]. The end effect of level three on the remaining levels can be described by the formulas below. Notice the second numerical subscripts of the newly calculated partial Thévenin equivalents is a '3', indicating they were modified by level three.

$$\begin{aligned} a_{23} &= a_{24} - a_{34}\left[z_{\alpha 3}\right]^{-1} z_{23} \qquad a_{13} = a_{14} - a_{34}\left[z_{\alpha 3}\right]^{-1} z_{13} \qquad e_{A3} = e_{A4} - a_{34}\left[z_{\alpha 3}\right]^{-1} e_{\alpha 3} \\ b_{23} &= b_{24} - b_{34}\left[z_{\alpha 3}\right]^{-1} z_{23} \qquad b_{13} = b_{14} - b_{34}\left[z_{\alpha 3}\right]^{-1} z_{13} \qquad e_{B3} = e_{B4} - b_{34}\left[z_{\alpha 3}\right]^{-1} e_{\alpha 3} \end{aligned} \tag{4.7}$$

Later it will be seen that this step makes up the bulk of the computation in the overall algorithm. The corresponding system matrix after level three is eliminated is shown in figure 4.7 with the updated quantities highlighted in yellow.

4.4.3 Elimination of Level 2

As with level three, block Gaussian elimination is performed again except now it is for the level two block border rows to obtain the link quantities. In addition, level three granularity is also present in the level two link equations as shown below. A2 and B2 are specified to clarify which instance of a_{23} and b_{23} is used.

$$z_{\alpha 2} = \overbrace{p_2^{(1)t} \cdot a_{23} + p_2^{(2)t} \cdot b_{23}}^{A2} + \overbrace{q_2^{(1)t} \cdot a_{23} + q_2^{(2)t} \cdot b_{23}}^{B2} + z_2 \tag{4.8}$$

$$e_{\alpha 2} = \underbrace{p_2^{(1)t} \cdot e_{A3} + p_2^{(2)t} \cdot e_{B3}}_{A2} + \underbrace{q_2^{(1)t} \cdot e_{A3} + q_2^{(2)t} \cdot e_{B3}}_{B2} \tag{4.9}$$

Just as before, additional impedance matrix result from the block elimination. Upon performing these operations, the system matrix will change as shown in figure 4.8.

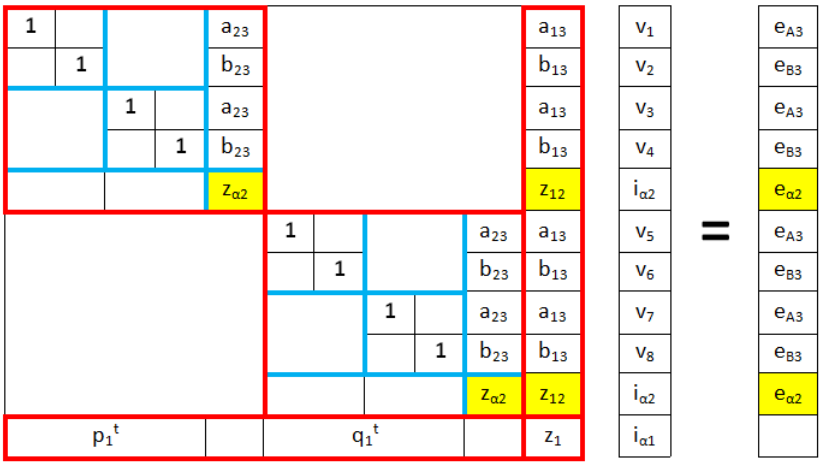


Figure 4.8: Level 2 Triangularized

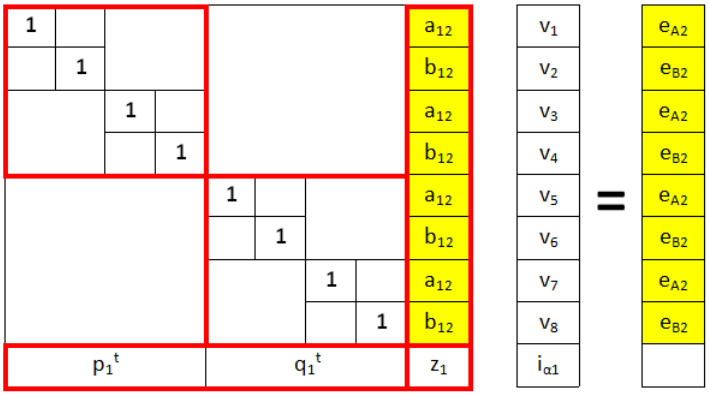


Figure 4.9: Level 1 Matrix After Factorization

$$z_{12} = \overbrace{p_2^{(1)t} \cdot a_{13} + p_2^{(2)t} \cdot b_{13}}^{A2} + \overbrace{q_2^{(1)t} \cdot a_{13} + q_2^{(2)t} \cdot b_{13}}^{B2} \tag{4.10}$$

Level two is then eliminated by transferring the level two block border into the block border of the remaining level one. Similar to level three, this is done with the equations shown below. These equations will then provide the final level one Thévenin equivalent values. The system matrix reflecting these changes are shown in figure 4.9.

$$\begin{aligned} a_{12} &= a_{13} - a_{23}\left[z_{\alpha 2}\right]^{-1} z_{12} \qquad & e_{A2} &= e_{A3} - a_{23}\left[z_{\alpha 2}\right]^{-1} e_{\alpha 2} \\ b_{12} &= b_{13} - b_{23}\left[z_{\alpha 2}\right]^{-1} z_{12} \qquad & e_{B2} &= e_{B3} - b_{23}\left[z_{\alpha 2}\right]^{-1} e_{\alpha 2} \end{aligned} \tag{4.11}$$

4.4.4 Level 1 Solution

Now that all the other levels have been eliminated, the original single-level MATE algorithm can be applied. Only now the system is decoupled into 8 subsystems but with a significantly smaller level one

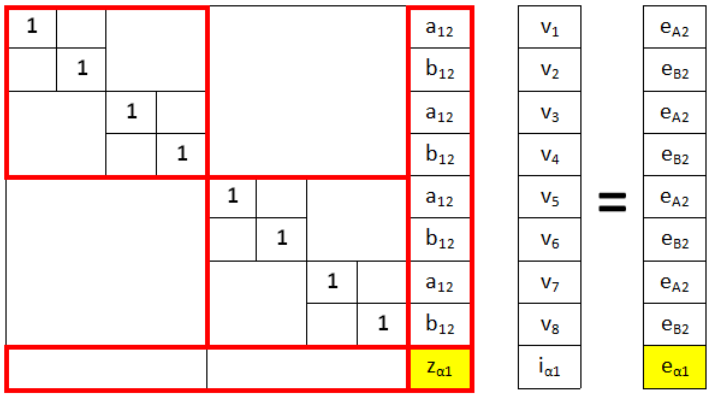


Figure 4.10: Level 1 Triangularized

link matrix. By performing block elimination, the link quantities can be calculated as shown below. Groups of terms within the calculation are labeled to identify which subnetwork they belong to in the level three hierarchy (see figure 4.4). The final system matrix is shown in figure 4.10.

$$z_{\alpha 1} = \overbrace{\underbrace{p_1^{(1)t} \cdot a_{12} + p_1^{(2)t} \cdot b_{12}}_{A2} + \underbrace{p_1^{(3)t} \cdot a_{12} + p_1^{(4)t} \cdot b_{12}}_{B2}}^{A1} + \underbrace{\underbrace{q_1^{(1)t} \cdot a_{12} + q_1^{(2)t} \cdot b_{12}}_{A2} + \underbrace{q_1^{(3)t} \cdot a_{12} + q_1^{(4)t} \cdot b_{12}}_{B2} + z_1}_{B1} \tag{4.12}$$

$$e_{\alpha 1} = \overbrace{\underbrace{p_1^{(1)t} \cdot e_{A2} + p_1^{(2)t} \cdot e_{B2}}_{A2} + \underbrace{p_1^{(3)t} \cdot e_{A2} + p_1^{(4)t} \cdot e_{B2}}_{B2}}^{A1} + \underbrace{\underbrace{q_1^{(1)t} \cdot e_{A2} + q_1^{(2)t} \cdot e_{B2}}_{A2} + \underbrace{q_1^{(3)t} \cdot e_{A2} + q_1^{(4)t} \cdot e_{B2}}_{B2}}_{B1} \tag{4.13}$$

The level one link currents can then be solved and distributed to all the subsystems whereby each subsystem can then solve their respective voltages. From this example there are two main advantages that become apparent when using a multi-level approach to just a single level approach.

1. Matrix operations are done with significantly smaller matrices.

2. The link matrix computation is distributed into smaller pieces which allows for parallel solving.

It is also important to note that every level that is introduced, is a level that needs to also be eliminated. Thus there is a trade-off, the additional levels introduced increases level elimination computation however at the same time they allow for further partitioning which can reduce the cost of matrix operations.

4.5 General Algorithm

The level three example can then be generalized into an algorithm for an arbitrary number of levels. The algorithm can be explained as follows:

I. Initial subsystem factorization

II. Level elimination (reduce levels starting from $L = L^{max}$ until $L = 2$)

   (a) Calculate link level matrices (Thévenin equivalent impedances)

   (b) Eliminate level L

III. Solve global links (level 1)

   (a) Form level 1 Thévenin equivalents

   (b) Solve level 1 link currents

IV. Solve system

   (a) Scatter level 1 link currents to all subsystems

   (b) Solve each subsystem

The bulk of the computation is spent performing level elimination. As was apparent from the level three example, the amount of computation and parallelism diminishes as more levels are eliminated. It should also be noted that the parallelism for the link matrix may appear to diminish much quicker than the subsystem operations. For instance there is only one level one link matrix. However a closer inspection of the level three example would show from equation 4.12 that the computation of the level one link matrix is made up of many terms. In general the level one link matrix will contain 2^L matrix add operations, where L is the number of levels used to decompose a system. Thus a system with ten level would require 1024 matrix add operations.

4.6 Flop Analysis

The goal of this section is to gain insight on the mechanics of the multi-level MATE algorithm. For instance, understanding where the flops are being distributed and how they vary with different choices in subsystem size or number of levels. Information such as knowing the distribution of subsystem to link matrix computation can be vital when implementing an optimized version of multi-level MATE.

However to begin a flops analysis a host system matrix is needed. Since it is a sparse matrix, the sparsity pattern needs to be addressed. For the purpose of this section's analysis, a simple model is presented which uses the following two assumptions:

I. All subsystems are *exactly* partitioned to the desired size

II. All link matrices of each level are *exactly* the same size as the subsystem matrices

These assumptions provide implications to the sparsity of the host matrix while helping to simplify the flop calculations. For instance, if the desired subsystem size is 5x5, then all subsystems are of size 5x5 including the link matrices. The fact that link matrices are of size 5x5 indicates that exactly five branches are cut every time a pair of subnetworks are decoupled. Hence these two assumptions provide implications to the sparsity of the host matrix.

From the description of the multi-level MATE algorithm and the model for the host matrix, the exact number of the total number of inter-subnetwork branches can be determined. To begin, consider the level hierarchy in figure 4.4 where level one contains 2^0 subnetwork split, level two contain 2^1 subnetworks splits and level three contains 2^2 subnetwork splits. Thus the total number of subnetwork splits is $2^0+2^1+2^2$, or in general the geometric series summation formula would be $\left(2^L-1\right)$. Therefore the total number of inter-subnetwork branches is equal to the product of $\left(2^L-1\right)$ and the number of branches between a subnetwork pair. Hence using assumption two from the host matrix model the equations below can be obtained where S represents the order of the host matrix.

$$\#\, of\ inter\ subnetwork\ branches = \left(2^L-1\right) \times (subsystem\ size) \tag{4.14}$$

$$\#\, of\ inter\ subnetwork\ branches = \left(2^L-1\right) \times \left(\frac{S}{2^L}\right) = S - \frac{S}{2^L} \approx S \tag{4.15}$$

Hence if the subsystem matrix size is much smaller than the system matrix, the number of inter-subnetwork branches is always equal to order of the host matrix. However this still doesn't account for the branches within the subsystems. For a sense of comparison, the ~15,000 node WECC system contains ~1.2 branches per node [55]. So suppose subsystems of size 3x3 are chosen, where the average number of branches within the 3x3 subsystems is two. Then the total number of internal branches will be the number of subsystems (i.e. 2^L or $\frac{S}{3}$ using equation 4.2) times the number of branches inside a subsystem (assumed to be two). Therefore the final number of branches in the system can be calculated as follows:

$$\#\, of\ total\ branches = [\#\, of\ inter\ subnetwork\ branches] + [\#\, of\ internal\ branches] \tag{4.16}$$

$$\#\, of\ total\ branches = [S] + \left[\frac{2}{3} \times S\right] = \frac{5}{3}S \tag{4.17}$$

Thus the sparse matrix model has a branch to node ratio of ~1.7 which is not far off from the WECC system's ~1.2 branches per node.

The next two sub-sections will apply the multi-level MATE algorithm to the just described sparse matrix model. In the coming analysis, the matrix size is chosen to be 15,000 x 15,000 to resemble the WECC system's sparsity. The term "flops" in this section will merely refer to plural form of flop instead of floating point operations per second. A flop analysis will be performed using two variations of the

MATE algorithm, a branch tearing version and node tearing version.

4.6.1 Multi-Level MATE Using Branch Tearing

This section will go through the number of floating point operations that each portion of the algorithm requires, then present the totals in graphical and tabular form. The following analysis uses the variables *n* and *L* for denoting the order of the a subsystem and **total** number of levels chosen, respectively, for a system of size 15,000 by 15,000. Derivations are done with geometric series summation formulae using the same approach as was done to derive the number of branches in a system. In the branch tearing implementation, the injection matrices p and q are simply pointers, and thus only reorder values. Hence these operations do not contribute any floating point operations however in the next sub-section on node tearing this won't be the case. This sub-section will cover the number of flops for both a single solution and a repeat solution.

4.6.1.1 Single Solution

The following covers all the operations necessary to perform full factorization to obtain the unknown voltage vectors.

I. **Initial Factorization** consists of two types of operations:

   (a) Factorize all subsystems ($[A_L]^{-1}$ and $[B_L]^{-1}$)

   (b) Solving level L Thévenin voltage equivalents ($e_{A(L+1)} = [A_L]^{-1} h_{AL}$ and $e_{B(L+1)} = [B_L]^{-1} h_{BL}$)

   The performance model assumes the a dense non-symmetrical LU decomposition algorithm is used to calculate the matrix inverses. The total flops for these steps are:

   (a) $2^L \cdot (\frac{8}{3}n^3 - \frac{3}{2}n^2 - \frac{2}{3}n)$ flops

   (b) $2^L \cdot \left(2n^2 - n\right)$ flops

II. **Level Elimination** consists of eight types of operations:

   (a) Link matrix calculations ($z_{\alpha N} = p_N^t \cdot a_{N(N+1)} + q_N^t \cdot b_{N(N+1)} + z_N$)

   (b) Link matrix inversion ($[z_{\alpha N}]^{-1}$)

   (c) Link voltage calculations ($e_{\alpha N} = p_N^t \cdot e_{A(N+1)} + q_N^t \cdot e_{B(N+1)}$)

   (d) Impedance matrix calculations ($z_{iN} = p_N^t \cdot a_{i(N+1)} + q_N^t \cdot b_{i(N+1)}$)

   (e) Impedance matrix multiplication ($[z_{\alpha N}]^{-1} \cdot z_{iN}$)

   (f) Thévenin equivalent voltage multiplication ($[z_{\alpha N}]^{-1} \cdot e_{\alpha N}$)

   (g) Partial Thévenin equivalent reduction ($a_{iN} = a_{i(N+1)} - a_{N(N+1)} [z_{\alpha N}]^{-1} z_{iN}$)

   (h) Thévenin equivalent voltage reduction ($e_{AN} = e_{A(N+1)} - a_{N(N+1)} [z_{\alpha N}]^{-1} e_{\alpha N}$)

where $N = 2...L$ for which $i = 1...(N-1)$; *N* represents the level to be eliminated and *i* represents a level getting modified. The total flops for these steps are:

(a) $(L-1) \cdot 2^L \cdot n^2$ flops

(b) $\left[2^{L+1} - 2\right] \cdot \left(\frac{8}{3}n^3 - \frac{3}{2}n^2 - \frac{2}{3}n\right)$ flops

(c) $\left[(L-2) \cdot 2^L + 2\right] \cdot n$ flops

(d) $\left[2^{L-1}\left(L^2 - 3L + 4\right) - 2\right] \cdot n^2$ flops

(e) $\left[(L-2) \cdot 2^L + 2\right] \cdot \left(2n^3 - n^2\right)$ flops

(f) $\left[(L-2) \cdot 2^L + 2\right] \cdot \left(2n^2 - n\right)$ flops

(g) $2^{L-1} \cdot \left[L^2 - L\right] \cdot \left(2n^3\right)$ flops

(h) $(L-1) \cdot 2^L \cdot \left(2n^2\right)$ flops

III. **Global Link Solution** consists of three types of operations:

(a) Link matrix calculation ($z_{\alpha 1} = p_1^{(1...2^{L-1})t} \cdot a_{12} + q_1^{(1...2^{L-1})t} \cdot b_{12} + z_1$)

(b) Link voltage calculation ($e_{\alpha 1} = p_1^{(1...2^{L-1})t} \cdot e_{A2} + q_1^{(1...2^{L-1})t} \cdot e_{B2}$)

(c) Link solution ($z_{\alpha 1} \cdot i_{\alpha 1} = e_{\alpha 1}$)

The total flops for these steps are:

(a) $2^L \cdot n^2$ flops

(b) $\left[2^L - 1\right] \cdot n$ flops

(c) $\left(\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{5}{3}n\right)$ flops

IV. **System Solution** consists of the following operation:

(a) All subsystem solutions ($v = a_{12} \cdot i_{\alpha 1} - e_{A2}$)

The total flops for these steps are:

(a) $2^L \cdot \left(2n^2 - n + 1\right)$ flops

The flop distribution is provided in table 4.2 which is performed for subsystem matrices of size ~1x1 to ~30x30. The subsystem matrix is size is chosen as the independent variable, while the rest of the quantities in the table are dependent. The table highlights the most expensive operations which happens to be predominately matrix-to-matrix multiplication and also some matrix addition. At the bottom of the table, a condensed version groups together total link and total subsystem operations where subsystem computation is ~70% and link computation is ~30% of the total overall computation.

Another observation that can be made is the amount of total flops as a function of subsystem size. The amount of flops dramatically increases with subsystem size due to the fact that the majority of the flops

Table 4.2: Branch Tearing Single Solution Flop Analysis

Average Sub-System Matrix Sizes		0.92	1.83	3.66	7.32	14.65	29.30
# of Levels Used		14	13	12	11	10	9
# of Sub-Systems		16384	8192	4096	2048	1024	512
Total Mflop		3.4	10.5	33.7	110.4	361.7	1172.7
Flop Distribution							
1. Initial Factorization	a) A^-1	0.1%	0.7%	1.2%	1.7%	2.2%	2.7%
	b) e_A(L+1)	0.3%	0.3%	0.3%	0.2%	0.1%	0.1%
2. Level Elimination	a) z_aN	4.0%	2.6%	1.6%	0.9%	0.5%	0.3%
	b) z_aN^-1	0.1%	1.3%	2.4%	3.3%	4.3%	5.4%
	c) e_aN	4.0%	1.3%	0.4%	0.1%	0.0%	0.0%
	d) z_iN	24.4%	14.8%	8.2%	4.3%	2.1%	1.0%
	e) zmz	3.1%	6.5%	9.2%	11.4%	13.0%	14.3%
	f) zmv	3.4%	3.5%	2.5%	1.6%	0.9%	0.5%
	g) a_iN	51.4%	63.0%	70.6%	74.5%	75.7%	75.0%
	h) e_AN	8.0%	5.3%	3.2%	1.8%	1.0%	0.6%
3. Global Link Solution	a) z_a1	0.3%	0.2%	0.1%	0.1%	0.1%	0.0%
	b) e_a1	0.3%	0.1%	0.0%	0.0%	0.0%	0.0%
4. System Solution	a) v	0.6%	0.4%	0.3%	0.2%	0.1%	0.1%
Condensed Flop Distribution							
1. Initial Factorization		0.3%	1.0%	1.4%	1.8%	2.3%	2.8%
2. Level Elimination	Links	39.0%	30.0%	24.3%	21.5%	20.9%	21.6%
	Sub-System	59.4%	68.2%	73.8%	76.4%	76.7%	75.5%
3. Global Link Solution		0.6%	0.3%	0.2%	0.1%	0.1%	0.0%
4. System Solution		0.6%	0.4%	0.3%	0.2%	0.1%	0.1%

comes from dense matrix-to-matrix multiplication which is an O{n^3} operation. Hence the multi-level MATE algorithm performs best with smaller subsystem matrices. However also notice that the ~1x1 subsystem matrices do exceptionally well. This is mostly due to the two assumptions that were specified at the start of the section to model the host sparse matrix. For instance, the system matrix is less sparse than normal because the subsystem size is 1x1, since there are no internal branches for a 1x1 subsystem. Hence the total number of branches in the system is equal to the number of nodes in the system. But the main culprit would be the inaccurate decoupling of the subnetworks. Since the subsystem matrices are of order one, the algorithm always assumes that only one branch needs to be decoupled to split a subnetwork. Thus most accurate values begin around 3x3 to 4x4 sized subsystems.

4.6.1.2 Repeat Solutions

Assumes the system matrix has already been factorized but a new RHS is introduced. Hence the matrix quantities in the formulas below are merely being loaded from memory and the computation mainly deals with Thévenin voltage quantities. In power flow, this would be the case when the Jacobian matrix is reused for the next iteration.

I. **Initial Factorization** consists of one operation:

(a) Solving level L Thévenin voltage equivalents ($e_{A(L+1)} = [A_L]^{-1} h_{AL}$)

The total flops for these steps are:

(a) $2^L \cdot \left(2n^2 - n\right)$ flops

II. **Level Elimination** consists of three types of operations:

(a) Link voltage calculations ($e_{\alpha N} = p_N^t \cdot e_{A(N+1)} + q_N^t \cdot e_{B(N+1)}$)

(b) Thévenin equivalent voltage multiplication ($\left[z_{\alpha N}\right]^{-1} \cdot e_{\alpha N}$)

(c) Thévenin equivalent voltage reduction ($e_{AN} = e_{A(N+1)} - a_{N(N+1)} \left[z_{\alpha N}\right]^{-1} e_{\alpha N}$)

where $N = 2...L$ for which $i = 1...(N-1)$; *N* represents the level to be eliminated and *i* represents a level getting modified. The total flops for these steps are:

(a) $\left[(L-2) \cdot 2^L + 2\right] \cdot n$ flops

(b) $\left[(L-2) \cdot 2^L + 2\right] \cdot \left(2n^2 - n\right)$ flops

(c) $(L-1) \cdot 2^L \cdot \left(2n^2\right)$ flops

III. **Global Link Solution** consists of two types of operations:

(a) Link voltage calculation ($e_{\alpha 1} = p_1^{(1...2^{L-1})t} \cdot e_{A2} + q_1^{(1...2^{L-1})t} \cdot e_{B2}$)

(b) Link solution, re-use LU factors ($z_{\alpha 1} \cdot i_{\alpha 1} = e_{\alpha 1}$)

The total flops for these steps are:

(a) $\left[2^L - 1\right] \cdot n$ flops

(b) $\left(2n^2 - n\right)$ flops

IV. **System Solution** consists of the following operation:

(a) All subsystem solutions ($v = a_{12} \cdot i_{\alpha 1} - e_{A2}$)

The total flops for these steps are:

(a) $2^L \cdot \left(2n^2 - n + 1\right)$ flops

The results for a repeat solution are shown in table 4.3. Notice there is ~10 times less computation than a full factorization. The amount of link and subsystem computation is now almost the same.

4.6.2 Multi-Level MATE Using Node Tearing

Similar to branch tearing sub-section, this sub-section will use the variables *n* and *L* for denoting the order of the a subsystem and **total** number of levels chosen, respectively, for a system of size 15,000 by 15,000. The node tearing version assumes that all block borders (p and q injection matrices) are dense and thus now contribute to the total amount of flops. However node tearing does benefit from the fact that it does not require additional equations to represent the link quantities.

This sub-section will cover the number of flops for both a single solution and a repeat solution.

Table 4.3: Branch Tearing Repeat Solution Flop Analysis

Average Sub-System Matrix Sizes		0.92	1.83	3.66	7.32	14.65	29.30
# of Levels Used		14	13	12	11	10	9
# of Sub-Systems		16384	8192	4096	2048	1024	512
Total Mflop		0.74	1.37	2.52	4.60	8.34	14.93
Flop Distribution							
1. Initial Factorization	a) e_A(L+1)	1.7%	2.9%	3.8%	4.4%	5.1%	5.8%
2. Level Elimination	a) e_aN	24.2%	12.1%	6.0%	2.9%	1.4%	0.7%
	b) zmv	20.1%	32.1%	37.7%	40.0%	40.7%	40.5%
	c) e_AN	48.1%	48.2%	48.0%	47.8%	47.4%	47.1%
3. Global Link Solution	a) e_a1	2.0%	1.1%	0.6%	0.3%	0.2%	0.1%
	b) i_a1	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
4. System Solution	a) v	3.9%	3.5%	3.9%	4.5%	5.1%	5.8%
Condensed Flop Distribution							
1. Initial Factorization		1.7%	2.9%	3.8%	4.4%	5.1%	5.8%
2. Level Elimination	Links	44.4%	44.2%	43.7%	43.0%	42.2%	41.2%
	Sub-System	48.1%	48.2%	48.0%	47.8%	47.4%	47.1%
3. Global Link Solution		2.0%	1.1%	0.6%	0.3%	0.2%	0.1%
4. System Solution		3.9%	3.5%	3.9%	4.5%	5.1%	5.8%

4.6.2.1 Single Solution

The following covers all the operations necessary to perform full factorization to obtain the unknown voltage vectors.

I. **Initial Factorization** consists of two types of operations:

   (a) Factorize all subsystems ($A_L a_{L(L+1)} = p_1$ and $B_L b_{L(L+1)} = q_L$)

   (b) Solve level L Thévenin voltage equivalents ($A_L e_{A(L+1)} = h_{AL}$ and $B_L e_{B(L+1)} = h_{BL}$)

   The performance model assumes the a dense LU decomposition algorithm is used to calculate the matrix inverses. The total flops for these steps are:

   (a) $2^L \cdot (\frac{8}{3}n^3 - \frac{3}{2}n^2 - \frac{2}{3}n)$ flops

   (b) $2^L \cdot \left(2n^2 - n\right)$ flops

II. **Level Elimination** consists of eight types of operations:

   (a) Link matrix calculations ($z_{\alpha N} = p_N^t \cdot a_{N(N+1)} + q_N^t \cdot b_{N(N+1)} + z_N$)

   (b) Link matrix inversion ($[z_{\alpha N}]^{-1}$)

   (c) Link voltage calculations ($e_{\alpha N} = p_N^t \cdot e_{A(N+1)} + q_N^t \cdot e_{B(N+1)}$)

   (d) Impedance matrix calculations ($z_{iN} = p_N^t \cdot a_{i(N+1)} + q_N^t \cdot b_{i(N+1)}$)

   (e) Impedance matrix multiplication ($[z_{\alpha N}]^{-1} \cdot z_{iN}$)

   (f) Thévenin equivalent voltage multiplication ($[z_{\alpha N}]^{-1} \cdot e_{\alpha N}$)

   (g) Partial Thévenin equivalent reduction ($a_{iN} = a_{i(N+1)} - a_{N(N+1)} [z_{\alpha N}]^{-1} z_{iN}$)

(h) Thévenin equivalent voltage reduction ($e_{AN} = e_{A(N+1)} - a_{N(N+1)} \left[z_{\alpha N}\right]^{-1} e_{\alpha N}$)

where $N = 2...L$ for which $i = 1...(N-1)$; *N* represents the level to be eliminated and *i* represents a level getting modified. The total flops for these steps are:

(a) $(L-1) \cdot 2^L \cdot \left(2n^3\right)$ flops

(b) $\left[2^{L+1} - 2\right] \cdot \left(\frac{8}{3}n^3 - \frac{3}{2}n^2 - \frac{2}{3}n\right)$ flops

(c) $\left[L-1\right] \cdot 2^L \left(2n^2\right) + \left[2 - 2^L\right] \cdot n$ flops

(d) $\left[2^L \left(L^2 - L\right)\right] \cdot n^3 + \left[2^{L+1}\left(1-L\right) - 2\right] \cdot n^2$ flops

(e) $\left[(L-2) \cdot 2^L + 2\right] \cdot \left(2n^3 - n^2\right)$ flops

(f) $\left[(L-2) \cdot 2^L + 2\right] \cdot \left(2n^2 - n\right)$ flops

(g) $2^{L-1} \cdot \left[L^2 - L\right] \cdot \left(2n^3\right)$ flops

(h) $(L-1) \cdot 2^L \cdot \left(2n^2\right)$ flops

III. **Global Link Solution** consists of three types of operations:

(a) Link matrix calculation ($z_{\alpha 1} = p_1^{(1...2^{L-1})t} \cdot a_{12} + q_1^{(1...2^{L-1})t} \cdot b_{12} + z_1$)

(b) Link voltage calculation ($e_{\alpha 1} = p_1^{(1...2^{L-1})t} \cdot e_{A2} + q_1^{(1...2^{L-1})t} \cdot e_{B2}$)

(c) Link solution ($z_{\alpha 1} \cdot i_{\alpha 1} = e_{\alpha 1}$)

The total flops for these steps are:

(a) $2^L \cdot \left(2n^3\right)$ flops

(b) $2^{L+1} \cdot n^2 - n$ flops

(c) $\left(\frac{1}{3}n^3 + \frac{3}{2}n^2 + \frac{1}{6}n\right)$ flops

IV. **System Solution** consists of the following operation:

(a) All subsystem solutions ($v = a_{12} \cdot i_{\alpha 1} - e_{A2}$)

The total flops for these steps are:

(a) $2^L \cdot \left(2n^2 - n + 1\right)$ flops

The results are shown in table 4.4. Much of the analysis discussed in the branch tearing sub-section holds here as well. Notice that there appears to be a ~20% reduction in total amount of flops from the branch tearing version. Also the two most expensive operations comes almost exclusively from matrix multiplication. However subsystem and link computation are now roughly the same.

Table 4.4: Node Tearing Single Solution Flop Analysis

Average Sub-System Matrix Sizes		0.92	1.83	3.66	7.33	14.66	29.35
# of Levels Used		13	12	11	10	9	8
# of Sub-Systems		8192	4096	2048	1024	512	256
Total Mflop		2.38	8.02	26.70	88.04	286.67	917.62
Flop Distribution							
1. Initial Factorization	a) A^-1	0.1%	0.5%	0.8%	1.1%	1.4%	1.8%
	b) e_A(L+1)	0.3%	0.2%	0.2%	0.1%	0.1%	0.0%
2. Level Elimination	a) z_aN	3.2%	3.4%	3.8%	4.1%	4.5%	4.9%
	b) z_aN^-1	0.1%	1.0%	1.7%	2.2%	2.9%	3.7%
	c) e_aN	6.6%	3.7%	2.0%	1.1%	0.6%	0.3%
	d) z_iN	34.3%	37.6%	39.4%	40.1%	39.9%	39.2%
	e) zmz	2.6%	4.6%	5.9%	6.8%	7.6%	8.3%
	f) zmv	2.9%	2.5%	1.6%	0.9%	0.5%	0.3%
	g) a_iN	41.3%	41.4%	41.5%	41.2%	40.5%	39.5%
	h) e_AN	6.9%	3.8%	2.1%	1.1%	0.6%	0.3%
3. Global Link Solution	a) z_a1	0.5%	0.6%	0.8%	0.9%	1.1%	1.4%
	b) e_a1	0.6%	0.3%	0.2%	0.1%	0.1%	0.0%
4. System Solution	a) v	0.6%	0.3%	0.2%	0.1%	0.1%	0.0%
Condensed Flop Distribution							
1. Initial Factorization		0.3%	0.8%	1.0%	1.2%	1.5%	1.9%
2. Level Elimination	Links	49.8%	52.8%	54.3%	55.3%	56.1%	56.7%
	Sub-System	48.2%	45.1%	43.5%	42.3%	41.2%	39.9%
3. Global Link Solution		1.1%	1.0%	1.0%	1.0%	1.2%	1.5%
4. System Solution		0.6%	0.3%	0.2%	0.1%	0.1%	0.0%

4.6.2.2 Repeat Solutions

Assumes the system matrix has already been factorized but a new RHS is introduced. Formulas given here are identical to the repeat solution for the branch tearing version, however there is an increase computation in step 3 Global Link Solution.

I. **Initial Factorization** consists of one operation:

   (a) Solving level L Thévenin voltage equivalents ($e_{A(L+1)} = [A_L]^{-1} h_{AL}$)

   The total flops for these steps are:

   (a) $2^L \cdot \left(2n^2 - n\right)$ flops

II. **Level Elimination** consists of three types of operations:

   (a) Link voltage calculations ($e_{\alpha N} = p_N^t \cdot e_{A(N+1)} + q_N^t \cdot e_{B(N+1)}$)

   (b) Thévenin equivalent voltage multiplication ($[z_{\alpha N}]^{-1} \cdot e_{\alpha N}$)

   (c) Thévenin equivalent voltage reduction ($e_{AN} = e_{A(N+1)} - a_{N(N+1)} [z_{\alpha N}]^{-1} e_{\alpha N}$)

   where $N = 2...L$ for which $i = 1...(N-1)$; N represents the level to be eliminated and i represents a level getting modified. The total flops for these steps are:

(a) $[L-1] \cdot 2^L \left(2n^2\right) + \left[2 - 2^L\right] \cdot n$ flops

(b) $\left[(L-2) \cdot 2^L + 2\right] \cdot \left(2n^2 - n\right)$ flops

(c) $(L-1) \cdot 2^L \cdot \left(2n^2\right)$ flops

III. **Global Link Solution** consists of three types of operations:

(a) Link voltage calculation ($e_{\alpha 1} = p_1^{(1...2^{L-1})t} \cdot e_{A2} + q_1^{(1...2^{L-1})t} \cdot e_{B2}$)

(b) Link solution, re-use LU factors ($z_{\alpha 1} \cdot i_{\alpha 1} = e_{\alpha 1}$)

The total flops for these steps are:

(a) $2^L \cdot \left(2n^3\right)$ flops

(b) $2^{L+1} \cdot n^2 - n$ flops

(c) $\left(\frac{1}{3}n^3 + \frac{3}{2}n^2 + \frac{1}{6}n\right)$ flops

IV. **System Solution** consists of the following operation:

(a) All subsystem solutions ($v = a_{12} \cdot i_{\alpha 1} - e_{A2}$)

The total flops for these steps are:

(a) $2^L \cdot \left(2n^2 - n + 1\right)$ flops

Results are shown in table 4.5. The data indicates that the repeat solution for node tearing only requires half the computation of the repeat solution for branch tearing. The overall reduction is a result of the subsystem nodes only containing 50% of the total number of system nodes, since the other 50% is used as link quantities for block borders.

4.7 Conclusions

This chapter redesigns MATE as a massively parallel algorithm. An analysis on the distribution of floating point operations was performed on two different versions, a branch tearing version and a node tearing version. The data from the flop analysis indicates that the node tearing version requires less computation to solve a sparse matrix problem. The flop analysis also determined that the majority of the floating point operations came from matrix-to-matrix multiplication. Since matrix multiplication is an O{n^3} operation, it was seen that smaller subsystem matrices where advantageous.

It should be noted that the flop analysis is not a performance model. The calculation of the total floating point values are approximate since they were based on the two assumptions stated at the beginning of section 4.6. The simplistic model allows for a clear view of the internal mechanics of the proposed multi-level MATE algorithm. It is capable of showing where the floating point operations are being allocated, which can serve as a good tool for implementing/optimizing the algorithm.

Table 4.5: Node Tearing Repeat Solution Flop Analysis

Average Sub-System Matrix Sizes		0.92	1.83	3.66	7.33	14.66	29.35
# of Levels Used		13	12	11	10	9	8
# of Sub-Systems		8192	4096	2048	1024	512	256
Total Mflop		0.35	0.65	1.20	2.19	3.95	7.05
Flop Distribution							
1. Initial Factorization	a) e_A(L+1)	1.8%	3.1%	4.0%	4.7%	5.4%	6.2%
2. Level Elimination	a) e_aN	23.6%	11.6%	5.6%	2.7%	1.3%	0.6%
	b) zmv	19.6%	30.8%	35.7%	37.5%	37.7%	37.0%
	c) e_AN	47.1%	46.6%	45.9%	45.3%	44.6%	43.8%
3. Global Link Solution	a) e_a1	3.9%	4.2%	4.6%	5.0%	5.6%	6.3%
	b) i_a1	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
4. System Solution	a) v	4.1%	3.7%	4.1%	4.7%	5.4%	6.2%
Condensed Flop Distribution							
1. Initial Factorization		1.8%	3.1%	4.0%	4.7%	5.4%	6.2%
2. Level Elimination	Links	43.1%	42.4%	41.4%	40.3%	39.0%	37.6%
	Sub-System	47.1%	46.6%	45.9%	45.3%	44.6%	43.8%
3. Global Link Solution		3.9%	4.2%	4.6%	5.0%	5.6%	6.3%
4. System Solution		4.1%	3.7%	4.1%	4.7%	5.4%	6.2%

The proposed algorithm's main strength is its ability to be partitioned into hundreds to thousands of subsystems. However this comes at a cost of performing dense matrix operations with both subsystems and block border quantities. Thus the algorithm would not perform as well on a single CPU than a normal sparse solver. However, as will be shown in the next chapter, the simplicity of the algorithm would perform well on a massively parallel SIMD architecture such as a GPU.

Chapter 5

The Graphics Processing Unit and MATE

In the last decade, GPUs have been outperforming CPUs by an order of magnitude in raw computational power [44]. This has led to a lot of interest from many scientific fields where general purpose computation on GPUs (GPGPU) has flourished for a variety of applications. However not all applications can be easily ported to the GPU. Programs must be able to be inherently parallel (aka embarrassingly parallel) to take advantage of the GPU's architecture.

The goal of this chapter is to take the first step towards checking the viability of the multi-level MATE algorithm on the GPU. From the previous chapter, it was found that the majority of the computation is spent on matrix-to-matrix multiplication. Thus if MATE is to succeed on the GPU, the matrix multiplication routine which can effectively handle MATE's data needs to run efficiently on the GPU. Hence this will be the main objective of this chapter.

To accomplish this intent, the chapter will first begin with a literature review of sparse solvers on the GPU from several different scientific communities, including power systems. An overview on the GPU's architecture and software is given to provide context as well as necessary concepts to be used later in the chapter. Then a short overview on issues surrounding GPU programming, such as important abstractions as well as the matter of hiding latencies. Finally a data representation for MATE is proposed along with the design, implementation and benchmarking of the matrix multiplication routine.

5.1 Literature Review

With the recent advancements in GPU technology, there has been significant performance improvements *reported* in many scientific disciplines. This literature review will focus on direct and iterative sparse matrix solvers from various areas of study. The section is divided into three parts: (1) conjugate gradient solvers, (2) multifrontal solvers and (3) solvers in power system simulations. Papers generally will discuss GPU and CPU hardware and software used to obtain some measure of speedup. Unfortunately, it was not uncommon, especially in the earlier years, to find papers with unfair comparisons. These were generally papers that claimed several orders of magnitude in speedup by using poorly optimized CPU code. It is not uncommon for an author to spend months optimizing GPU code, then rush CPU code for the sole purpose of benchmarking. Thus it is always important to check for signs of proper

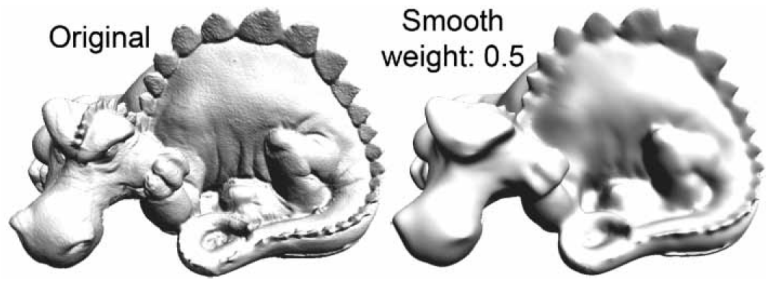


Figure 5.1: Phlegmatic Dragon

(extracted from: L. Buatois, G. Caumon, and B. Lévy. Concurrent number cruncher: a GPU implementation of a general sparse linear solver. International Journal of Parallel, Emergent and Distributed Systems, 24(3):205–223, 2009.)

optimized CPU code. For instance, the usage of OpenMp, SSE, or any standard BLAS/math libraries. Also checking for CPU comparisons with other papers or more simply just using standard software simulation packages from their respective field.

Conjugate Gradient Solvers: In [11], a CG solver called the *Concurrent Number Cruncher* (CNC) was designed and implemented for both the ATI and NVIDIA graphics cards. The application was in the realm of graphics computation, more specifically parametrization and smoothing of a mesh for various objects (see figure 5.1). The sparse matrices ranged in size from $1.5k \times 1.5k$ to $1M \times 1M$. The GPUs used were the NVIDIA QuadroFX 5600 (equivalent of GeForce 8800 Ultra) and an older ATI X1900XTX (competed with the GeForce 7000 series). The CPU used for comparison was a dual core AMD Athlon 64 X2 4800+. The preconditioned CG algorithm makes use of two BLAS routines: the sparse matrix-vector product and the vector-inner product. From benchmarking, it was found that the sparse matrix-vector product operation took roughly 80% of the execution time. The GPU implementations were compared to a multithreaded SSE3 optimized CPU code. To further verify the validity of the optimized CPU code, it was benchmarked against the routines in the MKL and ACML libraries. Their best case results were for their largest matrix ($1M \times 1M$) which showed speedups of 6.0 and 3.2 times faster with NVIDIA and ATI cards, respectively than the CPU implementation. However the smaller $1.5k \times 1.5k$ matrix actually preformed worst than the CPU implementation.

Another CG solver is detailed in [61], which is designed for digital optical tomography. Benchmarking was done with a test matrix of size $138,324 \times 138,324$. The GPU used was NVIDIA's GeForce 8800 GTX which was compared against Intel's dual core Xeon 5140 Woodcrest. The GPU implementation was compared against SSE optimized CPU code using MKL functions as well as OpenMP parallelism between the two cores. Similar to the previous CG solver described in this section, the sparse matrix-vector product took ~85% of the computation time. The GPU was recorded as 2.56 times faster, however for solutions with equal approximated relative error, the execution times were roughly the same.

Multifrontal Solvers: In [33] and [39], multifrontal solvers have been created for the application of modeling a Boeing wing and for mechanical engineering simulations, respectively. Both using finite element packages as a means of comparison, ANSYS for [33] and MCAE (Mechanical Computer Aided Engineering) for [39]. In both cases, the GPU is essentially used as the BLAS for the multifrontal algorithm (i.e. providing dense matrix factorizations for the frontal matrices). However in [39], only

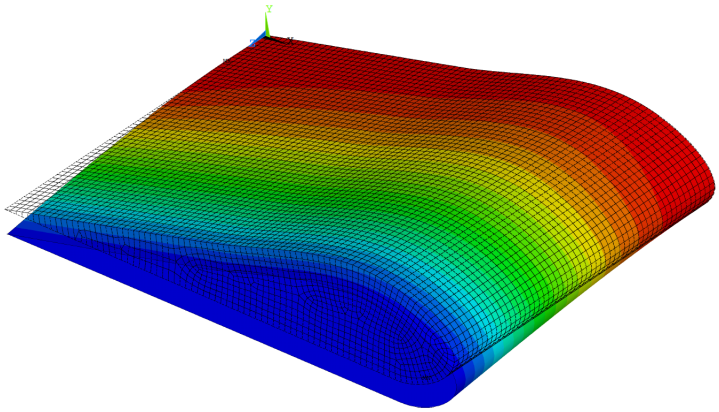


Figure 5.2: BM-7 Wing Model
(extracted from: G.P. Krawezik and G. Poole. Accelerating the ANSYS Direct Sparse Solver with GPUs.)

the largest frontal matrices were factorized on the GPU, the smaller matrices were done on the CPU. The GPU to CPU comparison in [33] was much more thorough in providing the test conditions. It used an NVIDIA Tesla C1060 and compared the performance of ANSYS running on two quad-core Xeon processors. ANSYS allowed for various matrix sizes to represent the wing model and was benchmarked for matrices of order 250 thousand to 2 million (which are much larger than power system grid). They were able to achieve speedups of up to 4 times faster than the two quad-core CPUs. They did note that the Boeing wing was a very blocky model (see figure 5.2), and did not contain any holes or thin shapes which resulted in large frontal matrices, instead of smaller matrices, which favours the GPU architecture. However in [39], these "less favourable" models were used in their simulations. The models they used were an I-beam, a hood and a prosthetic knee which corresponded to systems of order 70 thousand to 236 thousand. They achieved speedups of 1.96 times faster, however the CPU platform was never mentioned. However they did use an older GPU (NVIDA GeForce 8800) than [33].

Power Systems: There has also been GPU work done in the area of power systems, namely [7] in power flow and [30] transient stability simulation. Both papers used NVIDIA's GTX 280 and compared their results to a quad-core CPU. These two papers also used the conjugate gradient method on the GPU side as their sparse matrix solver. Unfortunately, the paper on transient stability analysis falls under the category of "unfair comparisons" as mentioned at the start of this section. There were two issues that could have led to this unfair comparison. The first issue has to do with the test matrix. An IEEE 39 bus system was said to be "duplicated several times to create systems of larger scales" from which the author created a system 1248 bus system. No further information was giving on how the duplication took place or what the 1248 bus system looked like. The second issue, and most likely the most problematic, was the CPU code prepared. The paper did claim that the PSS/E software package was used, but only to check the accuracy of their results. There was no mention of any kind of optimizations such as SSE instructions or any math or BLAS libraries being used. This led to a very suspicious GPU speedup of **340 times faster** than the CPU code. Notice that this result is utterly out of line with the results of the papers on CG solvers mentioned previously in this section, especially when considering the relatively small size of the power system matrix. Furthermore, assuming equal efficiency of processor usage on

both the CPU and GPU used, it is physically impossible to achieve a speedup of any more than roughly 10 times (more details on GPU and CPU architecture in section 5.2.1.2). This speedup would strongly imply unoptimized CPU code.

The second paper ([7]) on power flow does not appear to suffer from this problem as it used MATLAB's sparse solver for the CPU comparison. Assuming the author used MATLAB 6.0 (released in 2000) or later, the built in sparse solver would have been UMFPACK, a well known multifrontal solver. The paper compared the execution time required to factorize Jacobian matrices belonging to 685 node and 1138 node power systems. The best case speedup was only 1.5 times faster. However upon further investigation of the references, it turns out that the author used large tridiagonal matrices from the area of fluid dynamics to obtain their best case speedup. The actual speedup for for 685 node and 1138 node systems were 1.04 and 1.14 times faster, respectively.

Several conclusions can be made from this literature review on GPU sparse solvers. The general trend is that the current direct and iterative sparse matrix solvers on the GPU provide relatively small speedups in comparison with a CPU, especially when compared to the speedup claims from other fields. Both the CG and multifrontal solvers, only perform marginally well for matrices much larger than power systems (i.e. 1 million degrees of freedom or more). Furthermore the power system papers do not seem to show much promise either, although it was to be expected since they both used the CG method. The main bottleneck of the CG solver is the sparse matrix-vector product routine. A thorough research paper written by NVIDIA employees ([8]) on the sparse matrix-vector product routine benchmark several common approaches to the implementation. The overall performances ranged from 1% to 2% of the GPU theoretical flops. The multifrontal solver on the other hand, requires extremely large frontal matrices in order to use the GPU efficiently. The frontal matrix size is dependent upon the system size and grid connection pattern. Both of which are not favoured when it comes to a power system graph.

5.2 GPU Comparisons

This section will provide a brief overview of both the hardware and software of the two largest GPU producers: NVIDIA and ATI/AMD. The hardware comparison will provide some background on the processor architecture as well as the theoretical flops each machine can perform. The software comparison will provide a quick overview of the languages and high performance libraries available from either company. The pros and cons of each GPU are considered for attempting choose which GPU would be best for MATE.

5.2.1 Architectures

This section will first compare the architectures of the two main GPU producers followed by a comparison between the architectures of a GPU and a CPU.

5.2.1.1 ATI vs. NVIDIA

The processor architecture of both NVIDIA and ATI graphics cards will be examined in this section. The comparison is made between the top workstation grade GPUs from both companies (as of mid 2009), which are the Tesla C1060 from NVIDIA and the FireStream 9270 from ATI. The consumer equivalent ("gaming cards") GPUs are the GeForce GTX 280 (NVIDIA) and the Radeon HD 4870 (ATI). From the architecture design, a GPU's theoretical flops can be determined, which will provide greater insight as to the programming of GPUs.

Modern GPUs contain hundreds of scalar processors. However the architecture of the scalar or streaming processors (SPs) are much simpler than a CPU processor, thus the SPs are best thought of as simple ALUs. Furthermore, the streaming processors cannot act independently of each other. The SPs are partitioned into groups of 8 (NVIDIA's streaming multiprocessor) or 64 (ATI's SIMD Engine) which are called vector processors (see figure 5.3). Once an instruction is given to a vector processor, all the SPs will execute the same instruction in parallel. For instance, if an ADD instruction were issued to a given vector processor, then all SPs within that vector processor would have to perform an ADD operation on their respective data, hence SIMD (single instruction, multiple data) execution.

The GPU works in vectors. Thus independent scalar operations are not possible since instructions are always issued in vector form. As the architecture suggests for the ATI card, the vector length is 64 elements (or threads) since there are 64 streaming cores in a SIMD engine. For instance, consider the addition of two vectors with a length 64 elements on a SIMD engine. Each of the 64 cores would add one element from both vectors which is performed in parallel and thus take can take 1 clock cycle (since each scalar core has a throughput of 1 ADD per clock cycle). This group of 64 threads in which instructions are issued is referred to as a wavefront.

Instead of wavefronts, NVIDIA has warps which consist of 32 threads which are given to the SM. Since a multiprocessor consists of only 8 scalar processors, it takes an SM a minimum of 4 clock cycles to complete an instruction of 1 warp. NVIDIA's next generation architecture (Fermi) will contain a more appropriate size of 32 scalar processors.

Streaming processors are capable of performing 32-bit (single precision) operations. In figure 5.3, the SIMD engine contains T-Stream Cores, which perform transcendental operations but are also capable of performing multiply-add (MAD) instructions much like the SPs. The streaming multiprocessors (SMs) contain their own special functions units (SFUs) which perform transcendental operations but are not capable of performing single precision MAD instructions.

Double precision operations are handled differently on either architecture. The SM performs double precision calculation through the SFU, thus it only has one 64 bit processor (as listed in table 5.1). The SIMD Engine will combine four stream cores to produce 64 bit calculations, the T-Stream Core does not participate. Thus the SIMD engine has equivalently 16 double precision cores.

Within the vector processors, there is a small cache (16KB) of programmable memory that is directly accessible only by the SPs of a vector processor. Note that there is a lot more cache memory per scalar processor in NVIDIA's vector processor since it is only shared with 8 processors as oppose to 64 (see table 5.1).

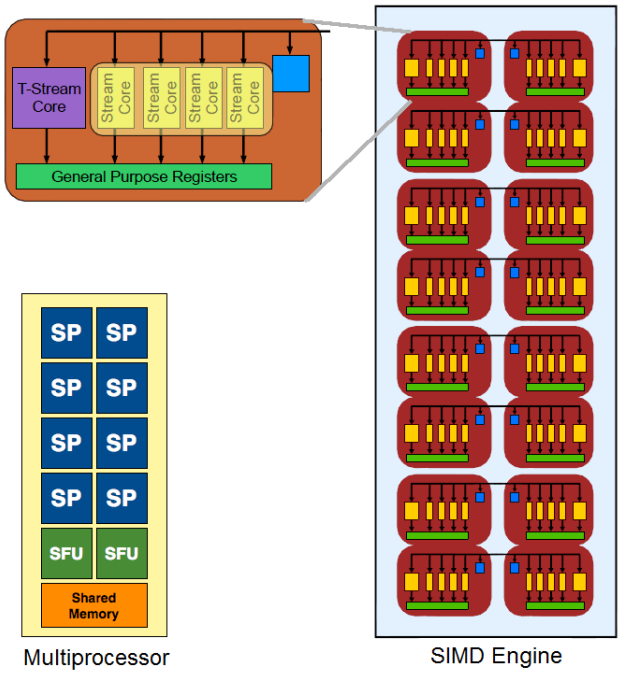


Figure 5.3: NVIDIA Streaming Multiprocessor vs. ATI SIMD Engine

(extracted from AMD/ATI. ATI Stream Computing. In Advanced Micro Devices, Inc., 2009.

and

Anand Lal Shimpi & DerekWilson. Nvidia's 1.4 billion transistor gpu: Gt200 arrives as the geforce gtx 280 & 260. http://www.anandtech.com/video/showdoc.aspx?i=3334&p=2, June 2008.)

Table 5.1: NVIDIA's Multiprocessor vs. ATI's SIMD Engine

Streaming Multiprocessor	SIMD Engine
8x FP32 cores	80x FP32 cores
1x FP64 cores	16x FP64 cores
16KB shared memory	16KB local data store

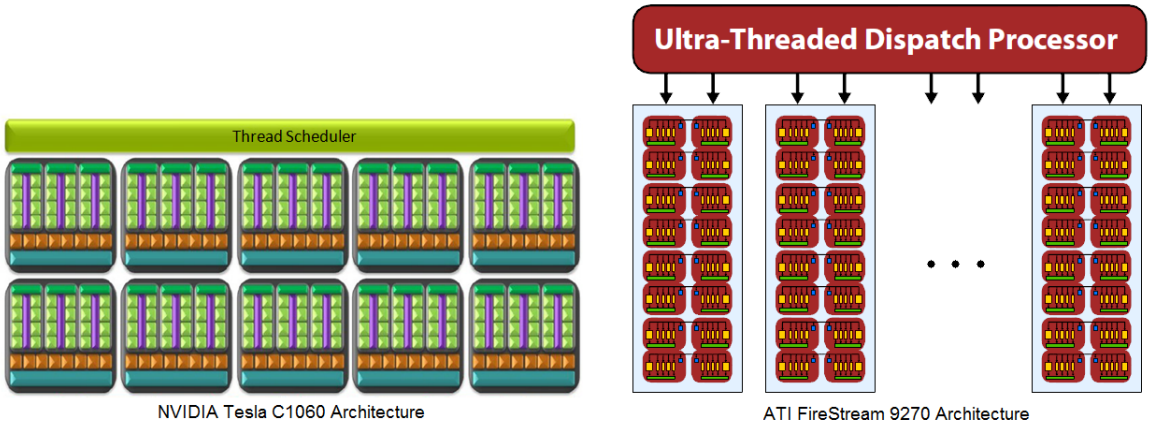


Figure 5.4: Architecture Comparison

(extracted from: AMD/ATI. ATI Stream Computing. In Advanced Micro Devices, Inc., 2009.

and

N.G.F.G.T.X. NVIDIA. 200 GPU architectural overview, second-generation unified GPU architecture for visual computing. Technical report, Tech. Rep., NVIDIA, 2008. URL www. nvidia. com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief. pdf, 2008.)

The total GPU is made up of vector processors, either 30 SMs or 10 SIMD Engines (see figure 5.4 and table 5.2). ATI has significantly more cores, but also notice that their SPs are running at almost half the clock speed. Theoretical flop performance can be calculated as follows:

$$Theoretical\ Flops = (\#\ of\ cores) \times (FP\ operations\ per\ clock\ cycle) \times (clock\ speed) \quad (5.1)$$

For instance, the FireStream 9270 has 800 cores, which can do 2 floating point operations per clock cycle, thus has theoretical flops of 800*2*0.75 = 1.2 Tflops. The Tesla card has 240 cores and *claims* its SPs are capable of performing both a MAD and a MUL instruction per clock cycle (thus 3 FP operations per clock cycle), which give 240*3*1.3 $\simeq$ 933 Gflops. However in practice NVIDIA's cards are at best only able to reach ~75% of the 933 Gflops and it is common to see the dual issue MUL instruction ignored in research papers, thus using 240*2*1.3 = 624 Gflops as the theoretical.

The dram speed of 102 GB/s and 109 GB/s may seem rather impressive, especially when considering a CPU's fastest memory, the L1 cache, only reaches top speeds of ~50 GB/s. However a caveat is in order. The GPU's dram speed is calculated as an aggregate bandwidth of data being sent to all the scalar cores in the GPU, thus to send data to just one core, would only reach speeds of $\frac{1}{240}$th or $\frac{1}{800}$th of the listed bandwidth.

NVIDIA has split up their GPUs in 3 categories: gaming (Geforce), CAD design (Quadro) and GPGPU (Tesla). All three have the same architecture however NVIDIA charges a premium for the Quadro and Tesla series. In relation to the Geforce equivalent cards, Tesla cards are ~3-4x more expensive and the Quadro are ~6-8x more expensive. The Tesla and Quadro series are workstation/server grade GPUs and thus go through far more quality assurance than the Geforce cards. According to NVIDIA's third quarter fiscal 2009 financial results, the Geforce cards were by far the highest selling

Table 5.2: Architecture Comparison

	Tesla C1060	FireStream 9270
# of Vector Processors	30 SMs	10 SIMD Engines
# of Scalar Processors	240 @ 1.3GHz	800 @ 750MHz
Single Precision	933 GigaFlops	1.2 Teraflops
Double Precision	78 GigaFlops	240 GigaFlops
Memory Bandwidth	102 GB/s	109 GB/s
Max Power Consumption	188 Watts	< 220 Watts
MSRP (USD)	~$1300	~$1100

GPUs however it was the Quadro cards that brought in the majority of the profits. Also note that ATI has the same 3 categories: gaming (Radeon), CAD design (FireGL/FirePro) and GPGPU (FireStream).

From the view point of only theoretical flops, it is clear that ATI's architecture is superior to NVIDIA's architecture. Thus, in general, algorithms that are compute bound would do better on the ATI cards (i.e. BLAS 3 routines). In fact, for dense matrix-to-matrix multiplication, it was found that on NVIDIA's GTX 280 (equiv. of Tesla C1050) reaches 375 Gflops [60]. A slightly modified[1] version of SGEMM on ATI's HD 4870 (similar to FireStream 9270) achieved as high as 880 Gflops [1], more than double NVIDIA's performance.

However, direct sparse solvers are a combination of compute bound routines (i.e. BLAS 3) and memory bound routines (i.e. triangular solve [15]). NVIDIA provides a lot more cache memory (10 times more programmable cache) than ATI allowing for larger pieces of data to be re-used. NVIDIA is also significantly increasing their L2 cache on their next generation card (Fermi) to further help algorithms sparse matrix algorithms. Fermi will also have ECC support for its RAM, whereas ATI's next generation card does not. EEC is of course a standard for server grade computing equipment.

5.2.1.2 GPU vs. CPU

This section will attempt to provide a fair comparison of the computational power of NVIDIA's Tesla C1050 and the FireStream 9270 to a server grade CPU (such as Intel's Xeon series or AMD's Opteron series). Only server grade equipment is compared, since it is assumed to be used in an electrical utility.

CPU theoretical flops can be computed similarly as that of a GPU. A single CPU core is similar to a GPU's vector processor. Current Intel cores contain 4 single precision (or 2 double precision) ALUs which execute in SIMD. Each of these ALUs are capable of performing a fused multiply-add operation each clock cycle (thus 8 single precision FP operations per clock cycle or 4 double precision FP operations per single CPU core). For instance an Intel quad core i7 running at 3.33GHz would have a theoretical double precision flops of 4*4*3.46 = 55.36 Gflops (as listed on Intel's website for the i7-975) or single precision of 111 Gflops.

[1]Input matrices required special storage. Matrices where divided into 2 parts (even and odd columns) and one matrix is stored in row-major while the other was stored in column-major [2].

A fair comparison between architectures is not easily done. Does one simply compare a single CPU to a single GPU? If such were the case, we could take the most powerful CPU being an Opteron six-core which can give up to 134.4 Gflops (theoretical single precision). In this case, the Tesla C1050 would have the equivalent computational power of ~4.6 Opteron six-core processors. However this particular Opteron processor has an MSRP of $2650 whereas the C1050 is only $1300. If we compare to a more economical quad-cores (~$200 which has 76.8Gflops), then more than 8 of these quad-core CPUs are needed to match the flops of the Tesla C1050. From this example, it becomes clear that the price must also play a role in the comparison.

Although we can find server grade quad-core CPUs starting at $200, one also needs to take into account the additional hardware required by CPUs, such as the motherboard, RAM/ROM, power supplies, fans, and rack-mount chassis. Similarly the GPUs require one or more CPUs to run them along with all the CPU's hardware components. One could conceivably put together 8 Tesla C1050s in along with a dual quad-core CPUs in a 5U chassis for roughly $13,000 (8*$1300+$2600). Similarly a cheap 1U dual quad-core CPU package can be made ~$1000. However to obtain the same theoretical flops, one would need 32 1U dual CPU racks to match the single 5U GPU rack. However the racks of CPUs will cost roughly 2.5x more than the GPUs (in this example) and consume ~3-4x more power.

Thus in terms of cost, for every GPU, one could purchase ~3.2 quad cores[2]. Thus the equivalent theoretical flops for 3.2 quad-cores is 246 Gflops where the Tesla is 624 Gflops (2.5x more) and the FireStream is 1200 Gflops (4.9x more). This reflects possible speed-ups for compute bound algorithms. However sometimes CPU based algorithms achieve higher percentages of their theoretical flops than GPUs can. For instance in Volkov's paper [60], he achieved 60% of NVIDIA's GPU flops while achieving 92% on an Intel quad-core CPU when performing matrix-to-matrix multiply. But on an ATI card, a modified version of matrix-to-matrix multiply achieved ~80% of the theoretical flops [1]. Although the GPU's memory bandwidth is also superior to the CPU, a direct comparison would be complicated as one would need to model the memory hierarchy on the CPU. Thus it is more difficult to determine how well memory bound algorithms would perform.

The prospect of hundreds of cores coupled with many research papers claiming two or even three orders of magnitude in speedup may lead to the expectation of GPUs being an extraordinary device. However from the analysis above, the raw computational power along with the price only leads to ~5x speed-up at best. Papers claiming speedups in excess of this are often comparing their GPU results to poorly coded CPU program. Such as programs that use only a single CPU core, or do not use SSE (streaming SIMD extensions), do not use optimized libraries, use double precision on CPU while using single precision on GPU, and sometimes use multiple GPUs. Ignoring such points can easily short change the CPUs results by as high as two orders of magnitude.

[2]The calculations was done using very rough figures for pricing since there are many different options when designing a server setup.

5.2.2 Programming Languages

A brief comparison of common programming languages available for NVIDIA and ATI's GPUs will be compared in this section.

CUDA (Compute Unified Device Architecture) is C style programming language that was released in November 2007 and runs only on NVIDIA hardware [44]. CUDA is the most widely used programming language for GPGPU applications. The programming forum is very active and there is plenty of support in terms of quality drivers. It is a proprietary API that is owned and controlled by NVIDIA. CUDA is currently the most mature language and is the easiest language to achieve good performance. There are many high performance libraries readily available (i.e. BLAS, FFT, along with many other open source programs released by the large programming community). NVIDIA does not release the actual ISA (instruction set architecture) of its graphics cards, however they do provide and virtual ISA called PTX (Parallel Thread Execution) which is an assembly level language [43]. However, a third party tool (not supported by NVIDIA) called decuda is often used in many research papers which allows the programmer to convert the compiler's machine code into PTX instructions.

ATI graphics cards have Brook+, a C level programming language originally developed by a research team at Standford University. Similar to NVIDIA's virtual ISA language, ATI has IL (Intermediate Language). However ATI also goes beyond the virtual ISA and publishes the actual ISAs as well. The vendor provides limited library support through ACML (AMD Core Math Library), which contains extensions for its GPUs (currently it only provides SGEMM and DGEMM). It should be noted that ATI has pledged to support OpenCL (to be discussed next) which may lead to Brook+ becoming obsolete.

In addition to vendor specific languages, there is also cross-platform language called OpenCL (Open Computing Language) that is compatible with both NVIDIA and ATI cards. OpenCL is a newer language and is modeled after the CUDA driver API. As the name suggests, the language is meant to be an open standard and is managed by a non-profit consortium (The Khronos Group). CUDA on the other hand is a proprietary language and is controlled by NVIDIA, which is why ATI refuses support the language on its GPUs. Generally, on NVIDIA cards, CUDA programs will execute slightly faster than OpenCL since CUDA is specifically designed for the NVDIA hardware.

From the perspective of someone beginning with GPU programming, it would seem that CUDA would be the best choice. The language is by far the most popular and thus has the most resources (i.e. forums, research papers, math libraries...) and isn't expected to go away any time soon.

5.2.3 Summary

ATI appears to have the advantage in terms of hardware (in terms of theoretical flops), but NVIDIA has the clear advantage in software. An ATI card has double the computational power of an NVIDIA card but NVIDIA has a lot more cache space that would benefit sparse matrix algorithms. NVIDIA's programming language CUDA is very easy to use and thus has gained the most programmers. This has led to extensive available resources for NVIDIA leaving ATI trailing behind. From the perspective of someone beginning GPU programming, NVIDIA would be the clear choice and thus the theory and designs discussed in this thesis will be with reference to NVIDIA graphics cards. However, since MATE

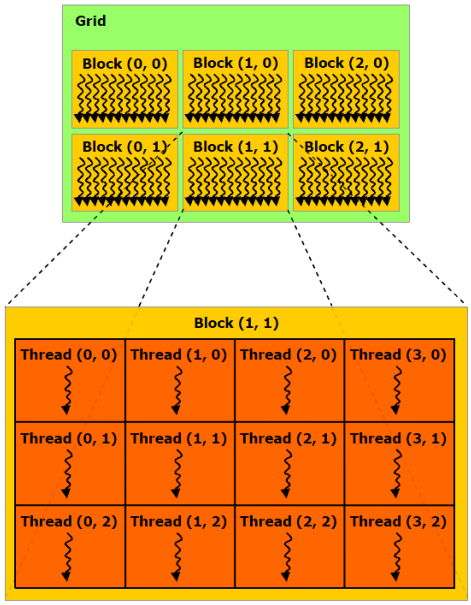


Figure 5.5: Blocks and Threads Abstraction
(extracted from: NVIDIA Corporation. NVIDIA CUDA Programming Guide, 3.1.1 edition, July 2010.)

is not purely compute bound a clear choice cannot be made simply on specifications alone. One would need to be implement MATE on both machines to truly determine which GPU would work best.

5.3 GPU Programming

This section is split into two parts which cover some key concepts necessary to achieve a decent percentage of the hardware's theoretical peak flops. The first part of this section will discuss the threads and blocks abstraction along with an example illustrating the vector nature of the GPU. The second portion will deal with the pipeline concept and the importance of hiding latencies of both arithmetic and move instructions.

5.3.1 Programming Model

The GPU's hardware (shown in figures 5.3 and 5.4) is composed of streaming multiprocessors (SM) which are themselves composed of 8 scalar processors. The most important abstraction in GPU programming is the concept of blocks and threads (see figure 5.5). Each SM is assigned one or more blocks. A block is then made up of many threads. Each scalar processor is assigned multiple threads. The exact block-to-SM or thread-to-processor mapping is hidden from the programmer. This abstraction is necessary to facilitate distribution of the computation amongst the hundreds of processors on the graphics card. Thus all programming is structured in terms of blocks and threads. The number of blocks on an SM and the number of threads in a block make up the *execution parameters*. The execution parameters are always specified at the start of any GPU program.

The threads belonging to a block operate in SIMD (Single Instruction, Multiple Data). More specif-

ically, a single instruction is made of up 32 threads (called a *warp*). Therefore a single scalar addition (i.e. equation 5.2) would actually be vectorized into 32 redundant additions shown in equation 5.3. Thus despite the SIMD environment, scalar operations can be forced, however they can be very wasteful in terms of useful computation ($\frac{1}{32}$ or ~3% efficiency). This highlights the importance of orienting code towards vector operations.

$$\begin{aligned} a &= 1; \\ b &= 2; \\ c &= a + b; \end{aligned} \tag{5.2}$$

$$\underbrace{\begin{bmatrix} \\ C \\ \\ \end{bmatrix}}_{32\ elements} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} + \begin{bmatrix} 2 \\ 2 \\ \vdots \\ 2 \end{bmatrix} \tag{5.3}$$

5.3.2 Pipeline Latency and Throughput

To code efficiently, it is important to know how long basic instructions (data movement or arithmetic) take to execute. To understand execution time, a familiarization with pipeline latency and throughput are a necessity. This section contains a brief description of the pipeline concept, common measures of throughput/latency on a GPU and typical values of throughputs and latencies for various arithmetic and move instructions.

To perform a floating point operation (such as add, subtract, multiply, etc.), the data passes through various *stages* of computation until the final result is achieved. This is known as a pipeline. The time it takes for one set of data to travel through the entire pipeline from start to finish is called the pipeline latency. There are typically many sets of data at various stages of a pipeline at a given time. Thus when a pipeline is fully saturated, instructions can effectively finish much faster than the overall latency; which is known as throughput. Ideally, the programmer would want to hide all latencies with useful computation so all that is seen is the instruction throughput.

Since instructions are issued for groups of 32 threads, a common measure of throughput and latency is in terms of clock cycles and warps per multiprocessor. The fastest a thread can execute is one clock cycle per processor. Since there are 32 threads in a warp and 8 processors in a multiprocessor, the fastest a warp can finish is 4 clock cycles. In this section both latency and throughput will be stated in terms of clock cycles per warp per multiprocessor.

Typical throughputs of single precision floating point arithmetic *instructions* such as (ADD, SUB, MUL, MAD[3]) can be executed every 4 clock cycles (i.e 1 *operation* every clock cycle per processor). Other arithmetic operations such as division or square root can take many more clock cycles to complete

[3] A MAD instruction performs a multiply-add operation in the form of $d = a \times b + c$

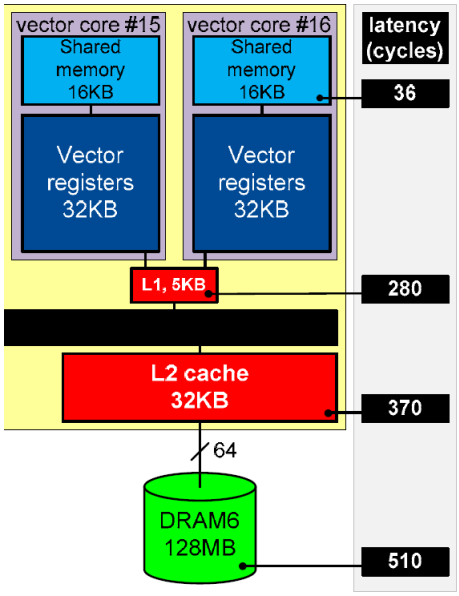


Figure 5.6: Memory Latencies

(extracted from: V. Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing, pages 1–11. © 2008 IEEE)

(i.e. DIV can take up to 36 clock cycles). However these throughputs actually assume all operands are stored within registers. Although the CUDA programming manual [44] claims that shared memory (L1 cache) is as fast as registers, it was found in [60] that MAD instructions took up to 6 clock cycles to execute when one of the operands are stored in shared memory.

Throughputs of data movement instructions, whether from registers, cache or DRAM only take 4 clock cycles. However it is the latency of the move instructions that the programmer must take into careful consideration.

For arithmetic floating point operations, pipeline latency for simple arithmetic instructions such as ADD, MUL, or MAD is ~24 clock cycles and is typically referred to as read-after-write latency. Thus to hide this latency, it is recommended by the programming manual ([44] section 5.2) to schedule at least 6 warps (192 threads) of instructions to hide this latency (since $6 \times 4 = 24$ clock cycles). The warps can be contiguous or segmented. For instances operations with vector size of 192 elements or 3 independent vectors of length 64 elements. Both cases achieve the necessary independent operations to hide the read-after-write latency.

The memory latency of instructions transferring data from various memory locations to a multiprocessor's register space is shown in figure 5.6. The benchmarking for figure 5.6 is described in [60] and are performed on a 8800GTX. The largest area of memory, the DRAM (which houses as much 4 GB on the C1050) has the largest latency of ~510 clock cycles and closely matches the programming manual's 400-600 clock cycle latency ([44] section 5.1.1.3). Just like the arithmetic latency, memory latency can be hidden by issuing useful arithmetic or memory instructions.

5.4 Test Case - Small Matrix Multiplication Routine

This section will demonstrate how the most common computation from the multi-level MATE algorithm can be done on the GPU. From the previous chapter, we see that the majority of the computation is spent performing small matrix multiplication. Similarly, modern sparse solvers (e.g. MUMPS, SuperLU, UMFPACK, etc.) off load the bulk of their computation to BLAS routines (optimized linear algebra libraries), where GEMM calls also consume the lion's share of the execution time.

Thus one of the first steps to realizing multi-level MATE on the GPU is ensuring there are efficient BLAS routines capable of handling the algorithm's computations. Although NVIDIA does provide BLAS routines, they are designed for very large matrices. In fact, the SGEMM routine only achieve peak performance for matrices of size 4096×4096 and larger. Moreover, the granularity of the SGEMM routine itself is significantly larger than the 3×3 matrices used in the proposed algorithm (SGEMM uses panels of size 16×64 [60]). Thus there is a need for a multiplication routine that is designed to handle the many 3×3 matrices from the multi-level MATE algorithm while achieving similar computational benchmarks as standard GPU BLAS routines. Such metric would provide insight to the viability of the proposed algorithm on the GPU.

This section divided into four parts to provide an in-depth analysis to the design and implementation of a multiplication routine suited for mutli-level MATE algorithm. The first part will discuss how the standard row/column major data storage is not efficient for the small matrices in the multi-level MATE algorithm, thus another data storage scheme is proposed. The next part will attempt to provide a thorough understanding behind the design choices in the routine. The third part goes into the clock cycle analysis of the instructions in the routine which will provide an accurate prediction of the measured results. The final portion provides the experimental setup along with the benchmarked results.

5.4.1 Small Matrix Representation (SMR)

Based on the GPU programming model, perhaps the simplest approach to accomplishing dense matrix-to-matrix multiplication is to just multiply the 3×3 or 5×5 matrices independently of each other. Standard matrix multiplication can be employed by assigning one *warp* of threads for each two pairs of matrices that are to be multiplied together. Thus using the common row-major or column-major layout would be appropriate to accomplish this task.

However such a method suffers from various inefficiencies. The number of independent operations to perform matrix multiplication of either 3×3 or 5×5 matrices are not a multiple of 32 (a warp contains 32 threads). Hence poor thread utilization will result in many wasted arithmetic and data movement operations. As a further consequence, improper memory access patterns will arise and reduce effective memory bandwidth. Finally, the use of just one warp per matrix operation will cause further problems in the form of register memory bank conflicts ([44] section 5.1.2.6). With all the efficiencies, it is clear that such a simple approach would be a viable candidate.

To overcome these problems, a small matrix representation (SMR) is proposed. The representation goes a step further by using the data from a large groups of matrices together as oppose to just focusing

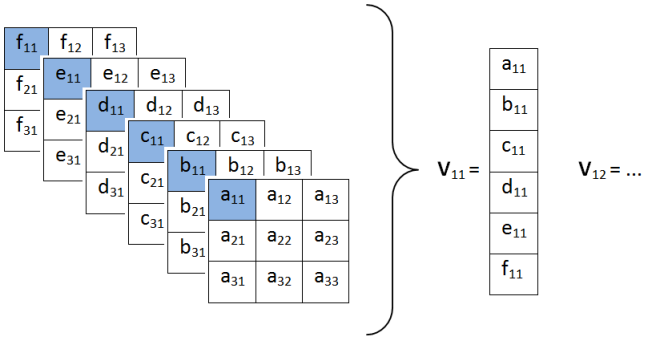


Figure 5.7: Small Matrix Representation

on 2 matrices at a time. Many large groups of matrices can be formed since large power networks can be partitioned into thousands of subsystems. For instance, if a 15000×15000 matrix is decomposed into 3x3 subsystems, it would result in 3000 subsystems. Furthermore, the situation of having only 3000 subsystems is actually the worst case scenario. As discussed in chapter 4, the elimination matrices (also 3×3 or 5×5) can outnumber the subsystem matrices by an order of magnitude (depending on how many levels are created), thus providing a plentiful amount of matrices to achieve this proposed representation. For large enough power systems, two representations of 3×3 or 5×5 matrices may be able to be supported concurrently.

Just as matrix-to-matrix multiplication requires two matrices to perform the operation, the proposed representation requires two *groups* of matrices. Both groups are composed of the same number of 3x3 or 5x5 matrices. For a given group, the set of matrices are decomposed into vectors. These vectors are formed by taking a single element (i, j) from every matrix in the group (illustrated in figure 5.7 for (1, 1) element vector). As a consequence, if a group consists of $n \times n$ matrices, the group will be represented n^2 vectors (thus a 3×3 matrix group will have 9 vectors).

There are several advantages to this representation. By just properly choosing the right number of matrices to put in a group the inefficiencies regarding low thread utilization, memory access pattern and latencies can be alleviated. But perhaps the most notable advantage is the ability to control individual elements as if performing scalar operations. Even the most challenging computations such as small matrix factorization can easily be done while maintaining vectorized calculations. The only requirement is that each matrix of a group requires the exact same operation performed, which is why sparse operations on a group of matrices are not possible.

5.4.2 Design of Small Matrix Multiply Routine

The small matrix multiply routine consists of 3 sections: (1) Load data into registers, (2) matrix multiplication using SMR, and (3) sending solution data from registers to cache. This section will provide insight behind the choices made to implement the matrix multiply routine.

In the **first section**, data is assumed to be stored in cache from previous operations since matrix multiplication is repeated many times in the multi-level MATE algorithm as discussed in chapter 4. The purpose of the first stage is to transfer data from cache to the registers since computations are performed

faster when all operands are stored in registers as oppose to cache [60]. In fact, this technique was used in a recent SGEMM routine designed on a NVIDIA GPU and was found to be 10-20% faster than the vendor's original SGEMM [14]. Furthermore this operation also helps reduce register memory usage which aids in reducing latencies.

The algorithm uses row or column major ordering (there is no computational advantage choosing one or the other in the scope of this routine) to store individual 3×3 matrix data consecutively in both DRAM and cache memory. Each time data is transferred from cache memory to registers, the data is converted into the SMR vectors as depicted in figure 5.7. The conversion from row/column major ordering into SMR vectors requires strided memory access. Strided access from cache memory incurs no penalty in transfer bandwidth for certain values of stride ([44] section 5.1.2.5 which discusses that the best strides are those that do not cause bank conflicts). The specific application of converting 3×3 matrix data into SMR vectors requires a stride of nine, which fortunately does not cause bank conflicts. Also noteworthy is fact that 4×4 matrix conversion will suffer from bank conflicts however 5×5 matrices will not. It should also be noted that heavy penalties would be incurred if strided memory access were performed to or from the GPU's DRAM.

The **second section** performs the actual computation. Since the data is now stored as SMR vectors within the registers of the GPU, the calculation is rather straightforward. The representation allows for direct manipulation of individual elements of the 3×3 matrices (while still maintaining vector operations) thus the calculation for each of the nine SMR solution vectors can be performed as shown in equation 5.4.

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} \tag{5.4}$$

This operation is split into 3 stages (see equation 5.5). Notice that the solution of the first stage is *dependent* for the next, thus read-after-write latencies (24 clock cycles) will be present. As discussed in section 5.3.2, a minimum of 192 threads (or vector elements) running *independent* computation (e.g. running all stage 1 computation first) is required to hide this latency. It is also possible to achieve the same goal by spreading out the independent computation amongst smaller segments of blocks with threads that sum to 192 (i.e. running three blocks of 64 threads). This is a much more favorable choice since it provides a higher granularity when dealing with SMR vectors which allows for more even distribution of computation among the multiprocessors.

$$\begin{array}{l} stage\,1:\ c_{ij}^{(1)} = a_{i1}b_{1j} \\ \hline stage\,2:\ c_{ij}^{(2)} = a_{i2}b_{2j} + c_{ij}^{(1)} \\ \hline stage\,3:\ c_{ij}^{(3)} = a_{i3}b_{3j} + c_{ij}^{(2)} \end{array} \tag{5.5}$$

Unfortunately hiding latencies with memory transfers seem to behave differently than latencies with computation. As mentioned above, 192 threads executing independent computation is necessary to hide read-after-write latency. For *computation*, there was some freedom in the thread/block allocation (i.e. using one block of 192 threads or 3 blocks of 64 threads would accomplish the same goal). However for *memory operations*, it behaves as if there were "block synchronization" whereby the compiler appears to

```
__global__ void min_ops( float *gA, float *gB )
{
    int i;
    const int tx = threadIdx.x;
    volatile float a[9], b[9];
    volatile float c[9] = {0,0,0,0,0,0,0,0,0};
    volatile __shared__ float sA[NUM_ELEMENTS], sB[NUM_ELEMENTS];

    #pragma unroll 20
    for ( i = 0 ; i < TEST_ITERATIONS ; i ++ )
    {
        smem2reg(a, b, sA, sB, tx);
        __syncthreads();
        mad_ops(a, b, c);
        reg2smem(c, sA, tx);
    }
    __syncthreads();
    reg2gmem(c, gA, tx);
}
```

Figure 5.8: Matrix Multiplication Kernel

threat blocks as dependent operations. Hence choosing blocks of 64 threads caused significant latencies with memory transfers and thus blocks of 192 threads were chosen for the small matrix multiply kernel.

The **third section** simply sends the result from the matrix multiplication group from the registers to the shared memory. This operation again uses strided access to return the data from SMR form to row/column major layout. Thus there is no issue with data transfer when it is eventually transferred into the GPU's DRAM.

As mentioned in section 5.3.1, all routines require execution parameters to specify the distribution of the computation on the GPU. The number of threads per block was chosen as 192 and was discussed in the second section of the multiply routine. One block per multiprocessor was chosen in order to reduce inter-block communication, which is required after the elimination of each level in the multi-level MATE algorithm. The block abstraction enforces that the data from each block is not directly accessible from another block, even if they happen to reside on the same multiprocessor. Thus if one multiprocessor had two blocks, communication between the two blocks would require sending data to the GPU's DRAM and back (~700 clock cycle latency each way) even though data from each block is already stored in the same cache memory. Figure 5.8 contains the code the multiplication routine outlined in this section. Full listing of the code can be found in appendix C.

5.4.3 Theoretical Results

This section will perform a clock cycle analysis of the routine to determine how close to the GPU's theoretical flops the matrix multiply routine is able to achieve. In the following analysis, a ratio will be taken of the arithmetic instructions to the total instructions to determine how close to the theoretical flops the algorithm comes to. Since it is a ratio, the actual length of the SMR vectors (or number of blocks per thread) is irrelevant.

There are two types of instructions that are used for this multiply routine, arithmetic and move

instructions. For move instructions, the multiply routine moves matrix groups A and B into shared memory, and then moves matrix C out of shared memory. Since the elements in a matrix group are SMR vectors, we know there are n^2 SMR vector move instructions required to move 1 group matrix, thus there are a total of $3n^2$ move instructions in the multiply routine. For arithmetic instructions, the computation of the first SMR vector (c_{11}) can be computed as shown in equation 5.6 (assuming small matrices are of size 3×3). Thus to obtain a given solution element c_{ij} there are:

- 1 multiply instruction
- (n-1) multiply-add instructions

Since there are n^2 elements in the solution matrix, the total number of arithmetic instructions are n^2 MUL instructions and $(n-1) \cdot n^2$ MAD instructions to perform a matrix to matrix multiplication.

$$\begin{aligned} c_{11} &= \underbrace{a_{11}b_{11}}_{MUL} + a_{12}b_{21} + a_{13}b_{31} \\ c_{11} &= \underbrace{c^{(1)} + a_{12}b_{21}}_{MAD} + a_{13}b_{31} \\ c_{11} &= \underbrace{c^{(2)} + a_{13}b_{31}}_{MAD} \end{aligned} \tag{5.6}$$

Then by taking the ratio of arithmetic instructions to total instructions (i.e. arithmetic + move instructions) we can determine the percentage of the theoretical flops that the algorithm is capable of achieving. This is possible for this routine because both the arithmetic and move instructions theoretically take the same about of time to execute (4 clock cycles per warp assuming latencies are hidden). However there is a distinction that needs to be made between the arithmetic instructions. A scalar processor is capable of performing a multiply-add operation (i.e. $d = a * b + c$) every clock cycle, which is 2 floating point operations. But if just a multiply operation is done (i.e. $d = a * b$) , then only 1 floating point operation is being performed, which is only 50% of what the GPU is capable of doing on a clock cycle. Thus the following formula can be constructed:

$$\%\, theoretical\ flops = \frac{\frac{1}{2}\,(\#\ of\ MUL) + (\#\ of\ MAD)}{total\ (arithmetic + move)\ instructions} \times 100\% \tag{5.7}$$

Then substituting in the arithmetic and move instructions for the small matrix multiplication routine:

$$\%\, theoretical\ flops = \frac{\frac{1}{2}\left(n^2\right) + ((n-1) \cdot n^2)}{(n^3) + (3n^2)} \times 100\% \tag{5.8}$$

$$\%\, theoretical\ flops = \frac{n^3 - \frac{1}{2}n^2}{(n^3) + (3n^2)} \times 100\% \tag{5.9}$$

Table 5.3 shows performances for various subsystem matrix sizes. A trade-off becomes evident, the larger we choose the subsystem matrices, the better performance on the GPU (in terms of flops). However the larger the subsystem matrices are, the more fill-in is incurred thus increasing execution time. The analysis from chapter 4 indicates a huge increase in total operations from just minor increases

Table 5.3: Predicted Performance

Matrix Size	% of theoretical flops
3x3	41.7%
5x5	56.3%
7x7	65.0%
15x15	80.6%

in subsystem size, thus the target size is around 3×3 or 5×5. Table 5.3 indicates performances of 41.7% to 56.3%. As a comparison, the previous NVIDIA GPU BLAS implementation which would only achieve 36-44%, the newer version currently runs at 58-60%.

5.4.4 Measured Results

This section will discuss the benchmarking setup followed by the results. Also a closer look at the code is taken to get a more accurate clock cycle analysis.

The experiment was run on GT 9800 card. It contains 112 cores (14 multiprocessor) running at a frequency of 1.62GHz providing a theoretical flops of 363 Gflops. The code assumes 3×3 subsystem matrices using 192 threads per block, running 1 block per multiprocessor. Thus each SMR vector is of length 192, and each multiprocessor is handling 192 matrices from both matrix groups A and B. The multiplication routine's kernel is shown figure 5.8 (full code in appendix C). The code was iterated one million times to achieve an accurate measurement and was unrolled 20 times ('20' was chosen because the code nicely fit into the GPU's instruction memory). Measured flop rate was determined from the total number of floating point operations divided by the execution time and was found to be **135 Gflops** or **37.2%** of the theoretical 363 Gflops.

A closer instruction analysis can reveal several issues that are the cause of discrepancy with the measured result (37.2%) and theoretical result (41.7%).

I. Memory Transfer Efficiency. When independently testing shared memory-to-register transfers using 192 threads per block (1 block per multiprocessor), it was found that only 88% of the theoretical bandwidth was achieved.

II. Additional Instructions. The multiplication kernel contains one synchronization instruction that was not previously accounted for. Upon inspection of the assembly code (using decuda[4], a popular third party disassembler) we also see there is an additional address calculation being performed.

III. MAD Efficiency. The testing MAD throughput under ideal conditions using 192 threads per block, was found to have an efficiency of 98.6%

The main culprit behind the discrepancy is clearly latencies that haven't been completely hidden. These inefficiencies can be modeled by taking equation 5.9 and dividing each instruction term by their actual

[4]http://wiki.github.com/laanwj/decuda

Table 5.4: Modified Predicted Performance

Matrix Size	% of theoretical flops
3x3	37.5%
5x5	52.6%
7x7	61.6%
15x15	77.8%

efficiency to reflect additional time to complete each instruction. Also the additional instructions that were not originally included are added in by simply placing a '+2' to the denominator. The corresponding results for the new predicted results are shown in table 5.4 and now shows 37.5% as the theoretical flops, a much closer value to the measured result.

$$\%\, theoretical\ flops\ (adjusted) = \frac{\left(n^3 - \frac{1}{2}n^2\right)}{\left(n^3\right)/0.986 + \left(3n^2\right)/0.88 + 2} \times 100\% \tag{5.10}$$

5.5 Summary

In this chapter, a custom small matrix-to-matrix multiplication routine was implemented for the multi-level MATE algorithm. A data representation for MATE was also proposed which allowed to keep all processors busy while avoiding the majority of latencies encountered during the multiplication routine. The data representation also enables easy access and manipulation of individual elements within a sub-system matrix. Thus complicated routines such as small matrix factorization can be easily implemented without having to worry about hiding latencies. However this data representation does have its own downsides. The representation necessitates the need for uniformity of all subsystem and link matrix quantities. Thus smaller matrices will need to be padded with zeros and hence computation will be wasted in this case. The final benchmarked result for the small matrix multiplication routine was able to successfully achieve ~40% of the GPU's theoretical flops.

Chapter 6

Conclusion

The first step towards verifying the validity of the multi-level MATE algorithm on the GPU is to have a BLAS that can efficiently handle the necessary computation. A BLAS routine needed to be designed because standard matrix multiplication BLAS on the GPU is designed for large matrices and achieves the best performance for matrices of size 4096x4096 and over. However, the proposed algorithm is concerned with matrices of size 3x3. The implemented small matrix multiplication routine was able to successfully achieve ~40% of the GPUs peak flops while the regular large matrix multiplication routine reaches ~60%. Thus providing the possibility for multi-level MATE algorithm to transition to the GPU.

However, the small matrix multiplication imposes limitations to the MATE algorithm. The described algorithm assumes the sparse matrix will fit nicely into the described model. Ideally, allowing the algorithm to *bend and mold* to the inputted sparse matrix would provide an overall reduction in the amount of computation required to solve the system. For instance, using different size subsystems or occasionally partitioning a subnetwork at a certain level into pieces of three instead of the usual two. However straying away from a simple and uniform approach can cause a serious reduction in the algorithm's performance on the GPU. The small matrix representation uses the uniformity for easy distribution of computation onto the GPU's processors while avoiding various latencies. If the algorithm assumes 3x3 subsystems but a few the subsystems have a final size of 2x2, the matrix multiply routine will pad the 2x2 matrices with zeros to make it 3x3. Thus to allow the MATE algorithm to be more flexible results is wasted computation.

On the subject of the power flow implementations and comparisons, the programs did not contain any control adjustments. Although their inclusion would have provided a more accurate comparison, the main goal was to create a current equation power flow program that executed as fast as was published in the aforementioned papers. These programs were later used to perform iteration profiling to identify which sections of the program where speedup and which portions where slowed down. Thus the control adjustment's role was not necessarily central to the main argument.

Lastly, the proposed multi-level MATE algorithm is specifically designed to take advantage of the computational horsepower of the GPU. The choice of small 3x3 subsystem matrices fits well on the GPU based on preliminary test results. The small subsystem size also allows them to be treated as dense matrices thus removing the burden of sparsity to the matrix partitioning software. However by

modeling all the matrices in the simulation as dense, there is an associated computational cost. Thus the expectation is that this cost is outweighed by the ability to utilize many processors effectively. Therefore, the proposed algorithm would not be well suited on CPUs as it is currently described. From the literature review, a multi-level algorithm appears to be the best direct solver approach for a cluster of CPUs. In contrast to the proposed MATE algorithm, the goal of partitioning for a CPU cluster should be to split a system into only as many subsystems as there are available processors. Subsystem sparsity should also be retained since the CPU provides more programming flexibility than the GPU.

6.1 Summary of Thesis Contributions

This thesis has made the following contributions:

- **Unsymmetrical Generalized Link Equation**

  Allows for the application of MATE into power flow through a similar means as MATE in EMTP. It enables the decoupling of an unsymmetrical branch using only one equation. Two types of branch tearing techniques where provided. A discussion covered the trade-offs of using one version over the other when decoupling different types of nodes.

- **Current Equation Power Flow Implementation and Performance**

  Proposed a condensed formulation (described in section 3.2.4.2) using the updated PV bus representation found in [24]. Provided explicit descriptions of various current equation implementations that have not been described in research papers. Provided an extensive argument against the performance current equation PF performance claimed in several IEEE publications. The argument was supported through benchmarking and iteration profiling of multiple implementations of the current equation algorithm which was compared to the conventional power flow algorithm.

- **Multi-level MATE Algorithm for Massively Parallel Computing Architectures**

  Exploits the multi-level concept by using many levels to allow for the parallel computation of the link matrix. This allows for mass partitioning of the original system matrix. A simple model is provided which is used to give a flop analysis to determine the distribution of floating point operations. Two versions are provided, one using branch tearing and the other using node tearing.

- **GPU Data Representation and Matrix Multiplication Routine**

  Small matrix representation provides a framework to perform virtually all of the operations of the multi-level MATE algorithm in an efficient and uniform manner across all the cores on the GPU. A big advantage to the representation is its ability to manipulate individual elements easily, thus the implementation a dense small matrix LU factorization would be relatively simple. This data representation was tested via implementation and benchmarking of the small matrix multiplication routine.

6.2 Future Work

The following are outstanding issues recommended for future work:

- Full implementation of the multi-level MATE algorithm on the GPU. A matrix partitioner such as METIS can be used to iteratively split the matrices into pieces of two until the desired subsystem size is reached. The small matrix representation can similarly be applied to vectors to perform all operations necessary for the multi-level algorithm.

  Many design considerations must still be taken into account. Such as when to load data from DRAM or cache into registers. Also, deciding how much data to load into the cache or registers and ensuring all these latencies are either completely hidden or kept to a minimum.

  Once the main structure of the implementation is complete, it may be prudent to try various types of partitioning. For instance, both versions of the multi-level MATE algorithm should be implemented to determine which is best (i.e. branch tearing and node tearing). Furthermore, node tearing can be further subdivided into vertex-based and edge-based partitioning.

- Algorithm considerations. As mentioned at the start of this chapter, the small matrix representation limits the flexibility of the MATE algorithm on the GPU. If the representation can be modified or changed all together to allow for the algorithm to better accommodate the sparsity pattern of a power system matrix, it would have the potential to save a substantial amount of computation.

- Applications beyond power flow. For instance, the transient stability program is a natural transition. The program can potentially be used as a general sparse solver should the performance of the program prove to be competitive with current sparse solvers on the GPU.

Bibliography

[1] Faster dense matrix-matrix products on ati hardware. http://forum.beyond3d.com/showthread.php?t=54842, August 2009.

[2] Opencl blas (sgemm) performance on radeon 4000 and 5000 series. http://forums.amd.com/forum/messageview.cfm?catid=390&threadid=127963&enterthread=y, February 2010.

[3] S. Acevedo, LR Linares, JR Martí, and Y. Fujimoto. Efficient HVDC converter model for real time transients simulation. *IEEE Transactions on Power Systems*, 14(1):166–171, 1999.

[4] M.L. Armstrong. *Multilevel MATE Algorithm for the Simulation of Power System Transients with the OVNI Simulator*. PhD thesis, The University of Bristish Columbia, 2006.

[5] M.L. Armstrong, J.R. Marti, L.R. Linares, and P. Kundur. Multilevel MATE for efficient simultaneous solution of control systems and nonlinearities in the OVNI simulator. *IEEE Transactions on Power Systems*, 21(3):1250–1259, 2006.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[7] A. Asgari and JE Tate. Implementing the chebyshev polynomial preconditioner for the iterative solution of linear systems on massively parallel graphics processors. In *CIGRÉ Canada. Conference on Power Systems*, 2009.

[8] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. *NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004*, 2008.

[9] M.J.; Kornhuber R.; Widlund O. (Eds.) Bercovier, M.; Gander. *Domain Decomposition Methods in Science and Engineering XVIII*. Springer Publishing Company, Incorporated, 2009.

[10] H.E. Brown, G.K. Carter, H.H. Happ, and C.E. Person. Power flow solution by impedance matrix iterative method. *IEEE Trans. Power Apparatus and Systems*, 82:1–10, 1963.

[11] L. Buatois, G. Caumon, and B. Lévy. Concurrent number cruncher: a GPU implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems*, 24(3):205–223, 2009.

[12] J.S. Chai and A. Bose. Bottlenecks in parallel algorithms for power system stabilityanalysis. *IEEE Transactions on Power Systems*, 8(1):9–15, 1993.

[13] S.D. Chen. A study based on the factorization-tree approach for parallel solution of power network equations. *Electric Power Systems Research*, 72(3):253–260, 2004.

[14] LS Chien. Hand-tuned sgemm on gt200 gpu. http://forums.nvidia.com/index.php?showtopic=159033, February 2010.

[15] P. Cicotti, X.S. Li, and S.B. Baden. LUsim: A Framework for Simulation-Based Performance Modeling and Prediction of Parallel Sparse LU Factorization. *Lawrence Berkeley National Laboratory*, 2008.

[16] V.M. Da Costa, N. Martins, and JLR Pereira. Developments in the Newton Raphson power flow formulation based oncurrent injections. *IEEE Transactions on power systems*, 14(4):1320–1326, 1999.

[17] V.M. Da Costa, J.L.R. Pereira, and N. Martins. An augmented Newton-Raphson power flow formulation based on current injections. *International journal of electrical power & energy systems*, 23(4):305–312, 2001.

[18] J.W. Demmel, S.C. Eisenstat, J.R. Gilbert, X.S. Li, and J.W.H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[19] H. Dommel. Presentation: "power flow, optimal power flow, and sensitivity analysis.". *UBC*, May 2007.

[20] H.W. Dommel, W.F. Tinney, and W.L. Powell. Further developments in Newton's method for power system applications. In *IEEE Winter Power Meeting, Paper*, volume 70, 1970.

[21] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct methods for sparse matrices*. Oxford University Press, USA, 1989.

[22] O.I. Elgerd. *Electric energy systems theory*. McGraw-Hill New York, 1971.

[23] P.A.N. Garcia, J.L.R. Pereira, S. Carneiro Jr, V.M. Da Costa, and N. Martins. Three-phase power flow calculations using the current injectionmethod. *IEEE Transactions on Power Systems*, 15(2):508–514, 2000.

[24] P.A.N. Garcia, J.L.R. Pereira, S. Carneiro Jr, M.P. Vinagre, and F.V. Gomes. Improvements in the representation of PV buses on three-phase distribution power flow. *IEEE Transactions on Power Delivery*, 19(2):894–896, 2004.

[25] F.V. Gomes, S. Carneiro Jr, J.L.R. Pereira, M.P. Vinagre, P.A.N. Garcia, and L.R. Araujo. A new heuristic reconfiguration algorithm for large distribution systems. In *IEEE Power Engineering Society General Meeting, 2006*, page 1, 2006.

[26] J.J. Grainger and W.D. Stevenson. *Power systems analysis*. McGraw-Hill, Inc., 1994.

[27] L.L. Grigsby. *Power system stability and control*. CRC, 2007.

[28] H.H. Happ. Multi-Level Tearing and Applications. *IEEE Transactions on Power Apparatus and Systems*, 92:725–733, 1973.

[29] J. Hollman. *Step by Step Analysis with EMTP Discrete Time Solutions*. PhD thesis, The University of British Columbia, Vancouver, 2006.

[30] V. Jalili-Marandi and V. Dinavahi. Large-Scale Transient Stability Simulation on Graphics Processing Units.

[31] J. Johnson, T. Chagnon, P. Vachranukunkiet, P. Nagvajara, and C. Nwankpa. Sparse LU decomposition using FPGA. In *International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2008.

[32] DP Koester, S. Ranka, and GC Fox. Parallel block-diagonal-bordered sparse linear solvers for electrical power system applications. In *Scalable Parallel Libraries Conference*. Citeseer, 1993.

[33] G.P. Krawezik and G. Poole. Accelerating the ANSYS Direct Sparse Solver with GPUs.

[34] G. Kron. A set of principles to interconnect the solutions of physical systems. *Journal of Applied Physics*, 24:965, 1953.

[35] C.H. Lai. Diakoptics, domain decomposition and parallel computing. *The Computer Journal*, 37(10):840–846, 1994.

[36] M.; Keyes D.; Widlund O.; Zulehner W. (Eds.) Langer, U.; Discacciati. *Domain Decomposition Methods in Science and Engineering XVIII*. Springer Publishing Company, Incorporated, 2008.

[37] C. Lin and L. Snyder. *Principles of parallel programming*. Addison-Wesley Publishing Company, USA, 2008.

[38] L. R. Linares. *OVNI (Object Virtual Network Integrator) a new fast algorithm for the simulation of very large electric networks in real time*. PhD thesis, The University of British Columbia, Vancouver, 2000.

[39] R.F. Lucas, G. Wagenbreth, and D.M. Davis. Implementing a GPU-Enhanced Cluster for Large-Scale Simulations. In *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*. Citeseer, 2007.

[40] F. Maghsoodlou, R. Masiello, T. Ray, K. Inc, and N. Arnhem. Energy management systems. *IEEE Power and Energy Magazine*, 2(5):49–57, 2004.

[41] J.R. Martí, L.R. Linares, J.A. Hollman, and F.A. Moreira. Ovni: integrated software/hardware solution for real-time simulation of large power systems. In *Proceedings of the 14th Power Systems Computer Conference (PSCC 2002)*, 2002.

[42] K. Morison, L. Wang, and P. Kundur. Power system security assessment. *IEEE Power and Energy Magazine*, 2(5):30–39, 2004.

[43] NVIDIA Corporation. *NVIDIA Compute PTX: Parallel Thread Execution*, 1.2 edition, June 2008.

[44] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 3.1.1 edition, July 2010.

[45] University of Washington. Power systems test case archive. http://www.ee.washington.edu/research/pstca/, 1993.

[46] D.R.R. Penido, L.R. Araujo, J.L.R. Pereira, P.A.N. Garcia, and S. Carneiro Jr. Four wire Newton-Raphson power flow based on the current injection method. In *IEEE PES Power Systems Conference and Exposition, 2004*, pages 239–242, 2004.

[47] N.M. Peterson and W.S. Meyer. Automatic adjustment of transformer and phase-shifter taps in the Newton power flow. *IEEE Transactions on Power Apparatus and Systems*, 90:103–108, 1971.

[48] H.E. Pierce Jr, H.W. Colborn, D.W. Coleman, E.A. Marriage, J.C. Richard, L.J. Rindt, L.J. Rubino, G.W. Stagg, T.P. Traub, J. Vandergrift, et al. Common format for exchange of solved load flow data. *IEEE Transactions on Power Apparatus and Systems*, 92(6):1916–1925, 1973.

[49] H.A. Rahman, M. Armstrong, D. Mao, and J.R. Martí. I2Sim: a matrix-partition based framework for critical infrastructure interdependencies simulation. In *8th IEEE Electrical Power & Energy Conference*, pages 6–7, 2008.

[50] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial Mathematics, 2003.

[51] B. Stott. Review of load-flow calculation methods. *Proceedings of the IEEE*, 62(7):916–929, 1974.

[52] B. Stott and O. Alsac. Fast decoupled load flow. *IEEE transactions on power apparatus and systems*, 93:859–869, 1974.

[53] W.F. Tinney and C.E. Hart. Power flow solution by Newton's method. *IEEE Transactions on Power Apparatus and Systems*, 86:1449–1460, 1967.

[54] W.F. Tinney and J.W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *proc. IEEE*, 55(11):1801–1809, 1967.

[55] M. Tomim. *Parallel Computation of Large Power System Networks Using the Multi-Area Thévenin Equivalents*. PhD thesis, The University of British Columbia, Vancouver, 2009.

[56] M.A. Tomim, J.R. Martí, and L. Wang. Parallel computation of large power system network solutions using the Multi-Area Thévenin Equivalents (MATE) algorithm. In *16th Power Systems Computation Conference, PSCC2008 Glasgow, Scotland*, 2008.

[57] F. Tu and A.J. Flueck. A message-passing distributed-memory Newton-GMRES parallel power flow algorithm. *Power Engineering Society Winter Meeting*, 1:211–216, 2002.

[58] P. Vachranukunkiet. *Power flow computation using field programmable gate arrays*. PhD thesis, Citeseer, 2007.

[59] A.M. Variz, V.M. da Costa, J.L.R. Pereira, and N. Martins. Improved representation of control adjustments into the Newton–Raphson power flow. *International Journal of Electrical Power and Energy Systems*, 25(7):501–513, 2003.

[60] V. Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Press, 2008.

[61] WA Wiggers, V. Bakker, ABJ Kokkeler, and GJM Smit. Implementing the conjugate gradient algorithm on multi-core systems. 2007.

[62] A.I. Zečević. *New Decomposition Methods for Parallel Computation of Large Systems*. PhD thesis, Santa Clara University, 1993.

[63] A.I. Zečević and D.D. Šiljak. A Nested Decomposition Algorithm For Parallel Computation of very Large Sparse Systems. *MPE*, 1:41–57, 1994.

[64] A.I. Zečević and D.D. Šiljak. Balanced decompositions of sparse systems for multilevel parallel processing. *IEEE Trans. Circuits Syst. I*, 41:220–233, 1994.

[65] F. Zhang. *The Schur complement and its applications*. Springer Verlag, 2005.

Appendix A

Power Flow Convergence Characteristics

The convergence details of the 118 and 300 bus systems are provided here using the conventional PF program describe in section 3.1..

A.1 118 Bus System

Mismatch Tolerance = 1e-3

Iteration : 1
number of unsolved nodes = 117
number of solved nodes = 0
percentage of solved nodes = 0%
system error = -15.9
average nodal error = -0.0878
convergence state = 35.2%
Iteration : 2
number of unsolved nodes = 108
number of solved nodes = 9
percentage of solved nodes = 7.69%
system error = -4.7
average nodal error = -0.026
convergence state = 52.9%
Iteration : 3
number of unsolved nodes = 12
number of solved nodes = 105
percentage of solved nodes = 89.7%
system error = -0.0573
average nodal error = -0.000317
convergence state = 117%
Iteration : 4

number of unsolved nodes = 0
number of solved nodes = 117
percentage of solved nodes = 100%
system error = -1.06e-05
average nodal error = -5.84e-08
convergence state = 241%

A.2 300 Bus System

Mismatch Tolerance = 1e-3
Iteration : 1
number of unsolved nodes = 294
number of solved nodes = 5
percentage of solved nodes = 1.67%
system error = 127
average nodal error = 0.239
convergence state = 20.7%
Iteration : 2
number of unsolved nodes = 281
number of solved nodes = 18
percentage of solved nodes = 6.02%
system error = -51.4
average nodal error = -0.097
convergence state = 33.8%
Iteration : 3
number of unsolved nodes = 164
number of solved nodes = 135
percentage of solved nodes = 45.2%
system error = -2.92
average nodal error = -0.00551
convergence state = 75.3%
Iteration : 4
number of unsolved nodes = 6
number of solved nodes = 293
percentage of solved nodes = 98%
system error = -0.0321
average nodal error = -6.05e-05
convergence state = 141%
Iteration : 5

number of unsolved nodes = 0
number of solved nodes = 299
percentage of solved nodes = 100%
system error = -1.44e-05
average nodal error = -2.72e-08
convergence state = 252%

Appendix B

Power Flow Profiling

Output of profiling program is provided for 118 and 300 bus systems using the conventional PF program describe in section 3.1.

B.1 118 Bus System

Total time to run iteration #1 = 0.414603ms

Timer 1: RHS
75000 times = 1.78s
time to complete once = 0.0237333ms
percentage of total iteration time = 5.72435%
Timer 2: Jacobian
75000 times = 1.8s
time to complete once = 0.024ms
percentage of total iteration time = 5.78867%
Timer 3: SuperLU
7500 times = 2.7s
time to complete once = 0.36ms
percentage of total iteration time = 86.83%
Timer 4: Adjust Voltages
1.5e+06 times = 2.28s
time to complete once = 0.00152ms
percentage of total iteration time = 0.366615%
Timer 5: Sine & Cosine
400000 times = 2.14s
time to complete once = 0.00535ms
percentage of total iteration time = 1.29039%

Total time to run iteration #2 = 0.226078ms

Timer 1: RHS
75000 times = 1.78s
time to complete once = 0.0237333ms
percentage of total iteration time = 10.4978%
Timer 2: Jacobian
75000 times = 1.8s
time to complete once = 0.024ms
percentage of total iteration time = 10.6158%
Timer 3: SuperLU
7500 times = 1.28s
time to complete once = 0.170667ms
percentage of total iteration time = 75.4901%
Timer 4: Adjust Voltages
1.5e+06 times = 2.33s
time to complete once = 0.00155333ms
percentage of total iteration time = 0.687077%
Timer 5: Sine & Cosine
400000 times = 2.45s
time to complete once = 0.006125ms
percentage of total iteration time = 2.70924%

B.2 300 Bus System

Total time to run iteration #1 = 1.26143ms
Timer 1: RHS
75000 times = 4.23s
time to complete once = 0.0564ms
percentage of total iteration time = 4.47113%
Timer 2: Jacobian
75000 times = 6.06s
time to complete once = 0.0808ms
percentage of total iteration time = 6.40545%
Timer 3: SuperLU
7500 times = 8.3s
time to complete once = 1.10667ms
percentage of total iteration time = 87.7315%
Timer 4: Adjust Voltages
1.5e+06 times = 5s
time to complete once = 0.00333333ms

percentage of total iteration time = 0.264251%
Timer 5: Sine & Cosine
400000 times = 5.69s
time to complete once = 0.014225ms
percentage of total iteration time = 1.12769%
———————————————————————————-
Total time to run iteration #2 = 0.781028ms
Timer 1: RHS
75000 times = 4.26s
time to complete once = 0.0568ms
percentage of total iteration time = 7.27246%
Timer 2: Jacobian
75000 times = 6.06s
time to complete once = 0.0808ms
percentage of total iteration time = 10.3453%
Timer 3: SuperLU
7500 times = 4.69s
time to complete once = 0.625333ms
percentage of total iteration time = 80.0654%
Timer 4: Adjust Voltages
1.5e+06 times = 4.98s
time to complete once = 0.00332ms
percentage of total iteration time = 0.425081%
Timer 5: Sine & Cosine
400000 times = 5.91s
time to complete once = 0.014775ms
percentage of total iteration time = 1.89174%

Appendix C

Small Matrix Multiply Code

Benchmarked Kernel Code:

```
#include <stdio.h>
#include <cuda.h>
#define GRID_SIZE 14
#define BLOCK_SIZE 192
#define NUM_ELEMENTS (3*3) * BLOCK_SIZE
#define TEST_ITERATIONS 1000000
__device__ void reg2smem( volatile float *, volatile float *, const int);
__device__ void smem2reg( volatile float *, volatile float *, volatile float *, volatile float *, const int);
__device__ void mad_ops( volatile float *, volatile float *, volatile float *);
__device__ void reg2gmem( volatile float *, volatile float *, const int);
// Main Kernel
__global__ void min_ops( float *gA, float *gB ) {
   int i;
   const int tx = threadIdx.x;
   volatile float a[9], b[9];
   volatile float c[9] = {0,0,0,0,0,0,0,0,0};
   volatile __shared__ float sA[NUM_ELEMENTS], sB[NUM_ELEMENTS];
   #pragma unroll 20
   for ( i = 0 ; i < TEST_ITERATIONS ; i ++ ) {
      smem2reg(a, b, sA, sB, tx);
      __syncthreads();
      mad_ops(a, b, c);
      reg2smem(c, sA, tx);
   }
   __syncthreads();
   reg2gmem(c, gA, tx);
```

```
}
// Auxiliary Functions
__device__ void smem2reg( volatile float *a, volatile float *b, volatile float *sA, volatile float *sB,
const int tx) {
    a[0] = sA[tx*9];       b[0] = sB[tx*9];
    a[1] = sA[tx*9+1];     b[1] = sB[tx*9+1];
    a[2] = sA[tx*9+2];     b[2] = sB[tx*9+2];
    a[3] = sA[tx*9+3];     b[3] = sB[tx*9+3];
    a[4] = sA[tx*9+4];     b[4] = sB[tx*9+4];
    a[5] = sA[tx*9+5];     b[5] = sB[tx*9+5];
    a[6] = sA[tx*9+6];     b[6] = sB[tx*9+6];
    a[7] = sA[tx*9+7];     b[7] = sB[tx*9+7];
    a[8] = sA[tx*9+8] ;    b[8] = sB[tx*9+8];
}
__device__ void mad_ops( volatile float *a, volatile float *b, volatile float *c) {
    c[0] = a[0]*b[0] + c[0];    c[0] = a[3]*b[1] + c[0];    c[0] = a[6]*b[2] + c[0];
    c[1] = a[1]*b[0] + c[1];    c[1] = a[4]*b[1] + c[1];    c[1] = a[7]*b[2] + c[1];
    c[2] = a[2]*b[0] + c[2];    c[2] = a[5]*b[1] + c[2];    c[2] = a[8]*b[2] + c[2];
    c[3] = a[0]*b[3] + c[3];    c[3] = a[3]*b[4] + c[3];    c[3] = a[6]*b[5] + c[3];
    c[4] = a[1]*b[3] + c[4];    c[4] = a[4]*b[4] + c[4];    c[4] = a[7]*b[5] + c[4];
    c[5] = a[2]*b[3] + c[5];    c[5] = a[5]*b[4] + c[5];    c[5] = a[8]*b[5] + c[5];
    c[6] = a[0]*b[6] + c[6];    c[6] = a[3]*b[7] + c[6];    c[6] = a[6]*b[8] + c[6];
    c[7] = a[1]*b[6] + c[7];    c[7] = a[4]*b[7] + c[7];    c[7] = a[7]*b[8] + c[7];
    c[8] = a[2]*b[6] + c[8];    c[8] = a[5]*b[7] + c[8];    c[8] = a[8]*b[8] + c[8];
}
__device__ void reg2smem( volatile float *c, volatile float *sA, const int tx) {
    sA[tx*9] = c[0];
    sA[tx*9+1] = c[1];
    sA[tx*9+2] = c[2];
    sA[tx*9+3] = c[3];
    sA[tx*9+4] = c[4];
    sA[tx*9+5] = c[5];
    sA[tx*9+6] = c[6];
    sA[tx*9+7] = c[7];
    sA[tx*9+8] = c[8];
}
__device__ void reg2gmem( volatile float *c, volatile float *gA, const int tx) {
    gA[tx*9] = c[0];
    gA[tx*9+1] = c[1];
```

```
    gA[tx*9+2] = c[2];
    gA[tx*9+3] = c[3];
    gA[tx*9+4] = c[4];
    gA[tx*9+5] = c[5];
    gA[tx*9+6] = c[6];
    gA[tx*9+7] = c[7];
    gA[tx*9+8] = c[8];
}
```

Appendix D

Small Matrix Multiply Assembly Code

The following is a partial view of the assembly code used in small matrix multiplication.

```
// addressing instruction
movsh.b32 $ofs2, $r11, 0x00000000

// move A & B (18 move instructions)
mov.half.b32 $r27, s[$ofs1+0x0000]
mov.half.b32 $r26, s[$ofs1+0x0004]
mov.half.b32 $r23, s[$ofs1+0x0008]
...
mov.half.b32 $r6, s[$ofs2+0x0038]
mov.half.b32 $r2, s[$ofs2+0x003c]
mov.half.b32 $r0, s[$ofs2+0x0040]

// synchronization instruction
bar.sync.u32 0x00000000

// 27 MAD instructions with 9 MOV instructions (for matrix C)
mad.rn.f32 $r20, $r28, $r22, $r20
mad.rn.f32 $r25, $r28, $r8, $r25
mad.rn.f32 $r17, $r27, $r28, $r17
mad.rn.f32 $r16, $r13, $r27, $r16
mad.rn.f32 $r16, $r26, $r9, $r16
// total of 56 instructions
```