INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

LPGAS - Large Power Grid Analysis and Simulation

Fábio André dos Santos Barata

Master Thesis

Electrical and Computer Engineering

Juri

Presidente: Nuno Cavaco Gomes Horta

Orientador: Luis Miguel Teixeira D'Ávila Pinto da Silveira

Vogal: Jose Carlos Campos Costa

December 2011

I. Abstract

Power grid analysis is becoming a very challenging circuit design problem which defies the limits of computational resources – processing and memory. Current very large scale integration (VLSI) designs have already exceeded one hundred million nodes and any power grid analysis and verification problem tends to get enormous. Computer resources available are already too short to store and process the many gigabytes (GB) of data involved in the analysis. For that reason, very efficient parallelizable strategies, compatible with distributed data, are absolutely needed for this problem.

Electromigration analysis is one of the most common reliability problems in the VLSI circuitry. Power grids supply current to integrated circuits and their metallic connections suffer while the current travels through them. This situation has worsened for current technology nodes given the diminutive dimensions used for those connections. To study the electromigration problem, the current in the power grid connections must be computed. Power grid analysis consists in computing the voltage values in power grid circuit nodes, which can be used to obtain the connection currents and solve the electromigration problem.

In this study, general-purpose and power grid specialized solvers are examined, developed and applied to both realistic benchmark circuits and artificially generated power grids. It will be seen that some of the solution strategies fail to achieve a good acceptance in some of the most important characteristics, such as parallelism and data distribution flexibility, low communication rates and result scalability (in terms of solution error over performance), while others succeed for the time being (since these objectives always have margin for improvement). Furthermore, a parallel solution requires partitioning of power grids. As a power grid can easily be interpreted as a graph, dividing it can be accomplished through graph partitioning techniques. In this work, the most popular techniques will be applied, some relying on access to geometrical information and others that do not require it and must be used when such information is not available.

Block-Jacobi Preconditioned Conjugate Gradient, which is a popular solver yet to be applied to the power grid problem, seems to show the best results among the analysed strategies, topping about 5 hours analysing a 7.9 million node power grid with 300 current sources on a 10 octocore cluster. While much seems to be still required in terms of improving computational requirements to be able to solve such a grid in reasonable time, the projected memory footprint seems in line with what is available even today, which is a good sign.

Keywords: VLSI, Electromigration, Power grid analysis, Partitioning, System of linear equations, Parallel and distributed computing

I. Resumo

Análise de power grids tem vindo a emergir ao longo do tempo como um problema de circuitos que atinge os limites dos recursos computacionais, processamento e memória. Circuitos integrados de grande escala (VLSI) já excedem cem milhões de nós e qualquer análise e verificação de power grids tende a ficar enorme. Os recursos disponíveis são já muito curtos para guardar os muitos gigabytes de dados envolvidos no processo. Então, estratégias paralelizáveis e eficientes, compatíveis com distribuição de dados, são necessárias para atacar este problema.

O teste de Electromigração é um dos problemas de fiabilidade mais comuns nos circuitos VLSI. As power grids fornecem corrente ao circuito integrado e as suas ligações degradam-se à medida que a corrente passa. Este problema agravou-se com o avanço tecnológico dados os minúsculos comprimentos utilizados nas ligações. Para estudar o problema da Electromigração, a corrente que passa em cada uma das ligações da power grid deve ser determinada. A análise de power grids consiste em determinar os valores de tensão dos nós da power grid, o que pode ser utilizado para calcular as correntes nas ligações e resolver o problema da Electromigração.

Neste estudo, soluções de análise de power grids serão examinadas, desenvolvidas e aplicadas tanto a circuitos benchmark como a power grids criadas artificialmente. Algumas soluções falham em atingir bons resultados em algumas características importantes como paralelismo, flexibilidade na distribuição de dados, baixas taxas de comunicação e escalabilidade no resultado (no sentido de margem de erro face a performance), enquanto que outras atingem até ao momento (visto que tais objetivos têm sempre margem para melhoria). Porém, uma solução paralela requer particionamento das power grids. Dado que uma power grid pode ser vista como um grafo, a sua divisão pode ser obtida através de técnicas de particionamento de grafos. Neste trabalho, as técnicas mais populares serão aplicadas, sendo algumas dependentes do acesso a informação geométrica e outras que não necessitam e deverão ser utilizadas quando esse acesso não está presente.

O algoritmo de Gradiente Conjugadocom pré-condicionamento de Bloco-Jacobi, uma solução nunca antes aplicada ao problema das power grids, mostra os melhores resultados entre as várias estratégias analisadas, demorando 5 horas a analisar uma power grid de 7.9 milhões de nós com 300 fontes de corrente num aglomerado de 10 octocores. Mesmo sabendo que muito terá de ser melhorado para conseguir resolver uma grelha real em tempo razoável, a memória projetada está de acordo com o que existe hoje mesmo, o que é um bom sinal.

Palavras-chave: VLSI; Electromigração; Análise de power grids; Particionamento; Sistemas de equações lineares; Computação paralela e distribuida

II. Acknowledgements

Thanks to Professor **Luis Miguel Teixeira D'Avila Pinto da Silveira** for his highly supporting guidance throughout the whole realization of all the work related to this thesis.

Thanks to **IBM** and Dr. **Sani Nassif** for providing motivation for this work and several interesting and insightful conversations, as well as for sharing their current strategy to approach the problem addressed in this study.

Table of Contents

Figures

Index of Tables

Acronyms and Symbols

B.....Current source incidence matrix (NxM)
BI.....Block-Iterative
BJ.....Block-Jacobi
C4.....Flip-chip or Controlled Collapse Chip Connection
DD.....Domain Decomposition
DP.....Domain Partitioning
EM.....Electromigration
G.....KCL Conductance Matrix (NxN)
GB.....Gigabytes
G_{DD}.....Voltage source chatacteristic conductance
GJ.....Gauss-Jordan
GP.....Geometric Partitioning
i.....Current source amplitude vector
IC.....Integrated circuit
IChol.....Incomplete Cholesky factorization
ILU.....Incomplete LU factorization
LP.....Linear Programming
LPSA.....Locality-Driven Parallel State Analysis
K.....Number of voltage sources in one power grid
m.....Number of partitions
M.....Number of current sources in one power grid
MOR.....Model Order Reduction
N.....Number of nodes in one power grid
NZ.....Number of non-zeros in KCL conductance matrix, typically O(N)
PCG.....Preconditioned Conjugate Gradient
PDN.....Power Distribution Network or Power grid
QP.....Quadratic Programming
SOR.....Successive Over-Relaxation
SSOR.....Symmetric Successive Over-Relaxation
Δv .....Voltage drop from reference (V_{DD})
V_{DD}.....Nominal voltage fixed by voltage source
VLSI.....Very large-scale integration
z.....Number of iterations

1 Introduction

Power grid analysis is becoming more and more a critical task to ensure the proper functioning of integrated circuits. As the number of transistors in circuits keeps increasing, the density of power distribution networks (PDN or power grids) increases manifold. As a result, analysis of current power grids in VLSI designs threatens to exceed the formidable computational power available and has developed into a considerable problem. In reality, distributed systems and modern technology take an important role in solving the problem, because one computing instance is not enough to support the computational and memory burden.

1.1 Electromigration

This research focuses in solving the EM problem using Power grid analysis. EM is mass-electron movement in metallic interconnects. This phenomena can cause anomalies such as connection disruptions (caused by void failures and diffusive displacements) and component failure due to heating, eventually diseasing the IC. Measurement or estimation of the current density in interconnects is crucial to study the EM (an empirical model developped by J. R. Black in 1960 estimates the mean time to failure from the current density) and envision the IC lifespan. Even though it is a time-dependent case, the interest of this study is the EM DC problem. One way to address this problem is to model the power using an electric circuit equivalent where power grid connections are replaced with an appropriate electrical model. Then, in order to determine the currents in the power grid and to determine if they exceed some safety values and could lead to EM problems, one can analyse the power grid, solve for the nodes voltages in the model and compute the branch currents. Solving power grid analysis problems is therefore an essential key for EM analysis.

1.2 Power grids

Power grids are metallic grids that distribute the power supplied by a power supply and required by an integrated circuit (IC) functional blocks. The supply polarizes the grid with a fixed voltage V_{DD} and functional blocks depend on a steady voltage value for a proper functioning, which would ideally be equal to V_{DD}. However, due to the metallic conductor properties and current presence, the voltage of nodes distant from the contact points (where the power is supplied) may drop somewhat. The voltage source attached to the power grid must ensure the circuit proper functioning, given it is lower on the functional block terminals.

Typically, a power grid can be modelled by an RC circuit mesh (Resistors model metallic

connections between two nodes; Capacitances model variations in loading conditions and electromagnetic field induced currents) with current sources (modelling the functional blocks) and voltage sources (or bias contacts, modelling the connections to the power grid to a source establishing its overall voltage and providing current for the underlying electronics to function), as shown in Figure 1. Power grid analysis consists in determining the values of voltage and currents in the grid nodes and branches (connections), which implies simulating the system for time-constant (DC) and transient responses.

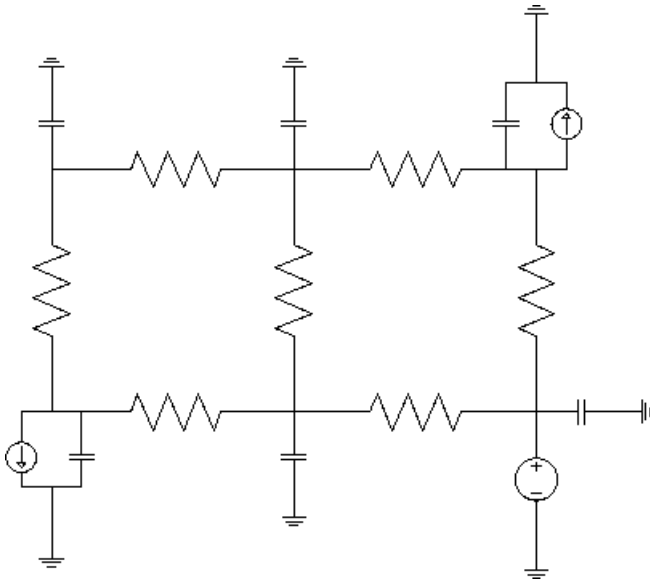

Figure 1: RC circuit mesh example with 6 nodes.

Last but not least, current sources are not constant. The functional blocks demand current when there exists a state transition on a logical component (or port), which varies with the transition type (0 to 1 or 1 to 0). Also, not all components change at the same time, it depends on an ongoing state and a new input. This will require the power grid analysis to be able to provide the branch currents given any possible state transition.

Knowledge of nominal value of all logic block's current demands is context dependent, meaning it depends on the state of the block, the sequential elements, inputs, etc. Such information is not easy to determine as it would require full-chip simulation of the underlying circuit, a very expensive endeavour. For power grid analysis, it is sufficient to treat the circuit elements at the block level and to model their current requirements in a global sense. In other words, it is common for power grid analysis that a single current source represents the current requirements of a whole cell of block. The number of current sources to consider is therefore of the order of the number of blocks, much smaller than the granularity of the grid contacts. A relevant analysis is the determination of the node voltage (and branch currents) variation, then the block currents vary. Such analysis allows us to account for the

fact that within a given cell or block one might experience small peaks of increased activity. Being able to perform such an analysis as a simple variation on the power grid inputs is an interesting capability that can be of much use to the designer.

1.3 Limitations of the analysis

Many manufacturers, like IBM, already routinely design power grids have reached 300 million nodes with 2000 current sources. They expect to be nearing designs with close to a billion nodes in the power grid model. A single common computing instance is hardly enough to contain the power grid information (300M nodes would use about 23GB just to store the power grid connection data), let alone memory overheads during the analysis (in fact, IBM routinely builds special purpose computers with vast amounts of memory in order to be able to handle such large power grid designs). As the power grids dimensions are extremely large, a distribution of processing and memory utilization (parallelism) through various computing instances is necessary. For this purpose, a set of tools implementing partitioning techniques is used to assign (distribute) the power grid nodes to computing instances. The efficiency of the analysis is strongly affected by the communication needed in the process, which is usually determined by the number of interconnections between the partitions (in other words, by the number of branches connecting a partition to other partitions). On the other hand, partitions must also be such that the load on various computing instances is balanced.

1.4 Objectives

Given the exigencies of the power grid analysis problem, this work is focused on setting up the current state in this domain, as well as identifying the drawbacks of the current strategies and proposing novel solutions.

1.5 Contribution of this work

The power grid DC analysis is abstracted to a system of linear equations with a graph geometrical interpretation, thus linear system solvers are applicable. In this research, a selection of existing methods to approach the power grid DC analysis through both a graph and a linear system problem is studied. Given the linear nature of the problem, this work presents a comparison and development of the existing power grid analysis solving methods, such as Block Iterative (BI) and Locality-Driven Parallel State Analysis (LPSA), as well as other popular linear solvers, like Cholesky factorization, Domain Decomposition (DD) and Preconditioned Conjugate Gradient (PCG) with Block-Jacobi Preconditioning (which is yet to be applied to the power grid analysis). Since the distributed data parallel analysis is absolutely necessary, this study will focus on these terms. In addition, partitioning

methods like Domain Partitioning (DP), Geometric partitioning (GP) and Ratio cut (an algebraic method which is also going to be applied for the first time to the power grids domain) are also studied.

1.6 Thesis structure

In Chapter 2, the power grid model and the theoretical background behind power grid analysis is explained. In Chapter 3 a first attempt to reduce the problem size is made using MOR reduction techniques. In Chapter 4, power grid analysis direct solvers like Cholesky factorization and DD and other iterative solutions: BI, LPSA and PCG are discussed and improved targeting a parallel solving using distributed data. In Chapter 5, widely known partitioning techniques and their advantages and disadvantages are shown. In Chapter 6, some implementation details are revealed. Results of the algorithms presented in Chapters 4 and 5 are illustrated in Chapter 7. Chapter 8 concludes the thesis.

2 Background

2.1 Power grid model

A power grid can be modelled by an RC mesh with current and voltage sources. Resistances model the metallic connections between two nodes and capacitances model electrostatic effects such as variations in loading conditions and electromagnetic field induced currents. Coil models are not required for frequencies in the order of GHz. The most commonly used model is a graph representation of these components. As the focused problem in this study is DC analysis, the capacitances are omitted. An example of the model of the power grid in Figure 1 is illustrated in Figure 2:

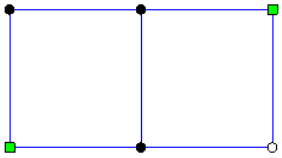

Figure 2: Power grid representation example. Current sources are connected to nodes represented in green squares. Voltage sources are connected to nodes in white circles.

In real power grid circuits, there is only one voltage source connected to multiple nodes. In this power grid model, that will be made into multiple sources with the same amplitude and characteristic conductance connecting the same nodes.

2.2 Power grid analysis

As stated in the introduction, to study potential EM problems, the branch currents must be determined. But, for that, the voltage difference in each branch suffices. A way of solving this problem is to determine the voltage of each of the N nodes in a power grid[1], given M current sources and K voltage sources (which are replaced by Norton equivalent current sources, using their characteristic conductance G_{DD}). This is called the power grid DC analysis and is obtained by solving for v in the following linear system:

$$G\,v = B\,i + B_v\,i_v \tag{1}$$

where:

G – a NxN matrix with NZ non-zero entries, imposed by the KCL rule for each grid node. This is

called the KCL conductance matrix

B – a NxM incidence matrix of the current sources (ie $B_{j,k}$ = 1 iif current source k is connected to node j)

i – a Mx1 vector with current source amplitudes

B_v – a NxK incidence matrix of the Norton equivalent current sources corresponding to the voltage sources

i_v – a Kx1 vector with Norton equivalent current sources amplitudes corresponding to the voltage sources

Note that, in the case of multiple current sources connected to the same node, they are beforehand merged into a single source. Consequently, the B matrix is composed by M specific identity matrix columns.

The equation (1) can be split into two components, leading to:

$$G v = B i + B_v i_v \Leftrightarrow G\left(v_0 + \Delta v\right) = B_v i_v + B i \tag{2}$$

where v_0 is the solution to the biasing voltage sources and Δv is the voltage drop caused by the current requirements from the logic blocks. Equation (2) can be split by superposition into:

$$G v_0 = B_v i_v \tag{3}$$
$$G \Delta v = B i \tag{4}$$

The solution to equation (3) is easily obtained by inspection if it is noted that the right-hand side corresponds to the biasing voltage sources. The node voltage at every grid node is therefore equal to V_{DD}. For equation (4), the voltage sources are grounded and the voltage variations is due only to the currents in the logic blocks. The power grid analysis problem is therefore described by equation (4), which will be used in the rest of this work to determined the voltage drop Δv.

An example of a 12 node power grid is illustrated in Figure 3. An example of system (4) (with solution) for this power grid is:

$$G = \begin{bmatrix}
12.5 & -10.0 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-10.0 & 11.0 & -0.5 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -0.5 & 11.0 & -10.0 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & -10.0 & 10.5 & 0.0 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 & 0.0 & 0.0 \\
-0.5 & 0.0 & 0.0 & 0.0 & 11.0 & -10.0 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 & 0.0 \\
0.0 & -0.5 & 0.0 & 0.0 & -10.0 & 11.5 & -0.5 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 \\
0.0 & 0.0 & -0.5 & 0.0 & 0.0 & -0.5 & 11.5 & -10.0 & 0.0 & 0.0 & -0.5 & 0.0 \\
0.0 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 & -10.0 & 11.0 & 0.0 & 0.0 & 0.0 & -0.5 \\
0.0 & 0.0 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 & 0.0 & 10.5 & -10.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 & -10.0 & 11.0 & -0.5 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 & -0.5 & 11.0 & -10.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 & -10.0 & 10.5
\end{bmatrix}$$

$$G \begin{bmatrix} 0.050 \\ 0.057 \\ 0.144 \\ 0.146 \\ 0.109 \\ 0.111 \\ 0.185 \\ 0.192 \\ 0.129 \\ 0.130 \\ 0.169 \\ 0.170 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 1.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} [0.1]$$

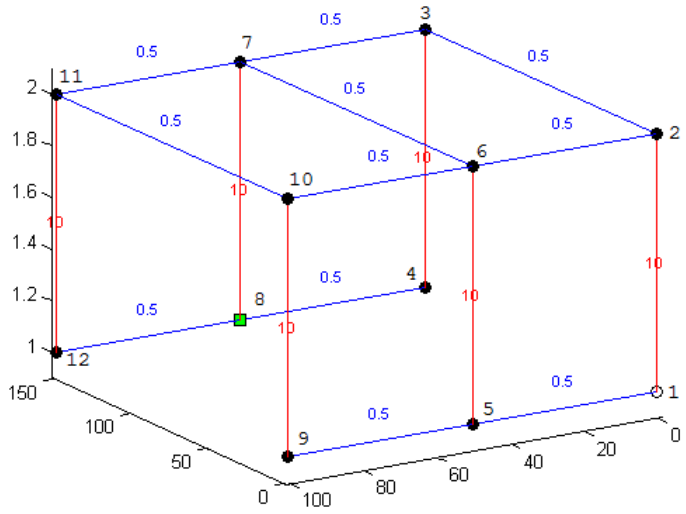

Figure 3: Power grid example with 2 layers. Conductances are marked in branches.

Typical power grids have single-directed connections only in each layer (meaning a node only connects two others on the same layer, and they form a line). In this pattern, a node only connects 4 other nodes maximum (one in the next upper layer, one in the next lower layer and two in their layer). Also, upper layers are less dense than lower layers, which means not all nodes connect other layers. In an IBM benchmark power grid, the mean connections per node is 3.25.

It is well known that KCL conductance matrices for the power grid DC analysis have the following properties:

1) Symmetry;
2) Positive Definite and Diagonally dominant;
3) NZ = c*N, with c equal to number of connections per node plus 1, typically 4.25 in the IBM benchmark power grids (in other words, G is very sparse, as shown in Figure 4).

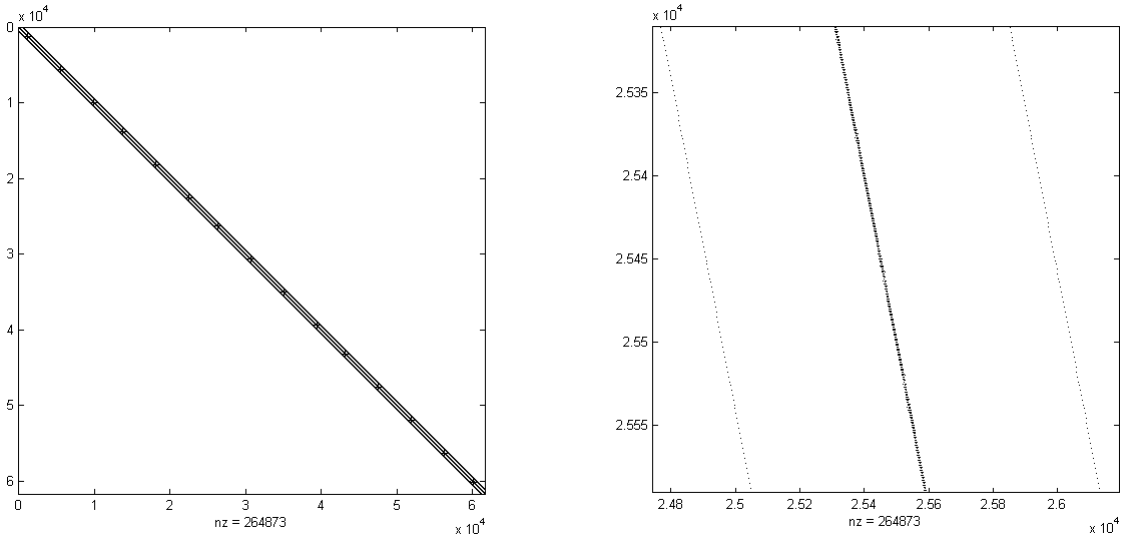

Figure 4: G matrix non-zero pattern of a 62k node IBM benchmark power grid. Right figure is a zoom of the full pattern on the left

2.3 Problem formulation

As stated in Chapter 1, current sources model the functional blocks of the IC. While logical elements of a functional block demand variable amounts of current depending on state transitions, the right-hand side of equation (4) is also variable. Calculating all possible state transitions is not doable, so the linear coefficients associated with each current source must be calculated. For this, two approaches can be used: Monte Carlo estimation (which is an approximation and typically requires a high amount of samples to be reliable) or the following equation (called full right-hand side):

$$s = G^{-1} B \Leftrightarrow v = s\, i \tag{5}$$

Instead of computing the full inverse G^{-1}, one can solve columns of B independently for a more efficient solution, avoiding the computation of the full inverse (since only M lines are needed). This is equivalent to the application of the superposition theorem. Either way, one needs to compute a high amount of B columns (typically 2000 for current power grids of 300 million nodes). As the Monte Carlo estimation requires an amount of samples comparable to the full right-hand side, it does not justify the use of a random process so early in the analysis.

2.4 Considerations

Currently designed power grids have a huge number of nodes. This suggests that the main difficulty to the power grid problem is a size issue. The amount of resources required are too much for a single computer to provide. For this reason, the following considerations should be contemplated a priori when approaching the power grid problem:

1) For realistic power grids in dense technology nodes, storage of the KCL conductance matrix is usually impossible on a singular computer.

2) Distribution of processing and data through various computing instances is required. However, the communication between instances will hamper the performance, so it should be decentralized and kept at the minimum possible. In addition, distribution should only depend on the number of available processing instances and not on external variables.

3) Reduction of data. Memory spent should be reduced substantially without conceding much performance.

4) Scalable solution. A user-provided maximum acceptable error for the solution will determine the time and/or memory spent in the whole process.

5) Structure. All structural information of the power grid must be taken advantage of. An example is locality: Currently used flip-chip (C4) technology provides a high density of voltage contacts (also called V_{DD} bumps), which implies that a current source effect in voltage drops decreases over distance (Figure 5).

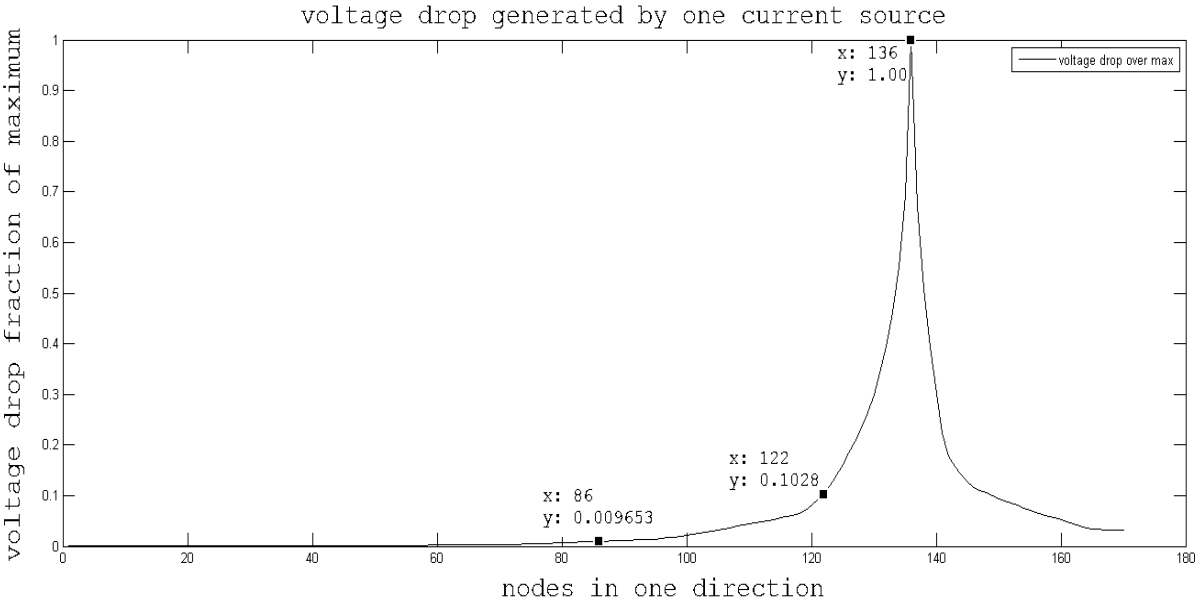

Figure 5: Voltage drop along one dimension after simulating a power grid with multiple voltage sources and one current source. After less than 10% nodes, the drop is less than 10% of the maximum.

3 Model Order Reduction

Model Order Reduction (MOR) techniques address size problems, by reduction of the number of nodes, producing an equivalent model without degrading the information needed in the analysis. As the problem at hand is very much about the size of the power grids, some compression would be very useful to tackle the problem more efficiently.

3.1 Node elimination

Node elimination is a MOR technique based on Gaussian elimination. The main idea of node elimination is to find an equivalent circuit (in terms of the voltage drop of the remaining nodes) with a reduced number of nodes. A widely known example of node elimination is the star-mesh transformation, which is a particular case of the single node elimination. The resulting circuit of a general single node elimination is given by linking all adjacencies (Figure 6).

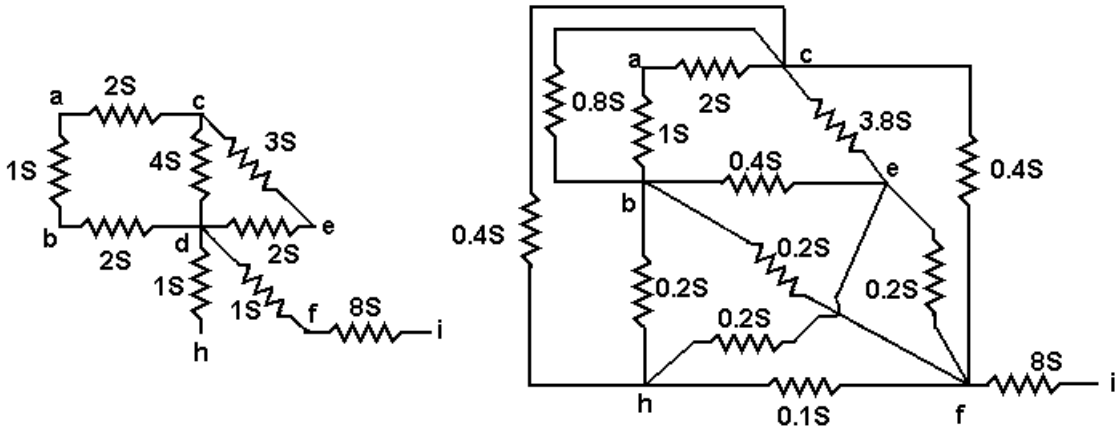

Figure 6: Node elimination example. The circuit on the right, after the elimination of node d, is equivalent to the one on the left. In this example, it is seen that nodes c and e were connected before the elimination of node d, which results in a parallel resistor that was simplified, but the overall number of resistors grow

Single node elimination affects adjacencies of the eliminated node only. The resulting interconnecting conductances are calculated using:

$$g'_{jk} = \frac{g_j g_k}{\sum_i g_i} \tag{6}$$

where:

g'_{jk} – additional conductance between nodes j and k, after the elimination

g_i – conductance between node to eliminate and adjacency i, before the elimination

The multiple node elimination is given by the Schur complement algorithm for equation solving:

Algorithm 3.1: Node elimination

Inputs: G, elim = nodes to eliminate, **Outputs:** Gr = Reduced G

1) keep = complementary set of elim

2) **Solve for** Q **in** G[elim;elim]*Q = G[elim;keep] ← submatrices of G, G[rows;columns]

3) $G_r = G[keep;keep]-Q^TQ$

Nodes with sources of any type cannot be eliminated directly without changing the right-hand side of the system (1). In [2], a method for source and capacitance expansion was proposed, which allowed the elimination of these nodes. In our case, this is a minor advantage as the number of sources is small compared to the total number of nodes.

This process is equivalent to iterating in a Cholesky factorization process. This and other MOR techniques effectively reduces the size of the problem, but the density of the reduced Gr matrix is generally much higher than the original if many nodes are eliminated (Figure 7), which is undesired.

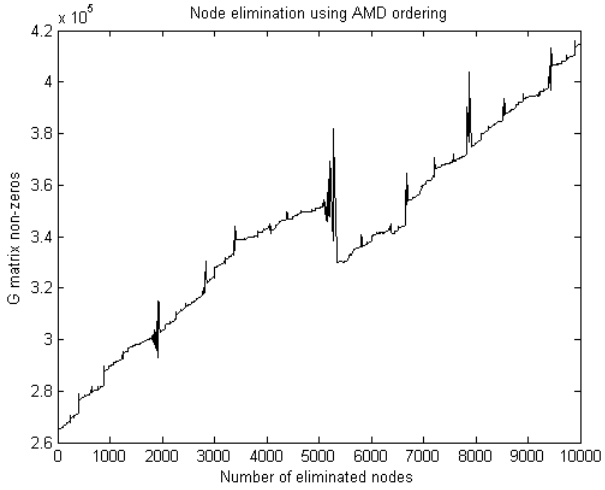

Figure 7: Number of non-zeros of G after the elimination of up to 10k nodes in a 62k node IBM benchmark power grid circuit ordered by AMD

Taking into account the fact that one can eliminate multiple nodes independently in any order, when a node with more than three adjacencies is eliminated, the total number of branches usually increases, and so does the number of non-zeros of G. As for nodes with less than three adjacencies, they are profitably eliminated one by one.

Figure 8 shows the best case scenario for the node elimination for a particular power grid. It can be seen that even the best case reduction for non-zeros is unsatisfactory.

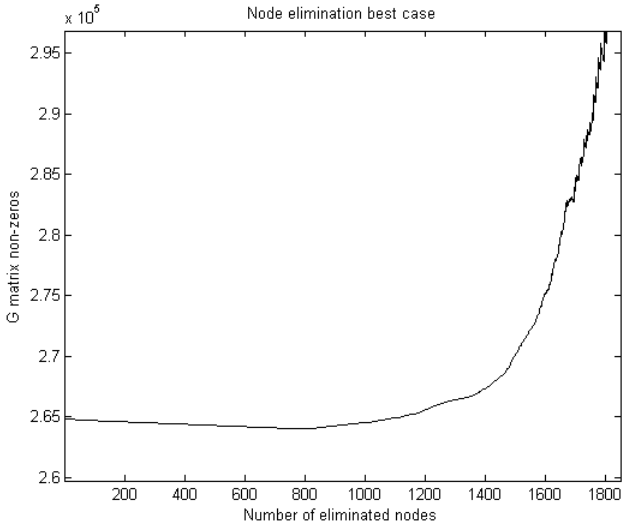

Figure 8: Number of non-zeros of G after the successive single elimination of the most profitable nodes in a 62k node IBM benchmark

3.2 Other MOR methods

MOR methods have a large number of applications in the electronics VLSI area, as they successfully compress the circuit to a smaller number of nodes. Very popular methods used in this context are PRIMA[3], SVDMOR[4], DeMOR[5], RecMOR[6], BSMOR[7] and PMTBR[8]. These methods make use of a projection matrix to transform the original model, but this projection scheme will generate a dense[9] (Figure 9) reduced matrices (in other words, they increase the number of nonzeros of the KCL conductance matrix) and thus is not very useful. In addition, the larger the number of inputs (voltage and current source nodes), the wider the projection matrix will be which leads to larger reduced models.

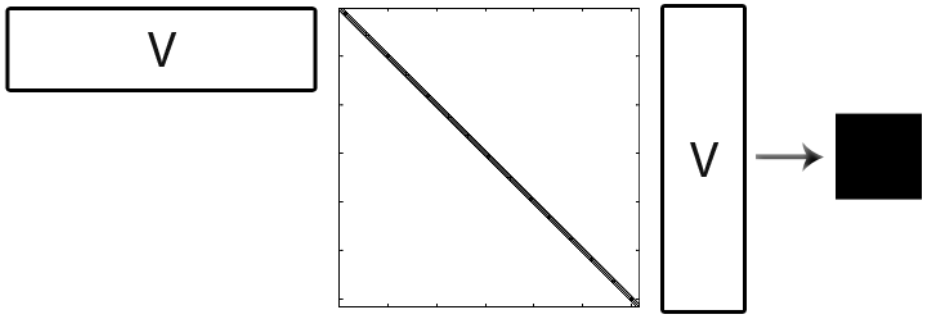

Figure 9: Fill-in caused by matrix projection V in a typical MOR algorithm

Overall, when applied to power grid systems which are very sparse but have a considerable number of inputs (sources), MOR techniques lead to disappointing results[9] and are not considered very appropriate.

4 System solution

As a result of the extremely large currently used power grids (some tens of million nodes), the computational resources needed to analyse them are enormous which quickly precludes the usage of methods difficult to parallelize and distribute the data used. In recent years, several strategies emerged to overcome these problems and tackle the power grid analysis in a more efficient way. Recognizing the computational burden of the problem leading to excessive processing times and memory overflows, these recent strategies have opted to pursue approaches where parallelism is exploited in some way in order to substantially reduce the memory used by each computing instance. In this section, some of the strategies that fall along those lines are presented, and they often need some preprocessing that employ a division of the grid, which is a subject examined in Chapter 5 .

4.1 Direct methods

4.1.1 Cholesky factorization

The most basic direct method is triangular solving (forward and backward) of the columns of B, using the Cholesky factorization of G or reordered G (usually by a minimum degree ordering).

Algorithm 4.1: Cholesky factorization method

Inputs: G, B, **Outputs:** Solution

1) **Find** a minimum degree ordering of G
2) **Apply** ordering in G rows and columns
3) **Apply** ordering in B rows
4) **Compute** L = Cholesky factor of G
5) **Solve for** u **in** Lu = B
6) **Solve for** s **in** $L^T s = u$
7) **Apply** reverse ordering in s rows

In Table 1, the time and memory complexity of the direct method is shown. The constants α, β and γ vary in the interval [1,2] and are strongly driven by the ordering of G (which is chosen), because of the fill-in caused during the factorization process. For AMD[10] (approximate minimum degree) ordering, a typical value for these constants is 1.3. The constant γ is always at least equal to β.

Step	Complexity	
	Time	**Memory**
Store G	-	$O(N)$
Factorize G	$O(N^{\alpha})$	$O(N^{\gamma})$
Solve	$O(N^{\beta}M)$	$O(NM)$
Overall	$O(N^{\alpha}+N^{\beta}M)$	$O(N^{\gamma}+NM)$

Table 1: Time and memory complexity of the Direct Method

Employing the Cholesky factorization method is actually very efficient, but a non-distributed solver rapidly exceeds the memory resources, as shown by the results (using CHOLMOD library) in the following table:

Grid Nodes	Current Sources	Time	Memory peak
62k	90	1.91s	130MB
404k	120	14.4s	817MB
1.2M	180	1070s	5.59GB

Table 2: Time and Memory spent using CHOLMOD library[11] for various power grid sizes

Elimination trees[12] have proved to be successful in parallelizing the Cholesky factorization solving over the years, even though it is not perfect in the sense that it does not guarantee a full parallelization of the process. However, in distributed architectures, the communication during factorization process is very high, heavily compromising the efficiency of the process. For instance, an 1.5k node (very small) power grid would have to communicate 100k nodes worst case, when divided in 4 pieces. This precludes the application of the Cholesky factorization direct method.

4.1.2 Domain decomposition

The Domain decomposition (DD) method (often called Traditional method) was first introduced to the power grid problem by Quming et. al.[13]. The DD algorithm is based upon a divide-and-conquer technique to solve the linear system in an exact way, with resort to the Schur complement for equation solving.

The circuit nodes are divided into a given number of non-overlapped sub-domains (or partitions) and are classified in two different types (Figure 10):

1. interior nodes: nodes that belong to one partition only and do not connect to any nodes in other partitions;

2. interface nodes: nodes that connect multiple partitions.

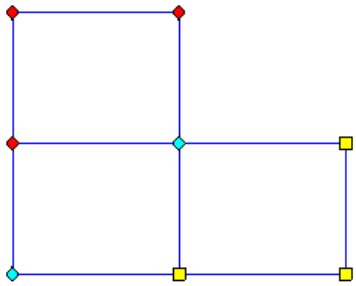

Figure 10: Interior and interface nodes example. Yellow squares and red diamonds represent two partitions interior nodes and cyan diamonds are interface nodes. Branches are conductances.

Given the circuit divided into m sub-domains (with the node classification), the linear system (1) is then rewritten as[13]:

$$\begin{pmatrix} A_D & E \\ E^T & A_\Gamma \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}=\begin{pmatrix} f \\ g \end{pmatrix}, \quad A_D=\begin{pmatrix} A_1 & 0 & 0 \\ 0 & \dots & 0 \\ 0 & 0 & A_m \end{pmatrix} \tag{7}$$

Matrices A_D, A_Γ and E are sub-matrices of G, corresponding to a different ordering such that: A_1 to A_m matrices are square and contain the connectivity information internal to each partition; E is rectangular and stores the connectivity information between interface and interior nodes; A_Γ is a square matrix holding the connectivity between interface type nodes only. It is easily seen that the vector (f,g), right-hand side of (7), is simply a reordered version of the original right-hand side of (4). The same is said about (x,y) in respect to the solution v. Taking for instance the example in Chapter 2 , dividing it in two parts (nodes 1 to 4 and 9 to 12, interface nodes 5 to 8), the reordered system will be:

$$\begin{bmatrix} \overbrace{\begin{matrix} 12.5 & -10 & 0 & 0 \\ -10 & 11 & -0.5 & 0 \\ 0 & -0.5 & 11 & -10 \\ 0 & 0 & -10 & 10.5 \end{matrix}}^{A_{D1}} & & & \overbrace{\begin{matrix} -0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & -0.5 & 0 \\ 0 & 0 & 0 & -0.5 \end{matrix}}^{E} \\ & \overbrace{\begin{matrix} 10.5 & -10 & 0 & 0 \\ -10 & 11 & -0.5 & 0 \\ 0 & -0.5 & 11 & -10 \\ 0 & 0 & -10 & 10.5 \end{matrix}}^{A_{D2}} & & \begin{matrix} -0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & -0.5 & 0 \\ 0 & 0 & 0 & -0.5 \end{matrix} \\ \underbrace{\begin{matrix} -0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & -0.5 & 0 \\ 0 & 0 & 0 & -0.5 \end{matrix}}_{E^T} & \begin{matrix} -0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & -0.5 & 0 \\ 0 & 0 & 0 & -0.5 \end{matrix} & \underbrace{\begin{matrix} 11 & -10 & 0 & 0 \\ -10 & 11.5 & -0.5 & 0 \\ 0 & -0.5 & 11.5 & -10 \\ 0 & 0 & -10 & 11 \end{matrix}}_{A_\Gamma} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.1 \end{bmatrix} \begin{matrix} \\ \\ \\ \\ f \\ \\ \\ \\ \\ \\ \\ g \end{matrix}$$

The definition of interior nodes suggests immediately that matrix A_D is a block diagonal matrix. This is where the parallelism is set in motion in this method. The DD solver algorithm (after preprocessing of partitioning and node identification) can be described in the following procedure[13]:

Algorithm 4.2: Domain decomposition

Inputs: A_D, A_Γ and E = sub-matrices of G, f = reordering of B*i, **Outputs:** Solution

1) **Solve** P **and** q **in** $A_D P = E$ **and** $A_D q = f$;
2) **Form** Schur complement matrix $S = A_\Gamma - E^T P$;
3) **Calculate** $g' = g - E^T q$;
4) **Solve** y **in** $Sy = g'$;
5) **Calculate** $x = q - Py$

The system is now solved in a more parallelizable way than the direct Cholesky method. The block diagonal structure of A_D allows the solve of step 1 to be performed in parallel in two ways: each block of A_D and each column vector of E and f. Likewise, the A_D matrix is also a sub-matrix of G, preserving its properties that are exploited for the solution process. This proves useful when it is well known that steps 1 and 4 are the most expensive steps of the process. Furthermore, basic matrix operations such as addition, subtraction and multiplication can also be parallelized (see [14]) and accelerated, therefore steps 2, 3 and 5 can also be parallelized. This algorithm therefore succeeds in distributing data in these steps and parallelizing the process.

The Schur complement matrix S can be seen as the result of node elimination of all interior nodes (see Section 3.1). For this reason, the S matrix is symmetric and positive definite but its sparsity tends to be very low and thus step 4 solving cost grows substantially with the number of interface nodes. This enforces the need for a good partitioning algorithm which keeps the number of interface nodes as low as possible (see Chapter 5). For the above example, the Schur complement matrix S results in (a completely dense matrix):

$$S = \begin{bmatrix} 10.7 & -10.3 & -0.0856 & -0.0816 \\ -10.3 & 11.2 & -0.595 & -0.0902 \\ -0.0856 & -0.595 & 11.1 & -10.4 \\ -0.0816 & -0.0902 & -10.4 & 10.6 \end{bmatrix}$$

As a result, step 4 does not profit from parallelism, but all computing instances can solve it so they have their data share for step 5.

In Table 3, computational complexity of the distributed Domain Decomposition method is shown,

given m even-sized partitions, N_{int} interface nodes and N_{in} interior nodes for each partition ($N_{in} = (N-N_{int})/m$).

Step	Complexity	
	Time	**Memory**
Store A_{Di}	-	$O(\frac{N}{m})$
Factorize A_{Di}	$O(N_{in}^{\alpha})$	$O(N_{in}^{\gamma})$
Solve Pi and qi	$O(N_{in}^{\beta}(N_{int}+M))$	$O(N_{in}(N_{int}+M))$
Form and assemble S	$O(N_{int}^{2})$	$O(N_{int}^{2})$
Form and assemble g'	$O(N_{int}M)$	$O(N_{int}M)$
Solve y	$O(N_{int}^{3}+N_{int}^{2}M)$	$O(N_{int}^{2}+N_{int}M)$
Calculate x	$O(N_{in}N_{int}M)$	$O(N_{in}M)$
Overall	$O(N_{in}^{\beta}(N_{int}+M)+N_{int}^{2}M+N_{int}^{3})$	$O(\frac{N}{m}+N_{in}^{\gamma}+N_{int}M+N_{in}(N_{int}+M)+N_{int}^{2})$

Table 3: Computational complexity for distributed Domain Decomposition Method for each computing instance

Even though computational complexity has more variables, time complexity is equivalent to the Cholesky factorization method, but does not take full advantage from the minimum degree ordering of the full matrix, but is a rather restricted one (each partition will have a minimum degree ordering independent from the others).

The Domain Decomposition method is the first main reference point to any power grid method that emerges. This is due to the great success of distributing some memory while maintaining a serial solution cost comparable to the Cholesky factorization method. However, due to the density of Schur complement matrix, it can turn into a very serious memory problem. If the number of interface nodes is large, the Schur matrix will contain a huge amount of nonzeros. For instance, if the number of interface nodes is 50k (which is not particularly high if the power grid has millions of nodes), the storage of a dense S requires approximately 20GB if 8 bytes per matrix entry are used (current double floating point precision) and it cannot be efficiently distributed.

4.2 Iterative methods

Given the large dimension of the power grids, the exact solution might still be very expensive to compute even when resorting to the DD technique. Inasmuch as a good approximation is enough to provide the information needed for the power grid analysis, iterative schemes emerged to favour the scalability and runtime efficiency.

4.2.1 Block-Iterative

The first algorithm, employing an iterative scheme, that will be studied here is the block-iterative approach[15] (BI). Again, the circuit is divided into various sub-domains. The nodes no longer have two different categories like in the DD method, but are all internal to one partition only and may connect other partitions (nodes that connect multiple partitions are called boundary nodes, but are still internal). The BI method is based on solving the system inside each partition using the neighbour solutions as current sources connected to the partition border nodes (nodes that connect to other partitions, a definition very similar to interface nodes), and thus modifying the right-hand side of (4).

The BI method borrows two different structures: Gauss-Jordan (GJ) and Successive over-relaxation (SOR). This governs the modification on the right-hand side of (4) previously enunciated. Given m partitions, the GJ modification takes the following pattern:

$$A_{ii}\, x_i^{(k+1)} = b_i - \sum_{\substack{j=1 \\ j \neq i}}^{m} A_{ij}\, x_j^{(k)} \tag{8}$$

where A_{ij} is a sub-matrix of G containing partition i lines and partition j columns (for example: if there are 3 partitions with nodes {1,2}, {3} and {4,5,6}, matrix A_{13} is a 2x3 matrix with the {1,2} lines and {4,5,6} columns of G); b_i is the lines of the right-hand side of (4) that belong to partition i; x_i is the lines of the solution that belong to partition i.

The BI GJ algorithm can be described in the following procedure:

Algorithm 4.3: Block-Iterative GJ

Inputs: Part = Partition sets, **Outputs:** Solution

8) **for each** p **in** Part
    1) factorize G[p;p] matrix
9) **set** starting solution **to** 0
10) **while not** converged
    1) **for each** set **in** Part
        1) **modify** right-hand side of the linear system using previous iteration solutions of neighbours (8)
        2) solve using previously computed factorization

As predictable, parallelism in the BI GJ algorithm is pretty straightforward to accomplish.

On the other hand, the original BI method is based on the SOR, in which the pattern is[15]:

$$A_{ii}\,\check{x}_i^{(k+1)} = b_i - \sum_{j=1}^{i-1} A_{ij}\,x_j^{(k+1)} - \sum_{j=i+1}^{m} A_{ij}\,x_j^{(k)} \tag{9}$$

$$x_i^{(k+1)} = (1-w)\,x_i^{(k)} + w\,\check{x}_i^{(k+1)} \tag{10}$$

The BI SOR method turns the original SOR technique into a block strategy, in which each block needs some of the neighbour solutions in order to proceed and there can be no dependency cycles. It seems like this characteristic binds the method to serialization, but this is not the case. Once a domain has all the neighbour solutions that it depends for current iteration, the solution can be assembled. Consequently, if two or more domains have all neighbour solutions needed, they can process in parallel.

The parallel BI SOR algorithm can be described as[15]:

Algorithm 4.4: Block-Iterative SOR

Inputs: Part = Partition sets, w = relaxation factor, **Outputs:** Solution

1) devise dependencies between sets
2) **assign** sets **to** computing instances
3) **for each** set **in** assigned sets
    1) factorize G[set;set] matrix
4) **set** starting solution **to** 0
5) **while not** converged
    1) **while any** set **in** assigned sets did not finish current iteration
        1) **find** an assigned set whose all dependencies have finished current iteration
        2) **modify** right-hand side of the linear system using available solutions of dependency neighbours (9)
        3) solve using previously computed set factorization
        4) **update** solution using equation (10)
        5) **send** updated solution to neighbour assigned computing instances

Note that only the boundary solutions need to be sent to neighbours, since others are not used. This is an acceptable communication rate per iteration.

In this case, one block is only allowed to begin processing after all dependencies have finished the current iteration, which leads to a parallelism type distinct from other algorithms. In order to capitalize

all the available resources, it makes sense to have a number of partitions that exceeds the number of computing instances so the ones responsible to process some partitions are not idle because they depend on others to start, but are rather processing another partition that is independent. This poses a problem in deciding how many partitions should exist in a general partitioning case (see Section 2.4).

The complexity of the Block-Iterative algorithm is very easy to analyse. It is determined by the number of iterations, complexity of solves and number of partitions and computing instances. So if there are C computing instances and m partitions, disregarding the dependency (in other words, all computing instances are always working), processing complexity is $O\left(\left(\frac{N}{m}\right)^{\alpha}\frac{m}{C}+z\,M\left(\left(\frac{N}{m}\right)^{\beta}\frac{m}{C}\right)\right)$, for z iterations. Memory complexity is $O\left(\left(\frac{N}{m}\right)^{\gamma}\frac{m}{C}+M\frac{N}{C}\right)$. Note that complexity is given by m Cholesky factorization solvers of N/m grids using C computing instances, doing one solve per each iteration. The α, β and γ parameters remain similar to ones described in Section 4.1.1 .

The convergence ratio of this method depends on the number of partitions. In general, a higher number of partitions leads to a lower convergence rate, but factorization and solving time per iteration decrease substantially (Figure 11).

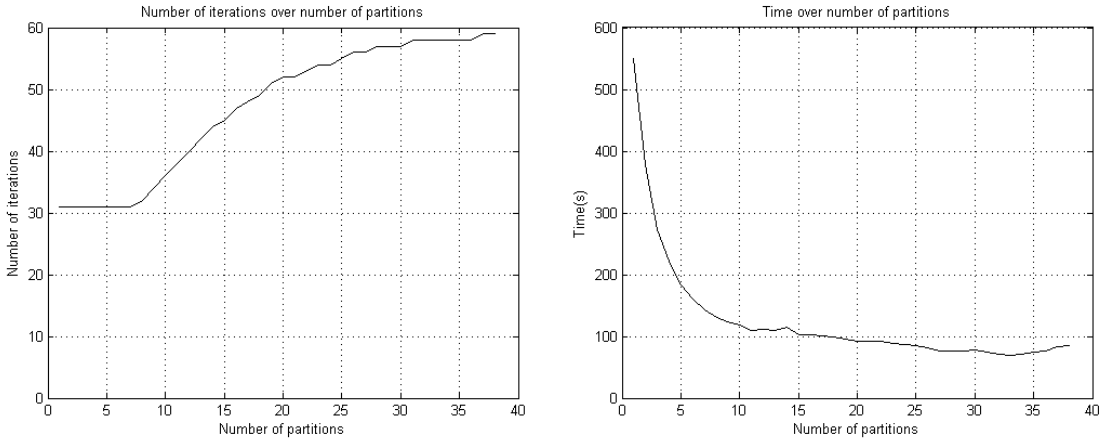

Figure 11: Block Iterative GS (SOR w = 1.8) method - number of partitions (m) convergence influence on number of iterations and time (500k node power grid with 20 current sources) in a single core computer

The performance stabilizes after about 10 partitions. This gives a good idea of how well the algorithm performs on a multiple core architecture. The relaxation factor is another parameter that influences the convergence rate, but not factorization and solution time, and thus the factor which reduces number of iterations the most also provides the least processing time.

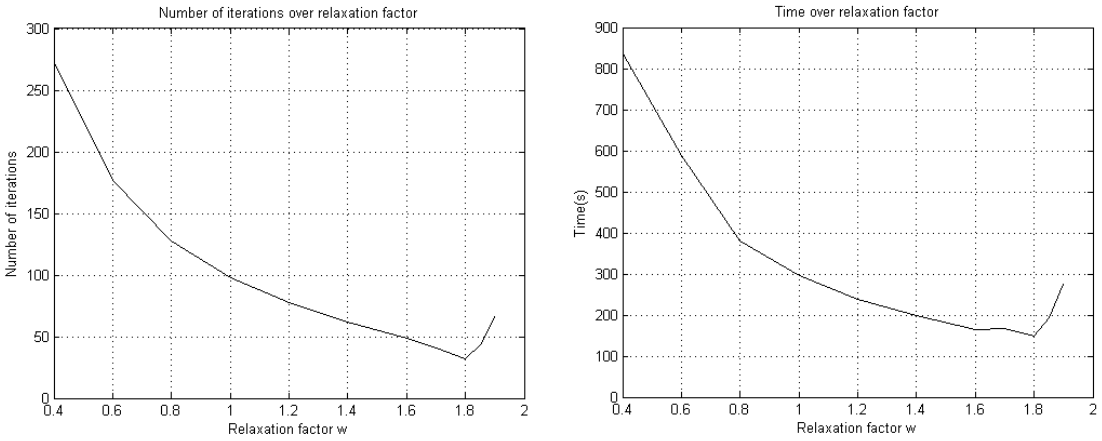

Figure 12: Block iterative SOR method - relaxation factor w convergence influence on number of iterations and time (500k node power grid with 20 current sources, m = 8) in a single core computer

The interesting thing to note here is the fact that the best relaxation factors lie in the 1.6 to 1.8 range.

On the other hand, the number of parallel instances depends on the partitioning graph (Figure 13 a graph which represents the connections between partitions) and the dependency projection (usually in the form of a directed graph).

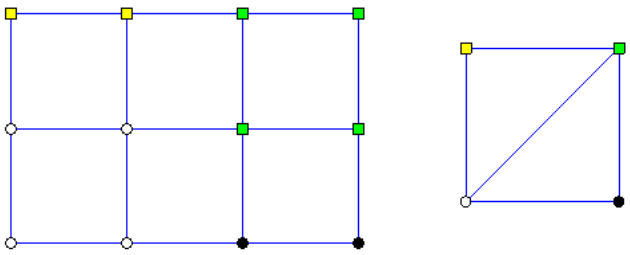

Figure 13: Partitioning graph example. Left side is a partitioned circuit with coloured nodes, one colour per partition. Right side is the equivalent partitioning graph.

The partitioning used by the authors of the BI SOR algorithm[15] is block partitioning (see Section 5.2) and a cardinal-based dependency graph is presented. In this study, the partition scheduling problem will be further analysed. An heuristic procedure is proposed to construct a good dependency graph for a generic partitioning algorithm:

Algorithm 4.5: Dependency graph constructor

Inputs: Part = Partition sets, m = number of partitions, **Outputs:** Dependency graph

1) **set all** sets **in** Part **to** unmarked
2) **for each** set **in** Part
    1) level(set) = 0
3) **for** n **from** 1 **to** m
    1) **find** bset = **min** level **of** unmarked sets (solving ties by least neighbour unmarked sets)
    2) **mark** bset
    3) **for each** neigh **in** neighbour unmarked sets of bset
        1) level(neigh) = **max**(level(neigh), level(bset)+1)
        2) **add** neigh **to** children of bset

Algorithm 4.5 is a greedy algorithm that seeks high parallelism without regard to number of dependencies between sets. For that reason, communication and computer instance assignment are problems yet to be solved. Nevertheless, they are very close to one another, given that the assignment must minimize the communication. On the other hand, the non-dependent partitions can be processed first in parallel, and then a partition can be solved after all its dependencies are done for current iteration. A good way of thinking is one that, given equal solving times for all partitions, occupies all the processing units during the BI algorithm execution.

The partitioning pattern favours a good result in the dependency plan algorithm application. To illustrate some patterns that stimulates a good dependency plan for the block partitioning strategy, an example of the previous algorithm application is presented for three different patterns in Figure 14. For this predictable partitioning type, it becomes very easy to assign computing instances. In the first pattern (Figure 14a) (m by 1), a possible assignment could be uniform, so if the number of computing instances is 3, the assignment would be (1, 1, 1, 2, 2, 2, 3, 3, 3), and the iteration solving order will be partitions (2, 3, 4) in parallel, followed by the rest. A similar strategy would apply in Figure 14e pattern ($\frac{m}{2}$ by 2). These patterns are preferable since they fully exploit parallelism in up to $\frac{m}{2}$ computing instances (and relationship between the pattern and the number of computing instances is important). On the other hand, Figure 14c pattern ($\sqrt{m}$ by $\sqrt{m}$) is not advised, since the optimal assignments have, in general, much more communication than the other two.

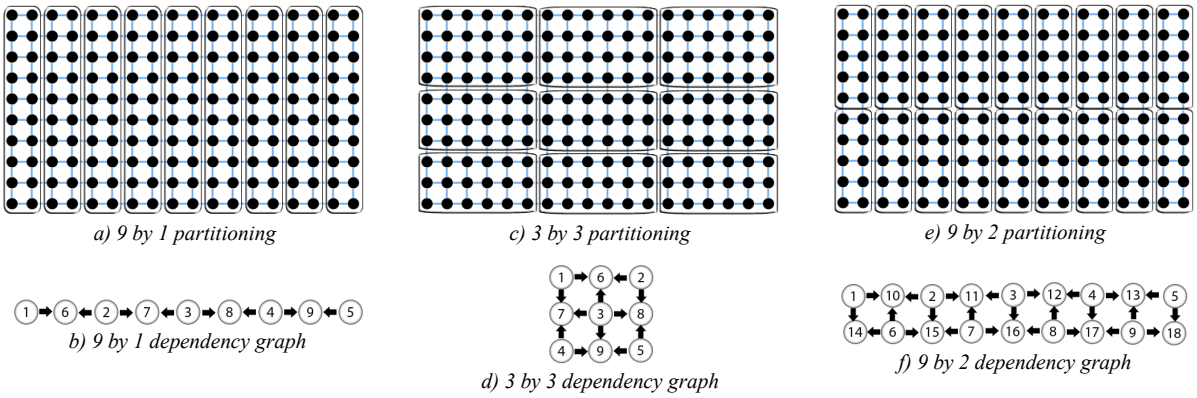

Figure 14: Grid possible partitions and respective dependency graphs generated by the above algorithm

In the general case, the number of computing instances is known, and so are the partitions graph. The problem at hand is to minimize the communication between computing instances. Putting those together, this calls for yet another graph partitioning. That said, the partitioning should balance the size and minimize the communication. This is a well-known partitioning problem and there are efficient strategies to tackle this problem (METIS[20], a graph-partitioning and fill-reducing software, is often a good choice). Ideally, this would take into account the non-dependent nodes, but the nature of Algorithm 4.5 tends to disperse these nodes.

The BI algorithm is a very efficient strategy to tackle the power grid problem as an iterative process. The GJ pattern is very easy to implement and completely distributes the workload and memory over computing instances. On the other hand, the SOR pattern is more tricky to implement and parallelize, but complements by improving the convergence rate per iteration considerably.

4.2.2 Locality-Driven Parallel Static Analysis

In the last subsection, an iterative algorithm named BI has been shown. Simple as it may be, it turns out to be a very efficient strategy. In this subsection, a different iterative scheme will be presented: the Locality-Driven Parallel Static Analysis (LPSA)[16].

Much like the BI algorithm, the LPSA also works in separated sub-domains, directly connected to each other. However, this iterative process relies on the superposition theorem, by developing the concept to the following: one can solve the system approximately, iterate by solving residual current sources and then sum all computed solutions to assemble the better approximation. That said, the LPSA method iteration has two phases: estimate boundary currents (in the branches that connect two partitions) and solve the system inside each partition using the estimated currents as current sources in the partition boundary nodes. The LPSA algorithm can be described as follows[16]:

Algorithm 4.6: Locality-Driven Parallel Static Analysis

Inputs: Part = Partition sets, thresh = convergence threshold, **Outputs:** Solution

1) **Determine** boundaries of partitions F
2) **Create** boundary windows W around boundaries in F with margins Mx and My
3) **for each** w **in** W
    1) **Compute** Cholesky decomposition of G[w;w] matrix
4) **for each** p **in** Part
    1) **Compute** Cholesky decomposition of G[p;p] matrix
5) Ires = I, $v^{(0)} = 0$, k = 0
6) **while** $||Ires||_\infty > thresh$
    1) **for each** w **in** W
        1) **Solve** the system for nodes within window w: $G[w;w]v_w = B_wIres_w$
        2) **Compute** boundary currents $I_B(F_i)$
    2) **for each** p **in** Part
        1) **Solve** the local system using boundary currents as current sources connected to the boundary nodes of p: $G[p;p]Vres_p = B_pIres_p - I_{Bp}$
    3) **Update** the solution $v^{(k+1)} = v^{(k)}+Vres$
    4) **Update** the residual Ires by computing the difference $Ires = Bi - Gv^{(k+1)}$
    5) **Increment** k

The notation A_p and A_w means submatrices of A with p or w row set and all columns

The fact that G[w,w] and G[p,p] are submatrices of G implies that outer nodes of the window and partitions are grounded (in other words, drop is fixed to zero) when the solution is being calculated inside the window or partition.

The estimation process proposed in [16] is to solve the system within a window around each partition boundary (Figure 15). The window contains a few V_{DD} bumps and the authors argue that a window size of 3 bumps width is enough for the algorithm to converge to the exact solution.

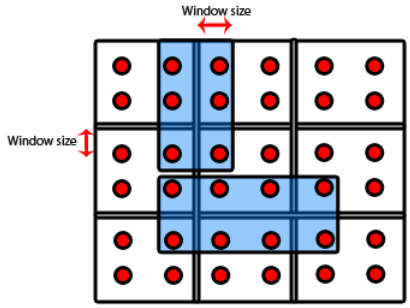

Figure 15: Partitioned (3 by 3) circuit boundary windows (blue) example. Red dots represent C4 Vdd bumps.

Most of the LPSA algorithm is effortless to parallelize. In fact, pretty much every expensive part of this algorithm is able to be distributed. However, there are downsides to this algorithm implementation and paralellization as well. The most obvious ones are related to the boundary windows. The number of boundary windows is not the same as the number of partitions, which poses a problem to the paralellization. Also, they are very hard to determine if the partitioning algorithm is generic. Once again, the partitioning algorithm proposed in [16] is the block partitioning (see Section 5.2), which makes this step trivial. In this work, this will also be the only partitioning algorithm used in conjunction with this power grid solving algorithm. The number of boundary windows is therefore given by $n_x(n_y-1)+n_y(n_x-1)$, where n_x and n_y are the number of divisions. For generic partitioning algorithms, one can use breadth-first search for each boundary node until a certain distance is reached and add those nodes to boundary windows. However, there will only be one boundary window for each of the partitions.

In the BI algorithm, partition number (or size) and the relaxation factor were important parameters for the convergence, even for the non-parallel case. In LPSA algorithm, there is no relaxation factor, but the window size is now a very important parameter for the convergence (Figure 16).

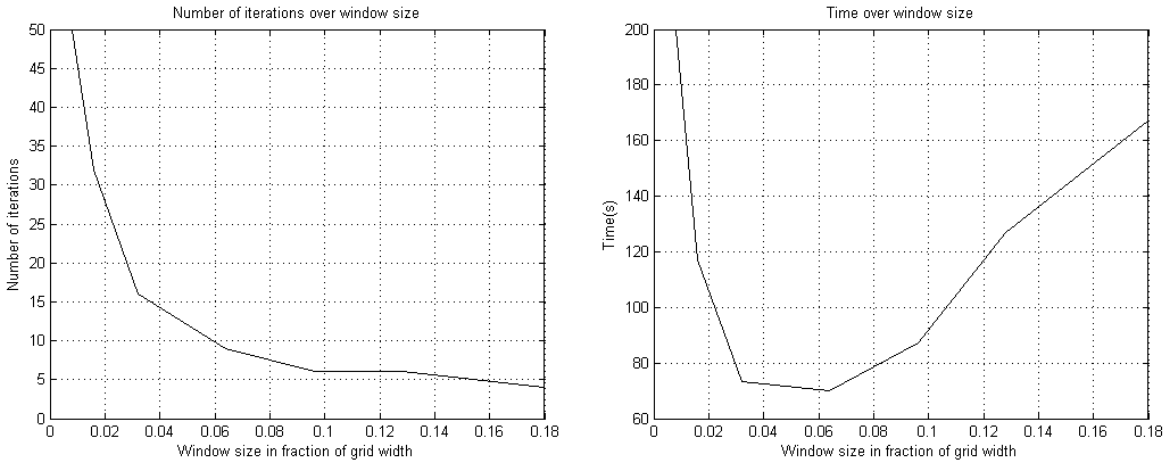

Figure 16: LPSA method - window size convergence influence on number of iterations and time (500k nodes power grid with 20 current sources, m=8)

As expected, the number of partitions still takes part of the algorithm convergence and time (Figure 17).

The complexity of the LPSA algorithm is very similar to the block iterative complexity, but with multiple solves each iteration (for inner-partition and boundary windows) instead of just one. Processing complexity is given by $O\left(\left(\frac{N}{m}\right)^{\alpha_1}\frac{m}{C}+\left(\frac{N}{nwin}\right)^{\alpha_2}\frac{nwin}{C}+zM\left(\left(\frac{N}{m}\right)^{\beta_1}\frac{m}{C}+\left(\frac{N}{nwin}\right)^{\beta_2}\frac{nwin}{C}\right)\right)$, with m = number of partitions, nwin = number of boundary windows, C = number of computing instances, z =

number of iterations. Memory complexity is $O\left(\left(\frac{N}{m}\right)^{\gamma_1}\frac{m}{C}+\left(\frac{N}{nwin}\right)^{\gamma_2}\frac{m}{C}+M\frac{N}{C}\right)$.

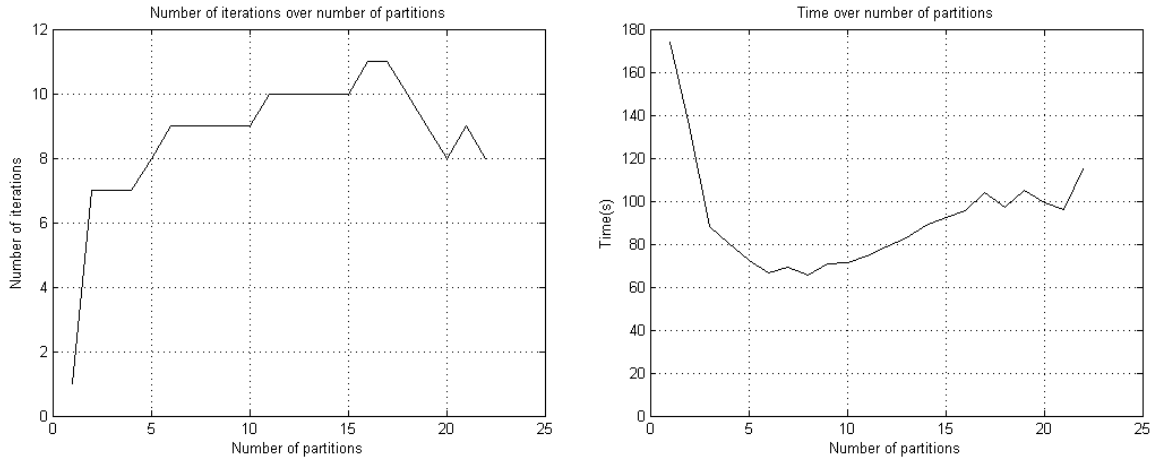

Figure 17: LPSA method - number of partitions convergence influence on number of iterations and time (500k node power grid with 20 current sources, window size = 6% of grid width)

The LPSA algorithm is very fast and has very good convergence properties, probably the best around in general. However, some windows are shared by two or more computing instances. In this case, two strategies can be used: either one of the computing instances process both parts of the window or both process; by distributing the factorization and solve. Unfortunately, both fail to achieve acceptable communication rates (order of square root of N) during the process. In the first case, half of the windows nodes need to be shared, which is intolerable: if a 2% of N window size is needed to achieve a good convergence, 2% of the power grid node values are communicated each iteration for each window associated with a computing instance. In the second case, the factorization process demands high communication rates, as stated in Section 4.1.1 .

4.2.3 Preconditioned conjugate gradient

The conjugate gradient algorithm is a widely known iterative algorithm for solving systems of linear equations, in which matrix A in linear system Ax = b is symmetric and positive definite. Theoretically, the number of iterations needed to reach the exact solution, neutralizing the numerical error, is equal to the number of equations. However, it usually takes a much smaller number to reach an acceptable solution in terms of residue, given by r = b-Ax. In order to accelerate the rate of convergence, a technique called preconditioning is often used. This technique consists in a transformation leading to the system $M^{-1}Ax = M^{-1}b$. The rational for this procedure is that solving this system is easier, i.e., requires less iterations, than solving the original system, at little additional cost per iteration. This results in an operation applied to the residue r that approximates a solution ς in $A\varsigma$ = r by solving $M\varsigma$ = r in a cheaper process. The ideal preconditioning operation would give the exact

solution ς in Aς = r and the preconditioned conjugate gradient (PCG) would reach the solution in the first iteration. Obviously, this requires a direct solution of the original system and is therefore fruitless.

The preconditioning options studied over the years are manifold. The most natural choices and

Algorithm 4.7: Block-Jacobi Preconditioned Conjugate Gradient

Inputs: Part = Partition sets, thresh = convergence threshold, **Outputs:** Solution

1) **for each** p **in** Part
    1) factorize G[p;p] matrix for preconditioning
2) **set** starting solution **to** 0
3) $r = B$
4) **for each** p **in** Part
    1) **Let** ς_p and r_p be submatrices of ς and r with p row set and all columns
    2) **Solve** $G[p;p]\varsigma_p = r_p$ using previously computed factorization
5) $q = \zeta$
6) **for** a **from** 1 **to** M
    1) let ζ_a, r_a, d_a and q_a be the a columns of ς, r, d and q
    2) $d_a = \zeta_a^T r_a$
7) **Loop until break**
    1) **for** a **from** 1 **to** M
        1) let ζ_a, r_a, q_a and x_a be the a columns of ς, r, q and x
        2) $Gqa = G\,q_a$
        3) $\alpha = d_a / (q_a^T Gqa)$
        4) $x_a = x_a + \alpha q_a, r_a = r_a - \alpha q_a$
    2) $\|r\|_\infty$ < thresh
        1) **break**
    3) for each set in Part
        1) **Let** ς_p and r_p be submatrices of z and r with p row set and all columns
        2) **Solve** $G[p;p]\varsigma_p = r_p$ using previously computed factorization
    4) **for** a **from** 1 **to** M
        1) **let** ζ_a, r_a and q_a be the a columns of ς, r and q
        2) $d_a^{next} = \zeta_a^T r_a$
        3) $q_a = \zeta_a + \frac{d_a^{next}}{d_a} q_a$
        4) $d_a = d_a^{next}$

their application to the power grid problem will be discussed in this section. In [14], the following preconditioner options, details and implementations are specified, as well as the preconditioned conjugate gradient.

The first and most basic preconditioner is called Jacobi preconditioner. This preconditioner works particularly well for diagonally dominant matrices, since its approximation grows better the more preponderant the diagonal values of A are.

The SSOR (Symmetric Successive Over-Relaxation) preconditioner is a little more sophisticated than Jacobi. It is derived from the decomposition of the A matrix and based on the SSOR method. The preconditioners discussed so far are easy to parallelize and distribute in terms of both initialization and application. For example, the SSOR preconditioner has the same non-zero pattern as the G matrix, hence solve can be parallelized by recurring to a technique very similar to the distributed matrix-matrix multiplication[14].

Next is the set of the so called incomplete factorization (ILU or IChol) preconditioners[14]. The IChol preconditioners are known for being volatile in structural and numerical ways, providing a good trade-off between initialization and convergence rate. The choices used in this work are the IChol(0) (no fill-in incomplete factorization) and ICholT (incomplete factorization with thresholding). The ICholT preconditioners usually provide very good results, but are very limited when it comes to parallelism in the terms already stated. IChol PCG was applied to the power grid analysis by Chen and Chen[17], who had fast results in comparison to Cholesky decomposition or Conjugate gradient.

The sparse approximate inverse (SPAI) preconditioners[18] are often a good choice for parallel types of preconditioning. These are based on a sparse estimation of the inverse of A, called M, which is the minimizer to the norm $||I-M*A||_F$ under sparse restrictions. These restrictions include dropping the values under some threshold or a fixed percentage of the values in each column of the estimated inverse amidst the process. This estimation is easily done in parallel, but requires too many matrix-vector multiplications in the initialization process, which inhibits its use.

Last but not least, the Block Jacobi (BJ) preconditioners[14]. This type of preconditioners are the most flexible to distribute. The BJ preconditioner consists in subdividing the matrix into smaller parts with resort to a partitioning algorithm (in which the partitioning options come forward) and independently computing the solution for each of the domains in parallel for each iteration. In other words, this preconditioner computes factorization and solution of smaller parts of the grid (just like BI algorithm), and uses them to approximate the whole solution, making use of the locality nature of the power grids. This solver is yet to be applied to the power grids and this study will make this step first.

The complexity of this solver is identical to the BI SOR complexity.

In Figure 18 and Figure 19, the previously stated preconditioner options for non-parallel case are compared. It is plain to see that Block-Jacobi performance is similar to Incomplete Cholesky. As the BJ provides a strong parallelization potential, it is evident that BJ is the best choice for the power grid analysis through the Preconditioned Conjugate Gradient method.

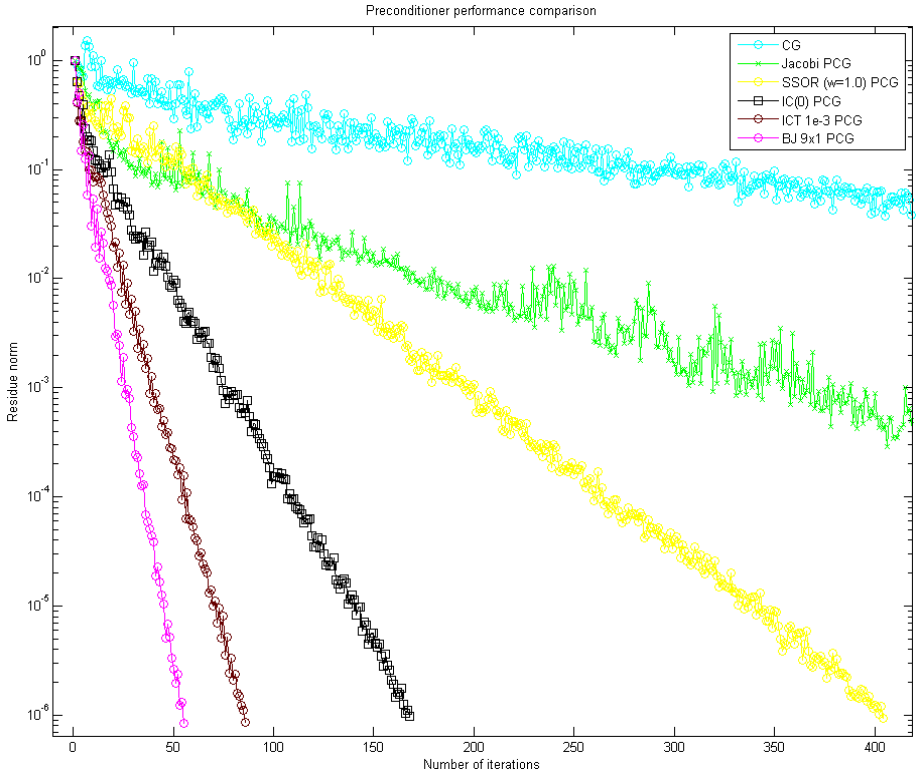

Figure 18: Preconditioner comparison – residual error for a 62k node IBM benchmark power grid with 50 current sources (MATLAB)

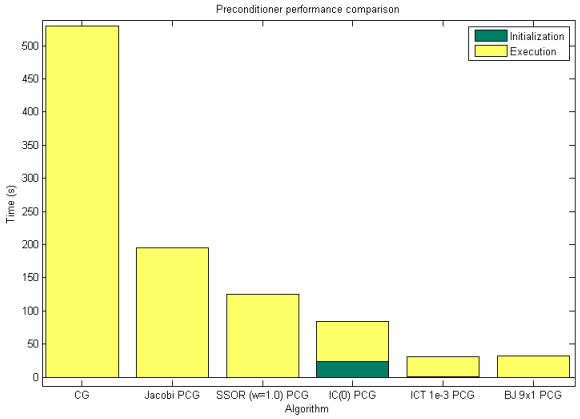

Figure 19: Preconditioner comparison - Time to reach 10^-6 error for a 62k node IBM benchmark power grid with 50 current sources (MATLAB)

4.3 Final analysis

In this chapter, strategies for finding the solution of the power grid analysis problem were seen. The direct methods like Cholesky factorization and DD are very fast and have a long history of improvements since the Cholesky factorization was invented. However, they very quickly meet their limits due to the lack of scalability, flexibility to distribute data and fill-in associated problems. On the other hand, iterative solutions such as BI SOR, LPSA and BJ PCG seem very promising and are worth comparing.

5 Partitioning

As described before, the power grid analysis can easily reach intolerable dimensions. In order to reduce the size of the problem or the intensity of the requirements, multiple techniques such as MOR were examined in Chapter 3 . MOR techniques reduce the size of the system matrix effectively, but are bound to increase the number of nonzeros in the matrix G amidst the reduction at some point. On the other hand, the distribution of resources to each computing instance also deals with the intensity problem. This calls for a division that must be accomplished in some way. The method explored in this study is to partition the power grid, which is useful to use in conjunction with the algorithms discussed in Chapter 4 .

In this Chapter, partitioning (sometimes called clustering) techniques will be studied. Graph partitioning is a NP-complete problem which consists in assigning the graph nodes to multiple sets, while minimizing some property, such as relationship amongst the sets. Graph partitioning application to power grids has two main purposes: to distribute the data, as balanced as possible in terms of number of inner-partition nodes, among computing instances; to maximize the approximation of the inner solution (when compared to the whole system solution and obtained using only inner-partition nodes) and real solution (of the partition nodes, obtained by simulating the whole circuit). That said, the relationship between sets is often measured as the sum of the branch weights interconnecting the sets (which are branch conductancess for the case of power grids). Also, the most attractive partitioners are those which balance the size of each set, avoiding very large sets that lead some instances to drudge while the remaining are idle, limiting the advantages of partitioning.

Graph partitioning is an old problem and there are many of graph partitioning algorithms targeting different applications, as well as efficient general-purpose programs (like METIS[19] and Chaco[20]). Kernighan and Lin proposed a partitioning algorithm[21] which later became very important in the area of VLSI layouts. In this work, the most widely known partitioning algorithms that could be applied to the power grid problem were examined. In this section, the presented algorithms are three, namely:

1. Domain partitioning
2. Geometric partitioning
3. Ratio cut

5.1 Domain partitioning

The domain partitioning[13] (DP) is a simple algorithm, formulated in conjunction with the DD

method (see Section 4.1.2). It is based on a fast label reproduction scheme and requires the selection of a fixed number of nodes for the starting domains. The method consists in expanding each domain until it cannot be further enlarged without colliding with other domains. The resulting domains are called partitions.

The DP algorithm is as follows[13]:

Algorithm 5.1: Domain Partitioning

Inputs: sel = selected initial nodes, adj = node adjacencies, **Outputs:** Partition sets

1) **Assign** label 0 **to** all nodes
2) **for each** v_i **in** sel **do** label(v_i) = i and $S_i = \{v_i\}$
3) **while not** converged
    1) **for each** v_i **in** subset S_i, i = {-1, 1, 2, 3, ..., m}
        1) **for each** v_j **in** adj(v_i)
            1) **if** label(v_j) **is** 0 **do** label(v_j) = label(v_i); $S_i = S_i \cup \{v_j\}$
            2) **else if** label(v_i) **is neither** label(v_j) **nor** -1 **do** label(v_j) = -1; $S_{-1} = S_{-1} \cup \{v_j\}$

Note that step 3.1.1.2 answers the purpose of identifying interface nodes (see Section 4.1.2 for interface node definition) in case they are needed. However, that should be omitted if no identification is needed and all the nodes will be associated with exactly one partition. Also, convergence is not reached as soon as all nodes are labelled, considering that nodes might still be relabelled and become interface nodes.

The initial selection takes an important role in the performance of Domain Partitioning. Initial nodes are autonomously chosen through innumerable possibilities. The method used in this study is a uniform distribution along the top layer of the grid (which is the least dense). The reason behind the one-layer distribution comes from the connections between layers. In fact, conductances between nodes in consecutive layers (also called vias) are very large and nodes with high connectivity should share the same label. This cannot be achieved perfectly without changing the algorithm process, but this selection method ensures a reasonable labelling based on this assumption.

Consider a 4x4 grid where domain partitioning with random initial selection is applied (Figure 20). The domains are iteratively expanded to the neighbours, and some of the interface nodes come from both unlabelled and labelled nodes.

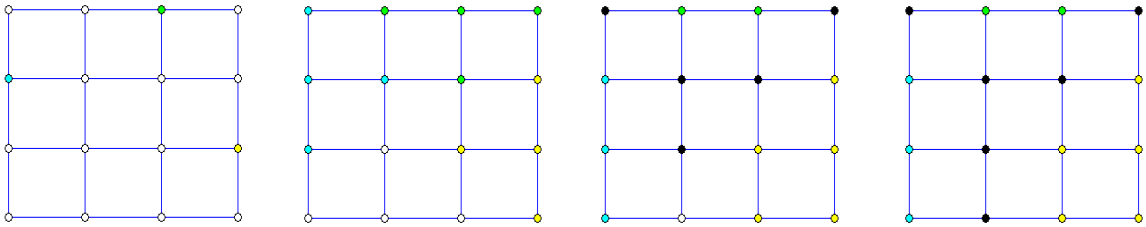

Figure 20: Domain Partitioning example. Colors: white - unlabeled; light blue, green, yellow - partitions; black - interface

The domain partitioning algorithm is a very simple and fast approach. It focuses on fast expansion and, generally, balances the set sizes. However, it disregards the relationship between different sets and the outcome is very sensitive to initial selection of nodes.

5.2 Geometric partitioning

The geometric interpretation is an essential procedure in graph partitioning. The power grids may not be regular structures, but they have geometric patterns (Figure 21). Hence, to partition the grid using geometric information (node position) makes sense and it turns out to be a very effective strategy that provides good results.

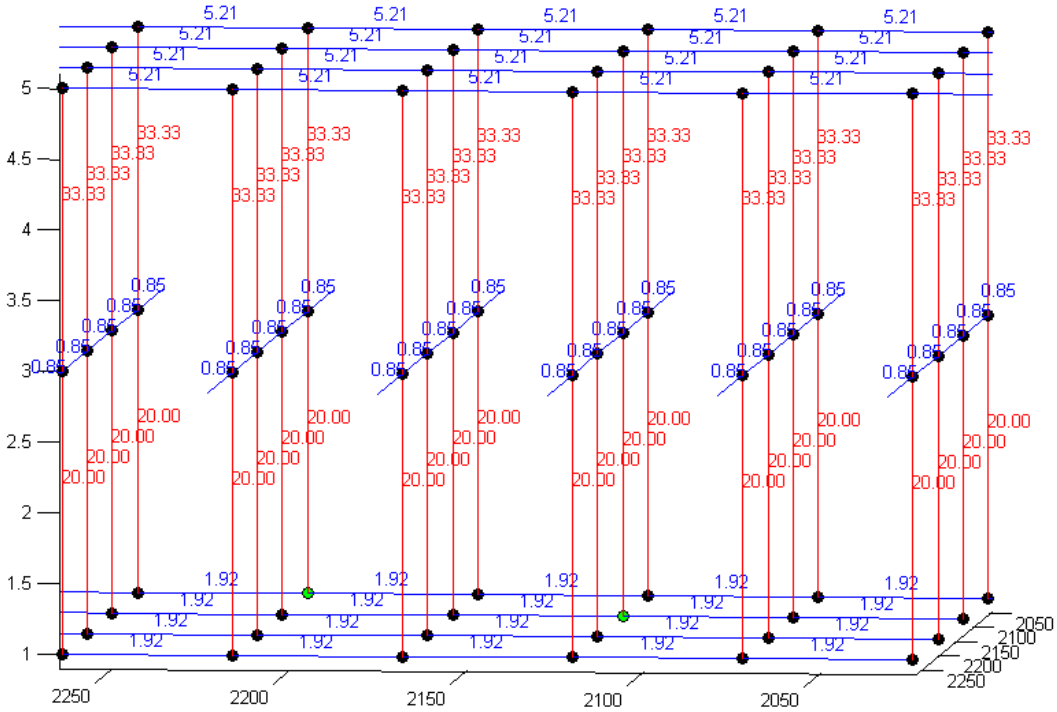

Figure 21: Representation of portion of an IBM benchmark power grid. Lines represent intra-layer (blue) and interlayer (red) conductances. Axes are Cartesian coordinates in 3D (micrometer units).

The geometric partitioning (GP), also called area-based partitioning and block partitioning, is not

particularly hard to achieve: Cut the grid uniformly and balance the nodes in each partition. Then again, the conductances between layers (also called vias) are large in comparison and thus the grid is cut based on the single layer directions only, x and y. Along these lines, a node in an upper layer will always belong to the same partition that the lower layer node it is connected to does.

The GP algorithm results in the following:

Algorithm 5.2: Geometric Partitioning

Inputs: cd = node coordinates, nx = number of cuts x, ny = number of cuts y, **Outputs:** Partition sets

1) xmin = min cd_x, xmax = max cd_x

2) ymin = min cd_y, ymax = max cd_y

3) width = xmax - xmin, height = ymax - ymin

4) **for** i **from** 0 **to** nx-1

    1) xlow = xmin + i*width/nx, xhigh = xlow + width/nx

    2) **for** j **from** 0 **to** ny-1

        1) ylow = ymin + j*height/ny, yhigh = ylow + height/ny

        2) $S_{i*ny+j} = cd_x \in [xlow, xhigh] \cap cd_y \in [ylow, yhigh]$

The choice of cut numbers has implications. Ideally, the cuts would be along one direction only[15], so to minimize the inter-partition connections (both in number of nodes/branches and number of partitions). However, this is not always best considering that some algorithms might require at least one grounded resistor (resulted from the grounded voltage sources) connection inside each partition to work properly (since matrix can be singular if that does not happen), and thus compromising the maximum number of cuts along one direction. This is not the case for any of the algorithms applied in this study.

In Figure 22, an example of geometric partitioning can be seen.

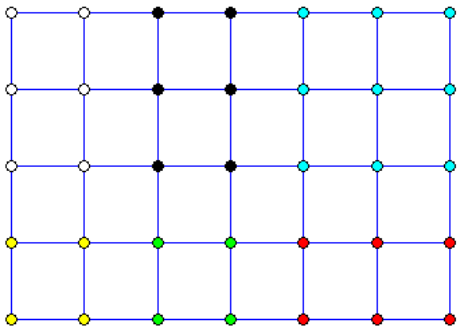

Figure 22: Geometric partitioning example: nx = 3, ny = 2 in a 7x5 grid. Each partition is denoted by a different color.

Once again, this partitioning strategy is apparently oblivious to the conductances interconnecting the sets. However, that is not entirely the case. The resistance values, typically obtained through an extraction process, are proportional to length and most nodes are equidistant, which promotes GP as a very efficient algorithm.

5.3 Ratio cut

The previous graph partitioning algorithms are based on geometric postulations. However, these postulations might be wrong, be it in the present or in the future. Besides, geometric or functional block information might be inaccessible or inaccurate, jeopardizing the partitioning performance substantially. In that event, a geometrically independent approach is desired, and one that optimizes the size of all partitions and the interconnecting conductances. In this study, a partitioning method called Ratio Cut is introduced to the power grid domain and applied for the first time.

The Ratio cut is a graph partitioning strategy which targets the following optimization problem[22] (metric):

$$\min_{\{V_1 \dots V_n\}} \frac{1}{\sum_k |V_k|\left(|V| - |V_k|\right)} \sum_{\substack{i \in V_p \\ j \in V_q \\ p \neq q}} W_{i,j} \tag{11}$$

where:

V_i – Partition set I, $|V_i|$ is the set size

$W_{i,j}$ – Weight on branch which connects node i to j

In short, Ratio cut is a strategy that attempts to optimize set partitioning such that the sum of interconnecting weights over the product of partition sizes is minimum. The number of partitions is either fixed or variable, depending on the algorithm and/or the application.

Ratio cut was introduced by Wei and Cheng[23] as a circuit partitioning subject and a preliminary solution based on linear programming was proposed. Hagen and Kahng[24] added an heuristic solution for the 2-way approximated Ratio cut: computing the eigenvector associated with the second smallest eigenvalue of the Laplacian of the graph, called *Fiedler vector*. This is cheaply computed resorting to the Lanczos algorithm for sparse symmetric eigenvalue problem[14]. For the multi-way cut problem, one can either apply the 2-way cut recursively or employ the strategy stated in [24]: find the most profitable cuts in the sorted eigenvector by brute force. In [25], the Ratio cut problem was deeply

studied and finer Linear (LP) and Quadratic programming (QP) solutions were proposed, but the excessive amount of variables stuns the application of these LP and QP approaches for circuit partitioning.

The Ratio cut heuristic algorithm is described as[24]:

Algorithm 5.3: Ratio cut Heuristic Partitioning

Inputs: m = number of partitions, W = weighted adjacency matrix, **Outputs:** Partition sets

1) **Compute** diagonal degree matrix $D = \underset{i}{diag}\left(\sum_{j=1}^{N} W_{i,j}\right)$
2) **Calculate** the Laplacian L = D-W
3) **Estimate** the Fiedler vector v using the Lanczos algorithm
4) **Sort** v values ascending or descending
5) **Compute** the most profitable m-1 cuts in v by testing all cutting possibilities and using the ratio metric (11)
6) Cluster the graph from the cuts found

Note that step 5 can be very expensive, even for n = 2. Alternatively, the cuts are computed in a greedy way, starting with uniform cutting (m sets with the N/m size) and moving the cut left or right until no longer profitable. A simulated annealing helps in improving the optimization. The maxima of the eigenvector discrete derivative (with care to the sort direction) is also used as a fast method.

An alternative to this approach is called Normalized cut[26], in which the objective is to maximize the degree (sum of internal weights) of each partition rather than the size in number of nodes[26]. Since the computation complexity of the algorithms are usually related to the number of nodes in the partitions, the Ratio cut approach is more appropriate for the power grid applications.

5.4 Interface node identification

Some of the partitioning algorithms previously seen do not identify interface nodes, needed for the DD method (see Section 4.1.2). The method employed for this identification is to simply find nodes with adjacencies that belong to another partition and label them as interface nodes:

Algorithm 5.4: Interface node classification

Inputs: label = partition identification, **Outputs:** Solution

1) **for each** node **in** grid
    1) **if** label(node) **is not** interface label
        1) **for each** nodeadj **in** adjacencies of node
            1) **if** label(nodeadj) **is neither** interface label **nor** label(node)
                1) **set** label(node) **to** interface label
                2) **break** inner loop

5.5 Final notes

In this section, different partitioning algorithms that rely on different prepositions were presented. These algorithms were used in conjunction with the strategies presented in the Chapter 4 and their results are shown in the Chapter 7 .

6 Implementation details

In this Chapter, the libraries and data structures used in the implementation of the previously stated solutions are referenced. In addition, partitioning and solution processes are explained, as well as how the multithreaded communication, common to every parallel algorithm, can be achieved.

6.1 Libraries used

The implementation of all algorithms previously discussed was written in C++ and compiled with GCC. For non-distributed data, matrix Cholesky decomposition, sparse matrix multiplication and permutation are provided by CHOLMOD library[11] (the currently fastest sparse Cholesky factorization library available). For multithreaded processing, pthreads library (IEEE standard 1003.1c) is used for BI algorithm and OpenMP API (version 3.0) for remaining ones. Distributed processing resorts to OpenMPI library[27] (version 1.4.3).

6.2 Data Structures

The KCL conductance matrix and its submatrices are very sparse, so a sparse matrix representation is appropriate for these matrices. For compatibility with the CHOLMOD library, the representation used is Column compressed storage. An example of the compressed column storage can be seen in the following scheme:

$$\begin{bmatrix} 1 & 0 & 2 \\ 0 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix} \begin{array}{ll} & \textit{column number of nonzeros}: \quad [0\,2\,4\,7] \quad (\text{size } N+1) \\ \rightarrow & \textit{row indices}: \qquad\qquad [0\,2\,1\,2\,0\,1\,2] \quad (\text{size } NZ) \\ & \textit{non zero values}: \qquad [1\,5\,3\,6\,2\,4\,7] \quad (\text{size } NZ) \end{array}$$

The column vector identifies the accumulated number of nonzeros at the start of each column (ending with the total), an information which can be used to access the rest of the data. The nonzero values for the c column are Values[**from** ColumnNonzeros[c] **to** ColumnNonzeros[c+1]-1] and their respective rows are RowIndices[**from** ColumnNonzeros[c] **to** ColumnNonzeros[c+1]-1].

All other matrices are stored in dense formats, represented by single arrays (storing columns consecutively). Even though B matrix is a very sparse matrix, iterative algorithms use it as residue, which becomes dense after very few iterations. Hence, B matrix is stored in a dense format.

6.3 Partitioning

Partitioning the grid is a two-step process to comply with the distributed and shared (multithreading case) data types of processing, as the first requires communication that need to be identified, a process which becomes harder if a one-step strategy like partitioning the grid by total number of threads in all computers is applied.

In the first step, the grid is partitioned in C parts, where C is the number of computing instances and each computing instance stores the G matrix columns (or rows, since it is a symmetric matrix) associated to its assigned first stage partition. B and s matrices share the same strategy. This successfully distributes memory and allows the algorithms to successfully execute on multiple computing instances. If the implementation is multithreaded only, first step is skipped. In a second stage, the "subgrid" is partitioned into the same number of pieces as the number of threads the computing instance supports, or more in the case of BI SOR algorithm (twice the amount, as it proved to be a good strategy in Section 4.2.1). This number can also be user specified, to exchange convergence rate for memory usage (the more the grid is partitioned, the less the convergence rate per iteration is, but the memory usage also diminishes, and the processing time may grow or decrease). The same flexibility exists for parameters such as the window size in LPSA and relaxation factor in BI SOR.

6.4 Partition solution

In the parallel algorithms, it is often useful to solve for some specific nodes (from one partition) only. One way to do this is by computing the submatrices of the right-hand side for the specific nodes, copy them to a new matrix, find a solution and fill the general solution matrix. An example of this is:

$$t = r_{set}, G_{set} x = t, s_{set} = x$$

This implies that memory used for s and r is doubled, and in a critical place (since matrices s and r are NxM and dominate the memory usage of every algorithm). To overcome this problem, it is a good choice to implement a subset solve. Since partitions have no overlap, this is very straightforward to accomplish:

Algorithm 6.1: Subset solve

Inputs: U = Upper factor with diagonal 0, D = Diagonal values of the upper or lower factor, r = right-hand side, part = partition nodes, perm = permutation, **Outputs:** s = solution

1) **for** k **from** 1 **to** M
   1) let s_k and r_k be column k of s and r
   2) **for** a **from** 1 **to** size(part)
      1) let U_a be column a of U
      2) $s_k[part[perm[a]]] = r_k[part[perm[a]]] - s_k[part[perm]]U_a$
   3) **for** a **from** 1 **to** size(part)
      1) $s_k[part[perm[a]]] = \frac{s_k[part[perm[a]]]}{D[a]^2}$
   4) **for** a **from** size(part) **to** 1
      1) let U_a be column a of U
      2) $s_k[part[perm]] = s_k[part[perm]] - s_k[part[perm[a]]]U_a$

This will provide a solving speed very similar to CHOLMOD solving routine, and will not require the memory and processing overhead of copying. For this to be possible, the CHOLMOD factorization cannot be supernodal (must be simplical) and the factor must be converted to a regular sparse matrix to work with. It is trivial to use OpenMP since there are no shared nodes between multiple partitions.

6.5 Message Passing Interface and Multithreaded data sharing

Each message shared by Message Passing Interface (MPI) must be identified with a unique tag, which can only be reused when the message it identifies has arrived its destination. As the algorithms that need to constantly share data are iterative, it is almost always possible to identify what is going to be shared in each iteration and synchronize what is expected to send and receive in the preprocessing. In this manner, the data can be buffered in an agreed order by the sender and the receiver for an asynchronous transference and there is no need for much overhead identifying messages. In addition, only a fixed amount of messages needs to be shared each iteration.

The user needs to be careful when multiple threads are responsible for generating the data that need to be transferred. As threads have different processing times, neither the data that is already processed nor the processor that has more to process need to wait. Two solutions can be employed to overcome this problem: an additional thread to manage transferences; or there has to be multiple messages per iteration, at least one from one thread to each thread in any other computer. In either

way, the synchronization preprocess has to assign different tags to each message according to which thread generated the data. The transference management must also take into account that message order is not guaranteed (in the case of a managing thread, this can be solved by periodically probing the messages with the routine provided by MPI instead of waiting for a specific one).

7 Results

The power grid analysis is a very expensive process in terms of memory and processing complexities. In this chapter, a comparison between the various strategies will be made using a multithreaded architecture. The best among the previously studied strategies will be studied in terms of performance, memory spent and flexibility of distributing data.

7.1 Environment

The results in this chapter obtained from running implementations of the algorithms discussed in Chapters 3 and 4 , using two different architectures:

1) Multithreaded architecture:
   1. Multi-core processor with 4 cores – Intel Xeon E5410 2.33GHz
   2. RAM Size: 24GB
   3. Operative System: Linux Fedora 13
2) Distributed architecture:
   1. Cluster with 10 Multi-core processors with 8 cores each – Intel Xeon E5504 2.00GHz
   2. RAM Size: 32GB per computer
   3. OperativeSystem: Linux CentOS 5.4 (Final)

7.2 Power grids

The power grids used for comparison were of two types: power grid benchmarks provided by IBM power grids (available from http://dropzone.tamu.edu/~pli/PGBench) and artificially created power grids. As the IBM provided power grids are very small in comparison to the target (of 300Million nodes), artificially created power grids with similar characteristics were generated.

7.2.1 IBM benchmark power grids

Real power grids used are IBM benchmark power grids. The V_{SS} part is eliminated and the zero resistance branches are simplified (terminal nodes are fused). The number of current sources is arbitrarily chosen in order to test scalability of algorithms with increasing problem size. In Table 4, the most relevant characteristics of the realistic benchmarks power grids are described.

Power grid	Nodes	Voltage sources	Current sources
IBMPG1	6085	100	30
IBMPG2	61677	120	90
IBMPG6	403915	132	130
IBMPG6-2	403915	132	300

Table 4: Power grid IBM benchmarks

7.2.2 Artificially created power grids

Artificial power grids have a pre-specified structure (that is user configurable). They are generated according to rules such as: each pair of layers (in the z-axis) has the same density, but lower layer pairs have an higher density than upper layer pairs; total resistance of the power grid is constant; each layer only has resistors in either x-axis and z-axis or y-axis and z-axis. These rules were suggested by observing the IBM benchmark power grids structural information. The number of nonzeros of G is approximately 4.2N.

Power grid	Nodes	Voltage Sources	Current sources
ART1	146700	120	120
ART2	782978	120	160
ART3	782978	120	300
ART4	1197252	120	300
ART5	1197252	120	500
ART6	1197252	120	750
ART7	1197252	120	1000
ART8	1197252	120	1250
ART9	1197252	120	1500
ART10	1197252	120	1750
ART11	1197252	120	2000
ART12	2918100	120	300
ART13	4998932	120	300
ART14	7982598	120	300

Table 5: Artificially created power grids

7.3 Comparison

In this section, the various strategies previously discussed in this study will be compared. All partitioning possibilities previously discussed will be compared, as well as all solvers, for some power

grids in a multithreaded architecture (Section 7.1).

The conditions for the comparison are:

1) Number of partitions m = 8 (12 for LPSA)
2) LPSA window size = 2% grid width
3) BI SOR w = 1.65
4) Acceptable residue error $|r|_{max} = 10^{-3}$

Tables 6, 7, 8 and 9 show comparison among various solution and partitioning strategies.

Part Algorithm	Part Time	DD time	BI SOR	BJ PCG	LPSA
DP	0.009s	0.027s	16 / 0.376s	15 / 0.213s	-
GP (m by 1)	0.001s	1.37s	16 / 0.448s	8 / 0.095s	did not run
METIS	0.007s	0.099s	16 / 0.207s	16 / 0.159s	-
Ratio Cut Multi-way	0.067s	0.026s	16 / 0.437s	11 / 0.114s	-

Table 6: IBMPG1 power grid results (iterations / time)

The first comparison show that all partitioning algorithms show potential on different solving strategies. The power grid used in this first test (IBMPG1) has four connected components (sets of nodes with no outer connections). As the LPSA implementation did not separate these into different problems, one of the boundaries has no connections and its boundary window also has no nodes. This is the reason why it did not run in this power grid. All other power grids have one connected component only (this can be very easily attained in a multiple connected component graph using breadth-first search).

Part Algorithm	Part Time	DD time	BI SOR	BJ PCG	LPSA
DP	0.064s	19.7s	1123 / 740s	203 / 120s	-
GP (m by 1)	0.009s	15.2s	16 / 6.61s	21 / 9.25s	14 / 6.92s
METIS	0.058s	12.9s	16 / 10.2s	22 / 10.2s	-
Ratio Cut Multi-way	1.38s	20.3s	17 / 15.3s	26 / 21.4s	-

Table 7: IBMPG2 power grid results (iterations / time)

Part Algorithm	Part Time	DD time	BI SOR	BJ PCG	LPSA
DP	0.133s	107s	19 / 35.7s	35 / 72.9s	-
GP (m by 1)	0.022s	81.7s	18 / 30.1s	33 / 56.5s	17 / 20.2s
METIS	0.152s	57.3s	28 / 132s	37 / 107s	-
Ratio Cut Multi-way	14.1s	223s	48 / 198s	46 / 141s	-

Table 8: ART1 power grid results (iterations / time)

Part Algorithm	Part Time	DD time	BI SOR	BJ PCG	LPSA
DP	2.76s	340s	192 / 651s	61 / 284s	-
GP (m by 1)	0.066s	284s	125 / 160s	47 / 133s	21 / 91.9s
METIS	0.220s	237s	17 / 239s	24 / 140s	-
Ratio Cut Multi-way	8.43s	435s	36 / 223s	24 / 68.9s	-

Table 9: IBMPG6 power grid results (iterations / time)

The Geometric Partitioning is the most reliable partitioning algorithm to use in conjunction with most of the algorithms (except for Domain Decomposition, which is METIS), providing the best results overall. There were some outliers like BJ PCG with Ratio cut partitioning on IBMPG6 power grid, which did not show good results in ART1 in comparison to any other strategy. The domain decomposition method is the only one profiting with a different strategy, but is left behind by the iterative methods (that is the downside of computing an exact result only and having no scalability). The LPSA algorithm cannot be distributed efficiently (as discussed in Section 4.2.2).

The most efficient solvers are BI SOR and BJ PCG. As the communication per iteration is identical in both strategies, comparing multithreaded versions is the same as distributed. A final comparison will be made to choose which strategy provides the best results. In Table 10, BJ PCG proves to successfully top BI SOR. Conditions are the same except the number of partitions m is 16 (GP pattern m x 1).

Power Grid	BI SOR	BJ PCG
IBMPG6	132 / 115s	51 / 145s
ART2	78 / 1650s	52 / 1235s
ART4	92 / 6888s	83 / 5242s

Table 10: Results for BI SOR and BJ PCG solvers with GP algorithm (iterations / time)

7.4 Distributed BJ PCG results

In this section, a distributed version BJ PCG will tackle increasingly difficult power grids on a

distributed architecture (Section 7.1) using Geometric partitioning. The next table shows some quick examples of how much is the distributed BJ PCG worth in terms of processing (with 8 partitions per computer):

Number of computers	Time (s)	Memory spent per unit (GB)	Speed-up
1	424	4.72	0%
2	381	2.37	11%
3	265	1.61	60%
4	185	1.20	129%
5	175	0.95	142%
6	155	0.79	174%
7	188	0.66	126%
8	139	0.60	205%
9	159	0.57	166%
10	125	0.52	239%

Table 11: Speed-up and memory results for distributed processing over 1 to 10 computing instances, simulating IBMPG6-2 power grid benchmark (404k node with 300 current sources)

It is evident that distributing the data will ease the memory burden over each computer, but the previous table showed that even the processing cost benefits from the distribution and provides some good speed-ups. Sometimes the speed-up may drop a bit (case 7 and 9 computers in the previous sample) due to where the power grid was cut if it is not perfectly regular.

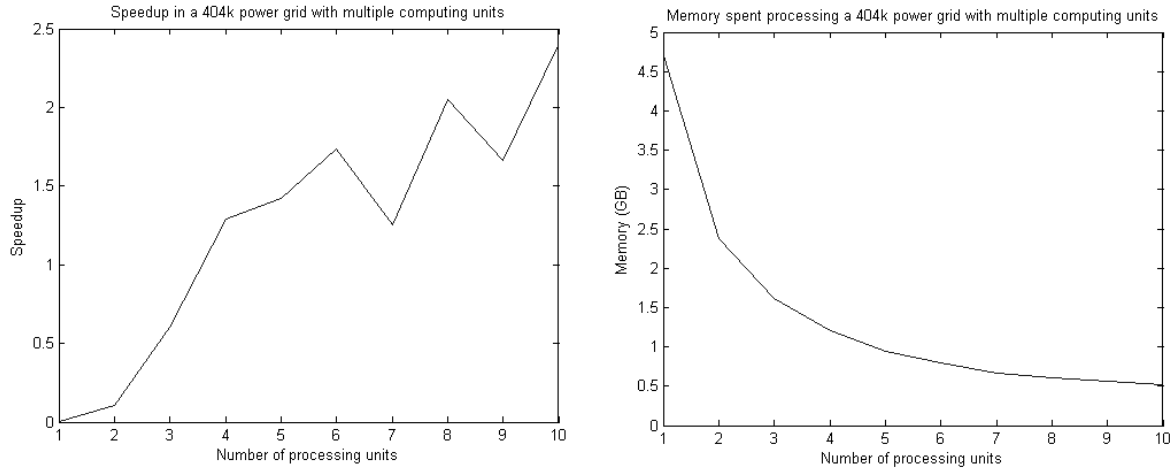

Figure 23: Plot results for Table 11

The next table shows results for increasing number of nodes.

Power Grid	Number of nodes	Number of Iterations	Time (s)	Memory used per computer (GB)
IBMPG6-2	403k	80	147	0.53
ART3	783k	127	494	0.96
ART4	1.2M	160	1036	1.47
ART12	2.9M	209	4440	3.60
ART13	5M	211	8309	6.18
ART14	7.9M	248	17660	10.03

Table 12: Block-Jacobi Preconditioned Conjugate Gradient on a 10 octocore cluster results in iterations, time and memory peak per computer

In Figure 24, the evolution of the previous results can be seen over the number of nodes. Memory is heavily dominated by matrices with dimensions N*M: the solution matrix s, the preconditioner solution ς, the projection vector q and residue matrix r = B-G*s, which become dense after just only a few iterations. G*q can also be preallocated. So, it is not surprising that the evolution of memory spent is linear with the number of nodes (and will be with the number of current sources too). The projected

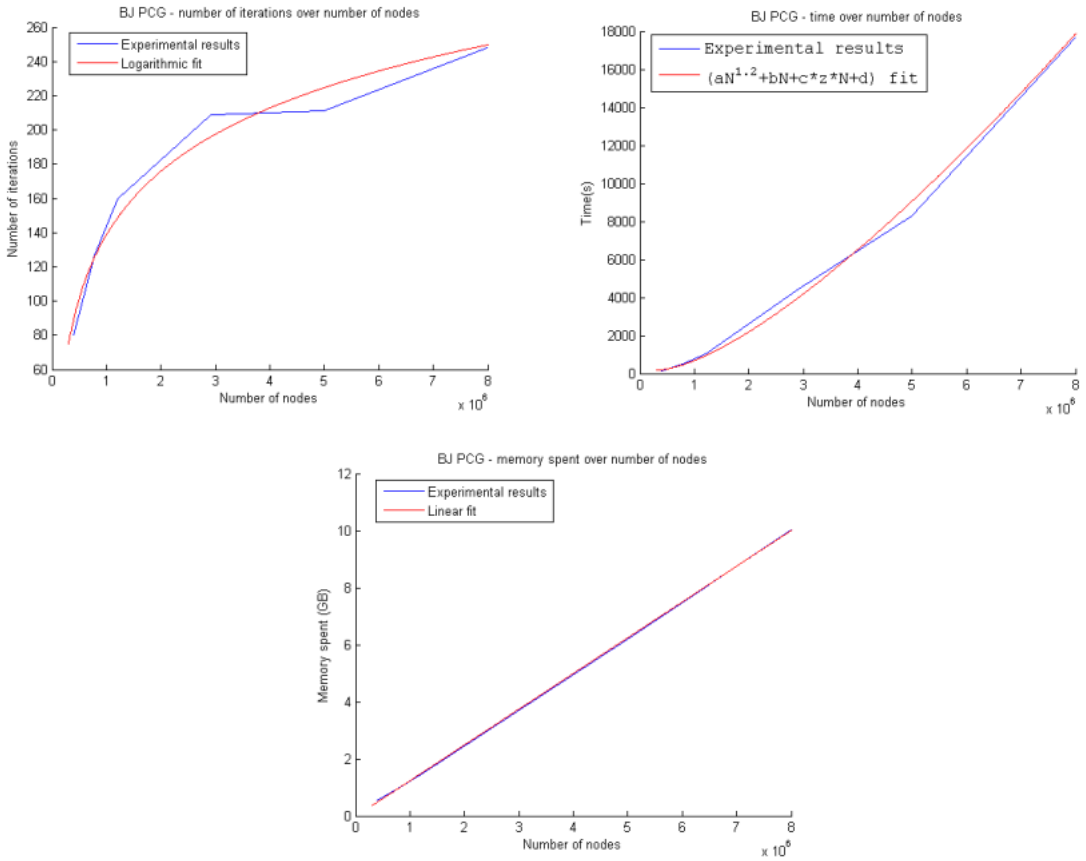

Figure 24: BJ PCG results - Number of nodes influence on number of iterations, time and memory spent

complexity $O(N^{1.2} + z*N*M)$ models the time curve fairly well.

In Table 13, the algorithm will run for increasing number of current sources:

Power Grid	Number of current sources	Number of Iterations	Time (s)	Memory used per computer (GB)
ART4	300	160	1036	1.47
ART5	500	158	2141	2.41
ART6	750	161	3589	3.48
ART7	1000	159	5062	4.61
ART8	1250	161	6266	5.71
ART9	1500	159	7838	6.84
ART10	1750	159	8911	7.95
ART11	2000	160	10660	9.06

Table 13: Block-Jacobi Preconditioned Conjugate Gradient increasing node number on a 10 octocore cluster results in iterations, time and memory peak per computer

As expected, the number of iterations does not vary considerably, since the grid is the same and is structurally regular, solving does not depend much on the chosen nodes and all pretty much take the same number of iterations to converge. Also, time and memory complexity are approximately linear with the number of current sources (since factorization times are constant). In Figure 25, the curves for the previous results and their linear fits can be seen.

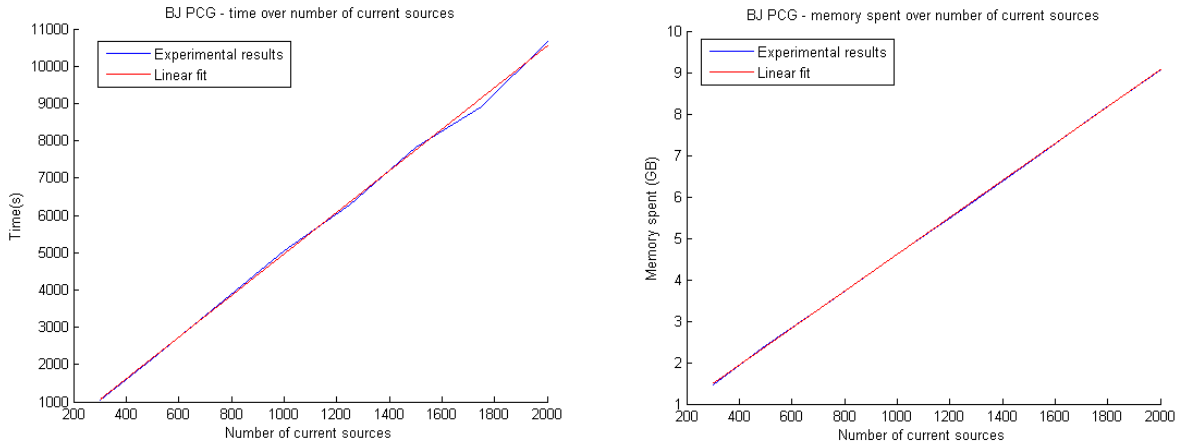

Figure 25: BJ PCG – Influence of the number of current sources

The memory resources rise very quickly with the number of nodes and current sources. However, if the right-hand side is made in blocks, the memory drops substantially without increasing the processing times much, and actually decreasing for a low number of blocks. The following table

shows the results for varying the number of blocks in ART7:

Number of blocks	Time (s)	Memory used per computer (GB)
1	5062	4.61
5	3212	0.99
10	3395	0.53
20	3870	0.31
40	5339	0.19
60	6145	0.16
100	5270	0.13
1000	40770	0.09

Table 14: Block-Jacobi Preconditioned Conjugate Gradient increasing number of blocks (of a total of 1000 current sources) on a 10 octocore cluster results in iterations, time and memory peak per computer. Please note that 1000 is not dividable by 60

7.5 Extrapolation

It is now intended to extrapolate the results and complexity of BJ PCG algorithm to solve a power grid of 300Million node with 2000 current sources.

The NxM matrices will use about 24 terabytes (TB) total. As the distributed architecture employed in this study is composed of 10 computing instances, there is a theoretical need for 2.4TB storage in each. For that reason, 120 current source blocks will be used, which is about 20GB per machine for each of these matrices. For a KCL conductance matrix of this size, with a mean of 4.25 non-zero values per column (a common value in the IBM benchmark power grids), would need a total of 23GB, which is 2.3GB if divided by 10 computing instances. Each computing instance will store 8 factorizations of 3.75M nodes which contain about 73.8M nonzeros (using N^1.2) and results in about 1.2GB size each factorization. This sums up to a memory usage of 29.2GB, a very much possible amount for the current machines to hold.

The processing is a little more tricky to extrapolate because it depends on the number of iterations. Using the logarithmic fit curve, the projected number of iterations of a 300Million node power grid is 443 (note that it does not depend on the number of current sources). The following table shows some results needed for extrapolation:

Number of nodes	Number of blocks	Total number of current sources	Number of iterations (mean)	Time (s)
150k	120	2000	85	988
1.2M	120	2000	160	7070
2.9M	120	2000	210	23690
5M	120	2000	215	41890

Table 15: Block-Jacobi Preconditioned Conjugate Gradient results for 120 blocks of 2000/120 current sources power grids

Using the complexity of BJ PCG to estimate the time curve in Figure 26, the time needed to tackle a 300M power grid with 2000 sources on this distributed architecture setup is $5.213*10^6$ seconds, or 60.3 days.

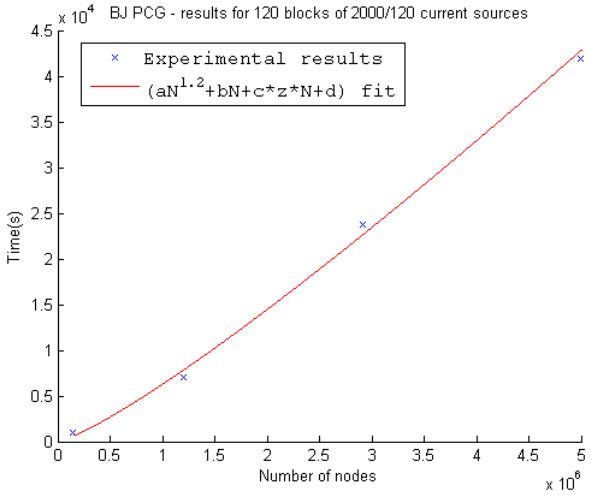

Figure 26: Experimental results of BJ PCG for 120 blocks of 2000/120 current sources

8 Conclusions

In this work, the power grid analysis formulation was used to determine the voltage values of power grid nodes and indirectly solve the EM problem. Current power grids in aggressive designs build using the latest technology nodes can have on the order of 300 million nodes and 2000 current sources. For these cases, the problem requirements exceed the limits of what a single computer can do, and parallel solutions compatible with distributed data had to be developed.

A tempting way of approaching the size problem is by model data compression. It has been seen that whole Model Order Reduction techniques when applied to power grids can produce smaller systems, the level of reduction achieved is disappointing and furthermore the resulting matrices are dense, as opposed to the very sparse power grid conductance matrices. The combination of these two facts implies that such techniques are not appropriate for the power grid analysis problem.

Several solution methods have been examined and developed. Direct solution methods have met limitations in terms of distributed solutions, low communication rates, low fill-in amounts and good result scalability. On the other hand, iterative solutions have proved to be successful in all these terms, and provided very satisfactory results. Block-Jacobi Preconditioned Conjugate Gradient, an algorithm applied in this work for the first time in the analysis of power grid, seemingly provided the best results in terms of convergence and performance, while keeping a communication rate and complexity very similar to the former best algorithm Block-Iterative.

Block-Jacobi Preconditioned Conjugate Gradient solved a 7.9M node power grid with 300 current sources in 5 hours in a 10 octocore cluster, shows a slightly superlinear complexity in terms of CPU and is therefore believed to be able to solve a 300M node power grid with 2000 current sources in 60.3 days using 29.2GB memory per computer. It is not a fantastic result, but with dedicated architectures, the time spent will drop. Even though this is a projection based on benchmark and artificially created power grid results, it shows there is still a long way to go to completely solve a power grid problem of such size.

A different solving strategy is not the only way to reach a better result. Experiments show that some algorithms have faster convergence in the first few iterations and slower afterwards, while others are slower first and faster after. Hybrid strategies employing multiple algorithms (that share a similar preprocessing) will improve the performance considerably. Also, the power grid connections have value and structure patterns, important properties to study in order to apply data compression and reduce both the memory spent storing the grid and the communication rates in the process.

9 References

[1] Nassif, S. R., *Power Grid Analysis Benchmarks*, IBM Research, Austin, USA, 2008

[2] Sheehan, B., *Realizable reduction of RC networks*, IEEE Transactions Computer-Aided Design of Integrated Circuits and Systems, 2007 Aug., pp. 1393-1407

[3] Odabasioglu, A., Celik, M., and Pileggi, L. T., P*RIMA: passive reduced-order interconnect macromodeling algorithm*, IEEE Trans. Computer-Aided Design, August 1998, pp. 645–654

[4] Feldmann, P., Model order reduction techniques for linear systems with large number of terminals. In DATE'2004, v. 2, 2004 Feb., Paris, France, pp. 944–947

[5] Yan, B., Zhou, L., Tan, S. X.-D., Chen, J., and McGaughy, B., *DeMOR: decentralized model order reduction of linear networks with massive ports*, In ACM/IEEE DAC-08, pages 409-414, Anaheim, CA, 2008 June

[6] Feldmann, P. and Liu, F., *Sparse and efficient reduced order modeling of linear subcircuits with large number of terminals,* In IEEE/ACM International Conference on Computer-Aided Design, 2004 Nov., pp. 88-92

[7] Yu, H., He, L. and Tan, S. X.-D., *BSMOR: Block structure preserving model order reduction*, In IEEE International BMAS, Sep. 2005

[8] Phillips, J. R. and Silveira, L. M., *Poor man's TBR: A simple model reduction scheme*, In DATE'2004, 2004 Feb., Paris, France

[9] Silva, J.M. and Silveira, L.M., On the Compressibility of Power Grid Models, IEEE ISVLSI '07, 2007 Mar., pp. 186-191

[10] Amestoy, P., Davis, T. A., and Duff, I. S., *An approximate minimum degree ordering algorithm*, SIAM Journal on Matrix Analysis and Applications, v. 17, issue 4, Dec. 1996, pp. 886-905

[11] Chen, Y., Davis, T. A., Hager, W. W. and Rajamanickam, S., *Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate*, ACM Trans. Math. Software, v. 35, issue 3, 2008 Oct.

[12] Golub, G. and Loan, C., *Matrix Computations*, JHU Press, 1996

[13] Quming, Z., Sun, K., Mohanram, K., Sorensen, D. C., *Large power grid analysis using domain decomposition*, In Proc. DATE '06, 2006, pp 27-32

[14] Saad, Y., *Iterative methods for Sparse Linear Systems,* 2000, Jan. 3rd

[15] Zhong, Y. and Wong, M.D.F., *Fast Block-Iterative Domain Decomposition Algorithm for IR Drop Analysis in Large Power Grid*, In Proc. 11th ISQED, 2010 Mar., pp. 277-283

[16] Zeng, Z., Li, P. and Feng, Z., *Parallel Partitioning Based On-Chip Power Distribution Network Analysis Using Locality Acceleration*, In Proc. 10th ISQUED, 2009, pp. 776-781

[17] Chen, T. and Chen, C.C., *Efficient large-scale power grid analysis based on preconditioned krylov-subspace iterative methods*, In Proc. DAC '01, 2001 June

[18] Benzi, M., Tuma, M., *A comparative study of sparse approximate inverse preconditioners*, Applied Numerical Mathematics, v. 30, issue 2-3, New York, 1999 June

[19] Karypis, G. and Kumar, V., *METIS, A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices ver. 4.0.1*, 1998 Nov. (http://glaros.dtc.umn.edu/gkhome/metis)

[20] Hendrickson, B. and Leland, R., *The Chaco User's Guide: Version 2.0*, Sandia Tech Report SAND94--2692, 1994 (http://www.sandia.gov/~bahendr/partitioning.html)

[21] B., Kerningan and S., Lin, *An efficient heuristic procedure for partitioning graphs*, The Bell system technical journal, v. 49, 1970, pp. 291-307

[22] Chen, S. J. and Cheng, C.K., *Tutorial on VLSI Partitioning*, In Journal VLSI Design, v. 11, 2000 Feb., pp. 175-218

[23] Wei, Y. C. and Cheng, C.K., *Towards Efficient Hierarchical Designs by Ratio Cut Partitioning*, In Proc. ICCAD-89, 1989 Nov., pp. 298-301

[24] Hagen, L. and Kahng, A.B., *New Spectral Methods for Ratio Cut Partitioning and Clustering*, IEEE Computer-Aided Design of Integrated Circuits and Systems, v. 11, 1992 Sep., pp. 1074-1085

[25] Fan, N. and Pardalos, P.M., *Multi-way clustering and biclustering by the Ratio cut and Normalized cut in graphs*, In Journal of Combinatorial Optimization, Springer, Issn 1382-6905, 2010, pp. 1-28

[26] Shi, J. and Malik, J., *Normalized cuts and image Segmentation*, IEEE Pattern Analysis and Machine Intelligence, v. 22, 2000, pp. 888-905

[27] Squyres, J. M. and Lumsdaine, A., *The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms*, In Proc. 18th ACM International

Conference of Supercomputing, Springer, pp. 167-185, 2004 July, France