N° Ordre : 1880

THÈSE

présentée

pour obtenir

LE TITRE DE DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

Spécialité : **INFORMATIQUE**

par

Jean-Christophe RIOUAL

Solving linear systems for semiconductor device simulations on parallel distributed computers

Soutenue le 23 Avril 2002 devant le Jury composé de :

M.	Americo	MARROCCO	*Président*
M.	Ivan G.	GRAHAM	*Rapporteur*
Mme	Marina	VIDRASCU	*Rapporteur*
M.	Patrick	AMESTOY	*Directeur*
M.	Iain S.	DUFF	
M.	Luc	GIRAUD	
M.	Gérard	MEURANT	

Thèse préparée au CERFACS, CERFACS Report Ref : TH/PA/02/49

Aknowledgments

Je remercie Madame Vidrascu, directeur de recherche à l'INRIA, d'avoir accepté de rapporter sur ce travail.

I would like to express my thanks to Professor Ivan Graham who accepted to act as a referee for my dissertation.

I would like to express my sincere gratitude to Professor Iain Duff, group leader in the Parallel Algorithm team at CERFACS. Thank you for taking part in this jury.

Je remercie Monsieur Patrick Amestoy, maitre de conférence à l'ENSEEIHT, d'avoir accepté d'être directeur de cette thèse. Je le remercie ainsi que tous les membres de l'équipe de développement du logiciel MUMPS pour leurs conseils et leur aide.

Mes remerciements vont tout particulièrement à Monsieur Luc Giraud, chercheur senior au CERFACS, qui a encadré cette thèse. Je le remercie pour sa rigueur scientifique et son attention permanente à mes travaux. Ces trois années d'apprentissage furent pour moi celles d'un enrichissement intellectuel constant.

Merci a Monsieur Marrocco, d'avoir accepté d'être un membre de ce Jury. Ses travaux, en tant que directeur de recherche dans l'équipe M3N de l'INRIA, sont une des bases de cette étude. Je tiens à le remercier d'avoir été toujours disponible pour répondre à mes questions.

Je remercie Monsieur Meurant, directeur de recherche au CEA, d'avoir accepté de prendre part à ce jury.

Je tiens également à remercier l'association EGIDE et en particulier le projet Aurora pour avoir financé une collaboration avec le laboratoire Parallab de l'Université de Bergen en Norvège. Merci à Jacko Koster, chercheur au Parallab.

Merci à tous les membres de l'équipe Algorithmes Parallèles du CERFACS et à ceux du service informatique.

A mes parents, pour m'avoir toujours soutenu. A ma soeur, Nathalie, et mes neveux et nièce, Mathilde, Nicolas, Clément.

A tous mes amis.

Solving linear systems for semiconductor device simulations on parallel distributed computers

Abstract

In this thesis, we study the parallel distributed implementations of the linear solvers for the systems involved in 2D semiconductor device modelling. The semiconductor devices are modeled using the drift diffusion equations with the electrostatic potential and the quasi-Fermi levels as unknowns. The objective of this work is to develop, based on an existing sequential code, a complete parallel code for a distributed memory environment with MPI as message-passing library. The main difficulty consists in the efficient implementation of suitable linear solvers. In this respect we investigate both parallel direct methods and non-overlapping domain decomposition techniques. As a central software tool we consider MUMPS that is a parallel distributed implementation of the multifrontal technique for sparse matrices. This software is used either as a black box or as a building box for implementing direct or iterative substructuring approaches. In the iterative case, we consider preconditioned Krylov methods for solving the Schur complement systems. Various preconditioners including multi-level techniques are considered as the Balanced Neumann-Neumann preconditioner for the SPD systems. We also investigate several scaling strategies for the Schur complement system. We report on comparative parallel performance of direct and iterative solvers on real test problems. Finally, we present a preliminary study of a two-level preconditioner that exploits some spectral information of the preconditioned systems.

keywords : semiconductor simulation, linear solvers, distributed computers, domain decomposition, multifrontal method, preconditioners, scaling.

Contents

Introduction

Nowadays, we can solve really challenging linear algebra problems by combining direct and iterative methods, see [46]. Domain decomposition techniques provide a rather natural way to combine those two approaches when solving partial differential equations numerically. From an algebraic point of view, the domain decomposition methods can be divided into two main groups, the Schwarz methods also referred to as overlapping domain decomposition algorithms and the Schur complement methods also referred to as non-overlapping domain decomposition algorithms.

At the end of the 19^{th} century, Schwarz [97] proposed, when possible, to divide the domain into two subdomains that have a simpler geometry and where the solutions of the equations are known. These subdomains overlap so that part of the solution of one problem in a simple subdomain can be used as boundary condition for the solution in the other subdomain. In the eighties, this idea was the basis for many iterative methods proposed for solving numerically PDE's, see for instance [44, 80, 81]. Today the domain decomposition methods that induce some overlap between the subdomains are called Schwarz methods.

The Schur complement method is an alternative that consists in dividing the domain into approximately equal subdomains that are disjoint, that is, they do not overlap. By dividing the unknowns into interior and interface unknowns, one computes the Schur complement of the matrix formed by the entries of the interior points in the complete problem, see [37]. Then, the reduced system for the unknowns defined on the interfaces is solved and, subsequently, the complete solution is computed. A general theory has been developed since the eighties to explain the underlying properties of those methods in order to obtain better solutions, see for instance [1, 42, 43, 82].

For solving the reduced problem on the interfaces that appears in a Schur complement method, one must decide whether to use a direct or an iterative method. The Schur complement methods are also referred to as substructuring algorithms. The method is referred to as direct substructuring if the Schur complement system is solved by a direct method and iterative substructuring if it is solved by an iterative method.

In the framework of a joint research effort between CERFACS (Centre Européen de Recherche et de Formation Avancées en Calcul Scientifique) and INRIA (Institut National de Recherche en Informatique et Automatique) we study the parallel distributed implementation of 2D semiconductor device modelling and in particular

substructuring methods for solving the linear systems involved.

A sequential code has been developed at INRIA, with an in-house skyline Cholesky or LU direct solver for solving the linear systems arising during the simulations. This sequential code is able to deal with problems based on a mesh with up to 30000 triangles. This solver was efficient for simulating homojunction transistors. If we turn to heterojunction transistors, the meshes must be strongly refined at the level of the heterojunctions and discretizations with more than 500 000 elements have to be considered. Parallelizing the code becomes mandatory to solve problems of this size as a complete simulation becomes very much CPU demanding.

We have considered a parallelization for distributed memory environment with MPI [70] as message passing library. The overall numerical simulation consists in a semi-implicit Euler time scheme coupled with a Newton solver at each time step. At each Newton step a linear system that can be either symmetric positive definite (SPD) or unsymmetric has to be solved. The formulation is mainly vectorial and most of the resulting code is naturally parallelizable. The main difficulty consists in the efficient implementation of some suitable linear solvers. This latter numerical kernel is the most time consuming part of the code. In this respect we investigate both direct and iterative substructuring algorithms.

As a central software tool we consider MUMPS that is a parallel distributed implementation of multifrontal technique for sparse matrices [4, 5, 47]. We use MUMPS in order to compute the factorization of the local internal subproblems but also to obtain an explicit computation and storage scheme for the Schur complement matrix. We also present an implementation of the direct substructuring algorithm in which the interface problem is solved using MUMPS.

In the case of iterative substructuring, the Schur complement matrix is generally badly conditioned and the Schur system has to be preconditioned in order to guarantee a small number of iterations of the solver. Preconditioners for Schur complement methods in the context of elliptic problems has been a prolific area of research in the last twenty years. State-of-the art preconditioners consist of local and global components. The basic role of the global part is to provide an overall mechanism for the communication of the residual at each iteration. Without such a mechanism, the condition numbers of the preconditioned matrices become exponentially dependent on the number of subdomains, see [110]. The local part captures the strong couplings that appear between neighbouring points on the mesh. In the area of local preconditioners, we can find quite a few propositions and we refer to: Dirichlet-Neumann [16, 17], Neumann-Neumann [38, 39, 103], Probing [31, 33, 77], and J-operator [42]. For a complete overview of these local preconditioners and other aspects related to domain decomposition, we refer to [32, 92, 101, 102]. The components that are responsible for the global coupling between the subdomains are referred to as coarse-space corrections. A well known and often cited two-level preconditioner is BPS [23] that was the first two-level preconditioner to be proposed. We can also cite the Balancing Neumann-Neumann [84, 104, 105], the FETI [54, 85], and the Vertex Space [43, 100]. A class of two-level additive Schwarz preconditioners

for the Schur complement has been proposed in [26, 27, 28].

The numerical behaviour of these preconditioners have been precisely studied in the case of linear elliptic partial differential equations that lead to the solution of SPD linear systems. In the context of our semiconductor simulations, we test similar ideas for nonlinear PDE's that can lead to SPD as well as to unsymmetric linear systems. A collaboration between CERFACS and Parallab laboratory of the university of Bergen has been developed in the framework of an Aurora project supporting the scientific collaboration between Norway and France. During this collaboration, we have tested Balancing Neumann-Neumann for the SPD systems arising during semiconductor simulations. In the case of the unsymmetric Schur complement systems we test two-level additive preconditioners as the ones presented in [27].

The Schur complement systems involved in semiconductor simulations are challenging to solve. The entries in the Schur matrices exhibit very large variations in magnitude with some jumps larger than twenty orders of magnitude. In order to improve the robustness of the numerical methods, we investigate some scaling techniques.

The linear systems solved are embedded in a nonlinear scheme and changing the linear solver might change the nonlinear path and the total number of Newton steps required to obtain the steady state of the simulation. The convergence of the nonlinear scheme is a criterion to test the robustness of the different algorithms. Therefore a fair comparison between iterative and direct methods is possible. For other works concerning the application of substructuring methods to semiconductor device modelling we refer to [36, 62].

This manuscript is organized as follows. In Chapter 1, we present the physical problem and the mathematical and numerical tools used to solve it. In Chapter 2, we describe the parallel implementation of the linear solvers. In Chapter 3, we propose preconditioners and scaling techniques for solving iteratively the Schur complement system. In Chapter 4, we discuss the performance of parallel direct and iterative solvers for the simulation of semiconductor devices. Finally, in Chapter 5, we present some ideas that might deserve future research investigations.

Chapter 1

Numerical simulation of a transistor

In this chapter, we present the main principles of the numerical simulation of a transistor. In Section 1.1, we present briefly the physics of semiconductor materials and the mechanisms of a transistor of the NPN type. We also present the mathematical model selected to simulate this device. In Section 1.2, we present on a model problem the numerical tools used. Finally, in Section 1.3, we describe how these tools are used to compute the steady state of an NPN transistor in amplification mode.

1.1 Physical application and mathematical modelling

1.1.1 The transistor effect

Quantum physics describes the energy levels of an electron in an atom (measured in electron-volts (eV)) as discrete and not continuous. Pauli's exclusion principle says that two electrons in the same atom cannot share the same energy level. When two atoms are close to a distance of their own atomic radius, each energy level is split into two energy levels of close intensities. In the case of a crystal, that is a large number of atoms linked together in a complex structure, the number of energy levels can become so important that we can speak of quasi-continuous energy levels. Then the electronic structure of an atom of the crystal can be decomposed in permitted and forbidden energy bands (see Figure 1.1). Two bands are especially important, the valence band contains the electrons at the periphery of the crystal atoms and the conduction band is the energy band immediately after the valence band. The electrons in the conduction band are no longer influenced by their original atoms.

Insulators, conductors or semiconductors are materials defined by the energy gap between the valence band and the conduction band in their atoms (see Figure 1.2). In the case of an insulator, the valence band is separated from the conduction band by a gap of several eV and the electrons cannot go from one to

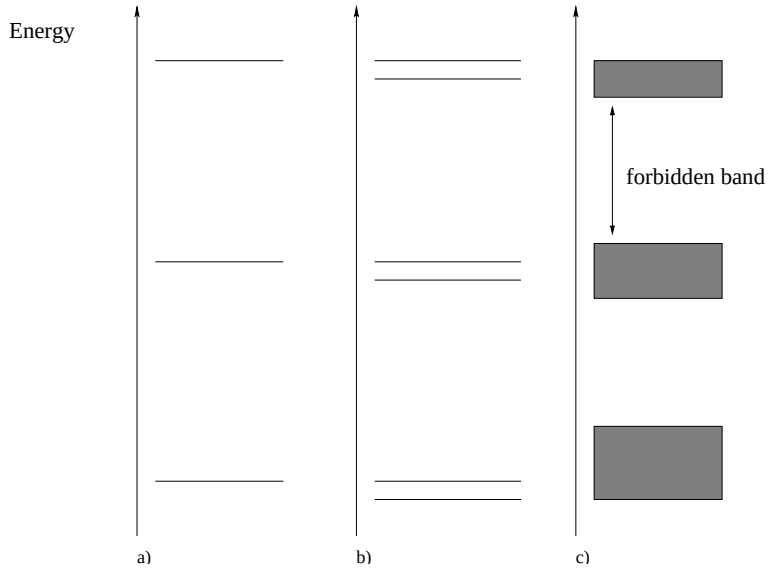

Figure 1.1: Energy levels for an isolated atom(a), two close atoms (b), a crystal (c).

another. Such a material cannot conduct electricity. On the contrary, in the case of the conductors, the valence band and the conduction band overlap so the electrons can go freely from one to another. Such a material has a very low resistivity and easily conduct electricity. Semiconductor materials are a trade-off between insulators and conductors. When the temperature of a semiconductor is low enough, the difference of energy between the valence band and the conduction band is only of the order of one eV. In this case, some electrons of the valence band can get enough energy to jump into the conduction band. One has to notice that these electrons are involved into conduction phenomena but they also leave holes in the electronic structure of the valence bands of the atoms. These holes in the valence bands of the atoms act as positive charges and are also involved in the conduction of the electricity current.

Doped N semiconductors are semiconductors with an excess of free electrons. As an example, it is possible to obtain a doped N semiconductor by introducing Phosphor atoms in a Silicon crystal. Phosphor atom has five electrons in its valence band and Silicon has four. When Phosphor atom establishes its covalent bonding with neighbouring Silicon atoms, one electron remains single in its valence band. This electron can acquire enough energy to jump in the conduction state (see Figure 1.3). Doped P semiconductors are crystals with an excess of holes.

A transistor is a widely used electronic device, in particular in the design of computer processors. Its structure is composed of three layers. The first layer is called the emitter, the second one, thinner, is called the base and the third one, which size is equivalent to the emitter one, is called the collector (see Figure 1.4). Transistors can be of two types, NPN or PNP. Here we will only present the NPN type transistor. In this case, the emitter and the base are composed of a N doped semiconductor material, while the base is composed of a P doped semiconductor.

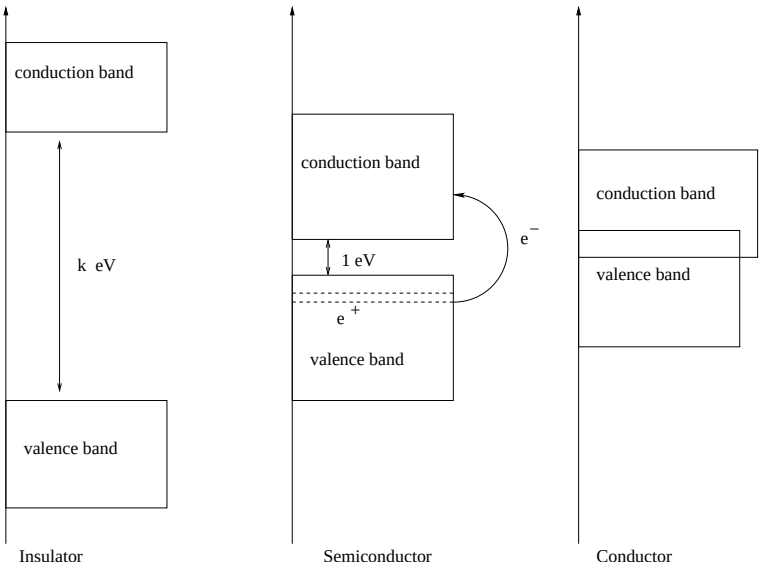

Figure 1.2: Energy bands for insulators, semiconductors and conductors.

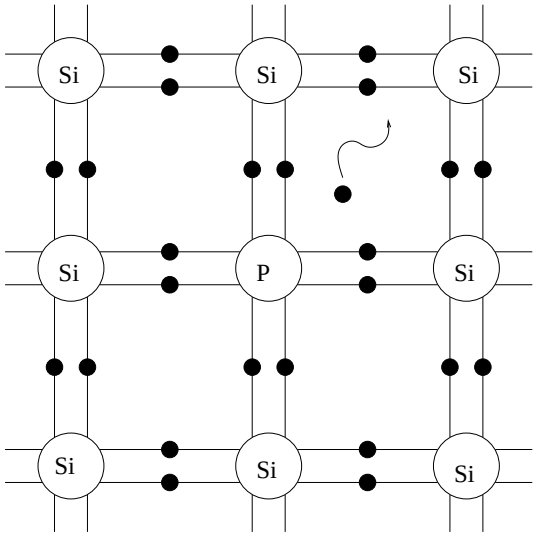

Figure 1.3: Doped N semiconductor: introduction of a Phosphor atom in a Silicon crystal.

We denote by V_e, V_b and V_c the potentials applied to the emitter, the base and the collector respectively. When no tension is applied on the bounds of the transistor, this one is said to be in the equilibrium state. In this state, there is no displacement of electric charges in the transistor. The normal direct regime of an NPN transistor is defined by the following constraints : $V_b - V_e > 0$ and $V_b - V_c < 0$. In this state an electric current crosses the base-emitter dipole. This current is called the base current and denoted by I_B. The free electrons of the emitter diffuse massively to the base. Only a few recombinate in the base. Most of them, driven by their high kinetic energy, cross the base which is thin, and arrive in the collector. Then they are driven outside of the collector by its strong electric field and they amplify the collector current, denoted by I_C. This amplification is called *transistor effect*. The gain in current of a transistor is defined by

$$\beta = \frac{I_C}{I_B}.$$

The aim of this study is to compute the gain in current of an NPN transistor in normal direct amplification mode. Figure 1.5 and Table 1.1 show the other modes of a NPN transistor.

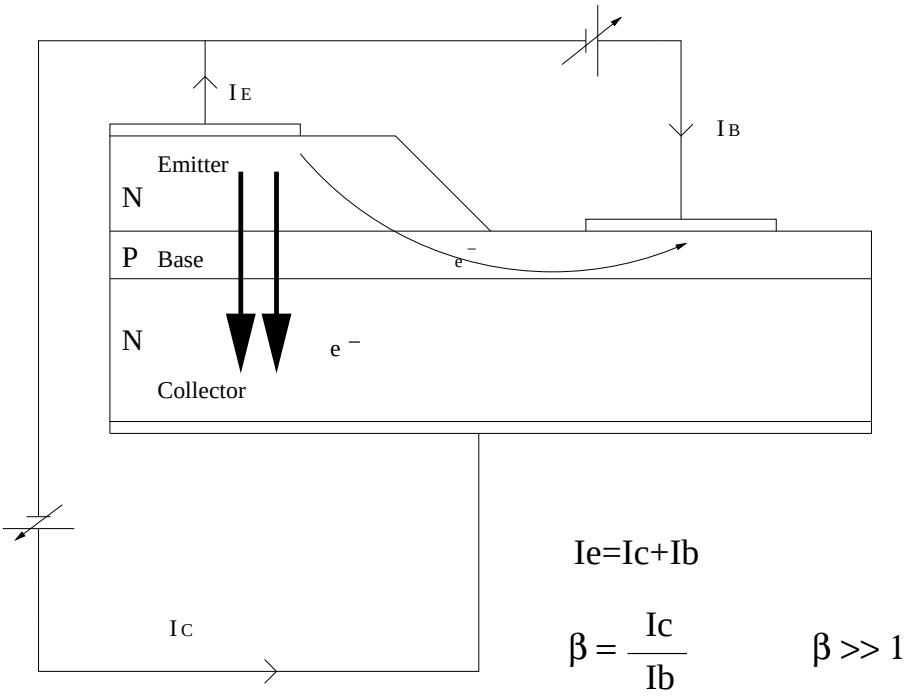

Figure 1.4: NPN transistor.

1.1.2 Mathematical modelling

There are three different levels of modelling for semiconductors. Quantum mechanics is the microscopic level of modelling. It describes the structure of the atoms of the

Regime	Polarizations		
Normal direct	$V_b - V_e > 0$	$V_b - V_c < 0$	
Normal inverse	$V_b - V_e < 0$	$V_b - V_c > 0$	
Saturated direct	$V_b - V_e > 0$	$V_b - V_c > 0$	$V_e < V_c$
Saturated inverse	$V_b - V_e > 0$	$V_b - V_c > 0$	$V_e > V_c$
Blocked	$V_b - V_e < 0$	$V_b - V_c < 0$	

Table 1.1: Regimes of a NPN transistor.

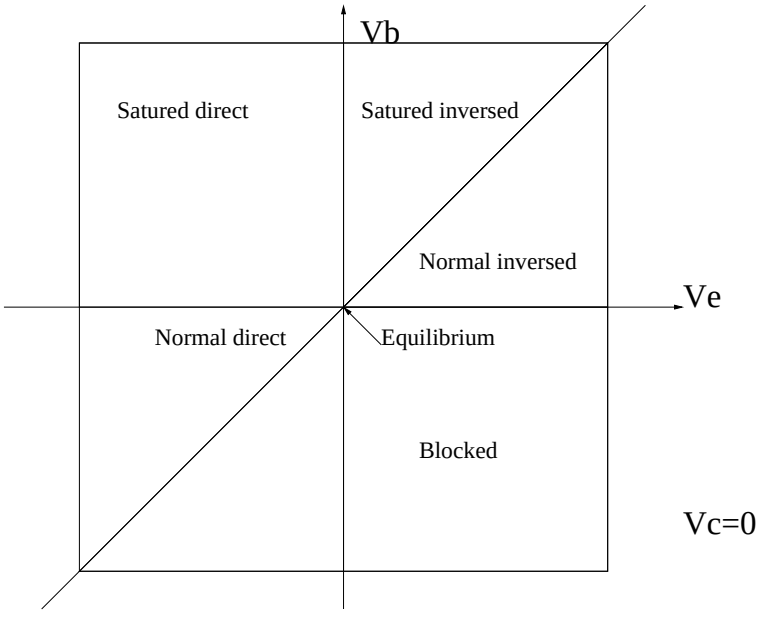

Figure 1.5: Regimes of a NPN transistor.

crystal and the different levels of energy inside them. The second is the statistical model that is based on Boltzmann equation and describes the behaviour of the holes and of the electrons from a statistical point of view. Finally the third is the hydrodynamic model that is the macroscopic level of modelling. It is a determinist model derived from the statistical model. We study here a hydrodynamic model based on drift-diffusion equations for the simulation of an NPN transistor. There are different possibilities for the formulation of the drift-diffusion model depending upon the choice of the unknowns. A first choice for the unknowns is the electrostatic potential ϕ, the concentration in electrons N and the concentration in holes P. But the strong dynamics of N and P make this choice numerically difficult. N and P can be replaced either by the Slotboom variables η_n and η_p or the Fermi levels ϕ_n or ϕ_p. The choice of the Fermi levels fits more for the modelling of heterojunctions structures like the ones we are studying. For more information on this topic we refer to [98].

The proposed model in 2D is the following :

$$\begin{cases} x \in \Omega \subset \mathbb{R}^2, \\ \\ -div(\epsilon(x)\nabla\phi) + q[N(x,\phi,\phi_n) - P(x,\phi,\phi_p) - Dop(x)] = 0, \\ \\ q\partial_t N(x,\phi,\phi_n) - div(q\mu_n N(x,\phi,\phi_n)\nabla\phi_n) + qGR(x,\phi,\phi_n,\phi_p) = 0, \\ \\ j_n = q\mu_n N(x,\phi,\phi_n)\nabla\phi_n, \\ \\ q\partial_t P(x,\phi,\phi_p) - div(q\mu_p P(x,\phi,\phi_p)\nabla\phi_p) - qGR(x,\phi,\phi_n,\phi_p) = 0, \\ \\ j_p = q\mu_p P(x,\phi,\phi_p)\nabla\phi_p \ . \end{cases} \tag{1.1}$$

The model describes at time t several quantities inside a domain Ω which corresponds to the transistor. The electron charge is q. The mobilities of the electrons and the holes are denoted by μ_n and μ_p and are material dependent.

The electrostatic potential (with value in $\mathbb{R}$) is ϕ. The densities of current for the electrons and the holes (with values in $\mathbb{R}^2$) are j_n and j_p. On the same way that the electrostatic potential is related to the electric field ($E = -\nabla\phi$), it is also possible to relate two quantities ϕ_n and ϕ_p to j_n and j_p. These two quantities are also called the Fermi levels associated to the electrons and the holes. They are measured in Volts and so they are homogeneous to the electrostatic potential.

The concentration in electrons and holes are N and P. An example of model for N and P is :

$$\begin{aligned} N(\phi,\phi_n) &= N_c f\left(\frac{\phi+\phi_n-\chi}{V_T}\right), \\ P(\phi,\phi_p) &= N_v f\left(\frac{\chi-\phi-\phi_p-\phi_g}{V_T}\right) \ . \end{aligned}$$

N_c, N_v, χ, ϕ_g are material-dependent physical parameters and V_T is the thermal voltage. The function f is the exponential function for the Boltzman statistics and

is given by

$$f(x) = \frac{2}{\sqrt{\pi}} \int_0^\infty \frac{\sqrt{t}}{1 + e^{t-x}} dt$$

for the Fermi statistics. GR in (1.1) is a function which represents the mechanism of generation-recombination of electrons and holes. This function may have different expressions, depending on the physics taken into account. Dop is a given function only depending on the space variable.

System (1.1) corresponds to the dynamic regime of the transistor. After a certain amount of time, the transistor reaches a steady state, called static regime. On a practical point of view, we only need to know the static regime of a transistor in order to compute its gain in current. That is the reason why we can neglect the derivatives relative to time in (1.1). Then we obtain the system

$$\begin{cases} \text{Find } (\phi, \phi_n, \phi_p) \in \{\Omega \times \mathbb{R}\}^3 \text{ so that} \\ \\ -div(\epsilon \nabla \phi) + q[N(\phi, \phi_n) - P(\phi, \phi_p) - Dop] = 0, \\ \\ -div(q\mu_n N(\phi, \phi_n) \nabla \phi_n) + qGR(\phi, \phi_n, \phi_p) = 0, \\ \\ -div(q\mu_p P(\phi, \phi_p) \nabla \phi_p) - qGR(\phi, \phi_n, \phi_p) = 0, \\ \\ + \text{ Dirichlet-Neumann boundary conditions} \\ \phi = g, \ \phi_n = g_n, \ \phi_p = g_p \text{ on } \Gamma_D, \\ \frac{\partial \phi}{\partial n} = \frac{\partial \phi_n}{\partial n} = \frac{\partial \phi_p}{\partial n} = 0 \text{ on } \Gamma_N = \partial\Omega - \Gamma_D \ . \end{cases} \tag{1.2}$$

In order to simplify the notations, the position variable x is implicit. The first equation is a nonlinear Poisson equation and is dealing with the electrostatic potential. The second one is called continuity equation for the electrons and the third one continuity equation for the holes. One can remark that j_n and j_p have been removed from the model. They can be computed from the solution (ϕ, ϕ_n, ϕ_p) of (1.2). In Section 1.2 and 1.3 we will explain our motivations for reintroducing them in the model. To define the System (1.2) we have added Dirichlet-Neumann boundary conditions to the initial model (1.1).

The main difficulties in solving (1.2) come from the high nonlinearity and large amplitude variation of the functions N and P included in the divergence terms. In Figure 1.6, we display on a logarithmic scale the values of the function N for a test example. We see that the values are varying from $10e^{-11.5}$ up to $10e^{17.11}$. Those large variations will appear later in the matrices associated with the linear systems to be solved during the numerical simulation.

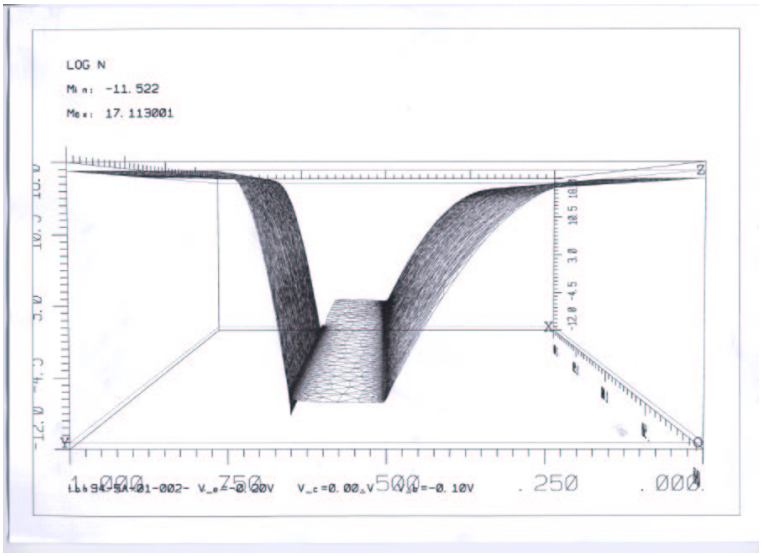

Figure 1.6: Concentration of the electrons in a NPN transistor in amplification mode (courtesy of A. Marrocco from INRIA).

1.2 Discretization and numerical solution of the equilibrium problem

1.2.1 Introduction

In this part, we present on a model problem the main numerical methods that are used to solve the static problem (1.2). This problem has a physical meaning, it is the equilibrium problem (see Figure 1.5) which is the particular case when no tension is applied on the contacts of the device. In this case there is no displacement of electric charges and we have $\phi_n \equiv 0$ and $\phi_p \equiv 0$. The system (1.2) reduces to :

$$\begin{cases} -div(\epsilon \nabla \phi) + F(\phi) = 0 \text{ on } \Omega, \\ \frac{\partial \phi}{\partial n} = g_n \text{ on } \Gamma_n \in \partial\Omega, \\ \phi = g_d \text{ on } \Gamma_d \in \partial\Omega \ . \end{cases} \tag{1.3}$$

We may remark that $\Gamma_d = \partial\Omega - \Gamma_n$ and that

$$F(\phi) = q(N(\phi, \phi_n) - P(\phi, \phi_p) - Dop)$$

with $\phi_n \equiv \phi_p \equiv 0$.

In this presentation we only focus on the numerical solution of the problem. For more theoretical aspects, like existence and unicity of the solution in the appropriate space, we refer to [19].

1.2.2 Mixed finite-element discretization

The mixed formulation of (1.3) is

$$\begin{cases} -div(D) + F(\phi) = 0 \text{ on } \Omega, \\ D = \epsilon \nabla \phi \text{ on } \Omega, \\ \frac{\partial \phi}{\partial n} = g_n \text{ on } \Gamma_n \in \partial\Omega, \\ \phi = g_d \text{ on } \Gamma_d \in \partial\Omega, \end{cases} \tag{1.4}$$

where now the unknowns are D and ϕ. The advantages of the mixed formulation are multiple [71]. On a mathematical point of view, it transforms one second order equation into two first order equations. From the numerical point of view, it computes directly the electric field. With a classical formulation, the electric field is obtained by the derivative of the electrostatic potential. This numerical derivative may be unstable.

Variational formulation

To obtain a variational formulation of (1.4) we introduce the following Sobolev spaces

$$H(div) = \{\vec{w}, \vec{w} \in (L^2(\Omega))^2, div(\vec{w}) \in L^2(\Omega)\}, \tag{1.5}$$

$$V_0 = \{\vec{w}, \vec{w} \in H(div), \vec{w}.n = 0 \text{ on } \Gamma_n\}. \tag{1.6}$$

For sufficiently regular data, we obtain the formulation

$$\begin{cases} \text{Find } D \in H(div) \text{ and } \phi \in L^2(\Omega) \text{ so that} \\ -\int_\Omega v.div(D)dx + \int_\Omega vF(\phi)dx = 0, \\ \int_\Omega [\epsilon]^{-1} D.\vec{w}dx = -\int_\Omega \phi.div(\vec{w})dx + \int_{\Gamma_d} g_d \vec{w}.nd\Gamma, \\ \forall v \in L^2(\Omega),\ \forall \vec{w} \in V_0. \end{cases} \tag{1.7}$$

Discretization

System (1.7) is discretized using mixed triangular finite elements of the Raviart-Thomas type of the lowest order [24]. Let τ_h be a triangulation of the domain Ω. In Figure 1.7 we display a mesh with 5194 triangles and 7923 edges. We see that the mesh is refined at the level of the heterojunctions (which are the interface between the emitter and the base and the interface between the base and the collector) to capture the strong variations along them.

L_h and V_h are two subspaces of $L^2(\Omega)$ and $H(div)$, defined as

$$L_h = \left\{ v_h \in L^2(\Omega) | \forall K \in \tau_h, v_{h|K} = const \right\}$$

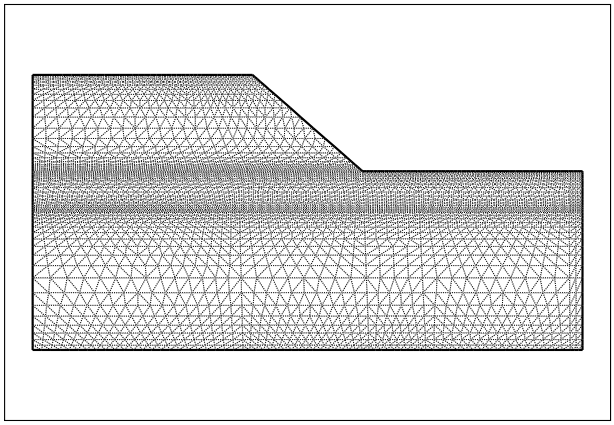

Figure 1.7: Triangular mesh with 5194 elements.

and

$$V_h = \left\{ \vec{w} \in H(div) | \forall K \in \tau_h, w(x,y)|K = \begin{pmatrix} \alpha_K \\ \beta_K \end{pmatrix} + \gamma_K \begin{pmatrix} x \\ y \end{pmatrix} \right\}.$$

If we suppose that the boundary Γ_n associated with the Neumann conditions can be obtained as an union of edges belonging to the triangulation τ_h, then we can define the subspace V_{0h} as

$$V_{0h} = V_0 \cap V_h,$$
$$V_{0h} = \left\{ \vec{w_h} | \vec{w_h} \in V_h,\ \vec{w_h}.n = 0 \text{ on } \Gamma_n \right\}.$$

Then we can give a discrete formulation of the problem (1.7) :

$$\begin{cases} \text{Find } D_h \in V_h \text{ and } \phi_h \in L_h \text{ so that} \\ -\int_\Omega v_h.div(D_h)dx + \int_\Omega v_h F(\phi_h)dx = 0, \\ \int_\Omega [\epsilon]^{-1} D_h.\vec{w_h}dx = -\int_\Omega \phi_h.div(\vec{w_h})dx + \int_{\Gamma_d} g_d \vec{w_h}.nd\Gamma, \\ \forall v_h \in L_h(\Omega),\ \forall \vec{w_h} \in V_{0h}. \end{cases} \tag{1.8}$$

Algebraic Formulation

Let n_t be the number of triangles and n_e the number of edges of the triangulation τ_h. It is possible to build a basis $v = (v_i)_{i=1}^{i=n_t}$ of L_h and a basis $w = (w_i)_{i=1}^{i=n_e}$ of V_{0h}. For a detailed description of these basis we refer to [19].

The solution D_h of (1.8) can be represented in the basis w and we denote its component vector $\overline{D_h}$. Similarly ϕ_h can be expressed in the basis v and we note $\overline{\phi_h}$ its component vector. Problem (1.8) is equivalent to compute $\overline{D_h}$ and $\overline{\phi_h}$. It can be written in matrix form as

$$\begin{pmatrix} M & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} \overline{D_h} \\ \overline{\phi_h} \end{pmatrix} + \begin{pmatrix} 0 \\ F(\overline{\phi_h}) \end{pmatrix} = \begin{pmatrix} b_{\Gamma_n} \\ 0 \end{pmatrix} \tag{1.9}$$

where

$$M = (M_{ij})_{i,j\in[1,n_e]},$$

with

$$M_{ij} = \int_\Omega \epsilon^{-1} \vec{w}_i \vec{w}_j, \text{ with } \vec{w}_i, \vec{w}_j \in w$$

and

$$A = (A_{ij})_{i\in[1,n_e],j\in[1,n_t]}$$

with

$$A_{ij} = \int_\Omega v_i div(\vec{w}_j), \text{ with } v_i \in v \text{ and } w_j \in w.$$

The Neumann boundary conditions are represented by the vector b_{Γ_n}.
The problem to solve is a nonlinear system of equations, that will be tackled by a Newton-Raphson technique.

1.2.3 Newton-Raphson method

In order to simplify the notations we identify the functions D_h and ϕ_h with their component vectors $\overline{D_h}$ and $\overline{\phi_h}$ in the basis w and v. Equation (1.9) can be solved by finding a zero value of the function

$$\left\{ \begin{array}{ccccc} \mathcal{F} & : & \mathbb{R}^{n_t+n_e} & \longmapsto & \mathbb{R}^{n_t+n_e} \\ & & \begin{pmatrix} D_h \\ \phi_h \end{pmatrix} & \longmapsto & \begin{pmatrix} MD_h + A\phi_h - b_{\Gamma_n} \\ A^T D_h + F(\phi_h) \end{pmatrix}. \end{array} \right. \tag{1.10}$$

The algorithm used to compute where $\mathcal{F}$ vanishes is a Newton-Raphson method that is described by Algorithm 1.

At step 1 of Algorithm 1, a linear system associated to the Jacobian matrix $J_\mathcal{F}$ has to be solved. In our case the block structure of $J_\mathcal{F}$ is

$$J_\mathcal{F} = \begin{pmatrix} M & A \\ A^T & \frac{\partial F}{\partial \phi}(\phi_h) \end{pmatrix} = \begin{pmatrix} M & A \\ A^T & C(\phi_h) \end{pmatrix}. \tag{1.11}$$

So the linear system to solve at each step is

$$\begin{pmatrix} M & A \\ A^T & C(\phi_h^l) \end{pmatrix} \begin{pmatrix} \delta D_h^{l+1} \\ \delta \phi_h^{l+1} \end{pmatrix} = \begin{pmatrix} D_h^l \\ \phi_h^l \end{pmatrix}. \tag{1.12}$$

This system has two remarkable properties. The first one is that it is symmetric positive definite and the second one is that the matrix $C(\phi_h^l)$ is diagonal [19]. The

Find an initial vector x^0
while non convergence **do**
Step 1. Solve the linear system

$$J_{\mathcal{F}}\ \delta x^{l+1} = -\mathcal{F}(x^l),$$

$J_{\mathcal{F}}$ is the jacobian matrix associated to the operator $\mathcal{F}$.
Step 2. : $x^{l+1} = x^l + \delta x^{l+1}$.
Step 3. : Test convergence with

$$\frac{\|x^{l+1} - x^l\|}{\|x^{l+1}\|} < \epsilon_1.$$

end while

Algorithm 1: Newton-Raphson algorithm to solve $\mathcal{F}x = 0$.

1 : Solve the linear systems on the fluxes

$$(M - AC(\phi_h^l)^{-1}A^T)\delta D_h^{l+1} = D_h^l - AC(\phi_h^l)^{-1}\phi_h^l.$$

2 : Compute the potentials $\delta\phi^{l+1}$ from the fluxes δD_h^{l+1} with the formula

$$\delta\phi_h^{l+1} = C(\phi_h^l)^{-1}(\phi_h^l - A^T\delta D_h^{l+1}).$$

Algorithm 2: Reduced linear system.

second property allows us to easily reduce the system (1.12) to a system only defined for the fluxes unknowns (see Algorithm 2).

The convergence of Algorithm 1 strongly depends upon the choice of the initial guess. If this vector is not close enough from the solution then the convergence can be very slow or worse, impossible to obtain. To improve the convergence of the Newton-Raphson method, we associate with the problem (1.4) an artificial evolution problem.

1.2.4 Artificial evolution problem

Problem (1.4) is a saddle-point problem. Augmented Lagrangian formulations can be associated to it [64]. One of them can be described as the computation of the steady state of the artificial time dependent problem

$$\begin{cases} S_1(x)\frac{\partial \phi}{\partial t} - div(D) + F(\phi) = 0 & \text{in } \Omega, \\ D = \epsilon \nabla \phi & \text{in } \Omega, \\ \frac{\partial \phi}{\partial n} = g_n & \text{on } \Gamma_n \in \partial\Omega, \\ \phi = g_d & \text{on } \Gamma_d \in \partial\Omega, \\ \phi(x,0) = \phi_0(x) & \text{in } \Omega. \end{cases} \tag{1.13}$$

S_1 is a linear, diagonal, positive definite and bounded operator introduced to obtain a better conditioning for the induced linear systems [24].

Problem (1.13) is then discretized in time by an Euler implicit scheme, described by Algorithm 3.

Each system (1.14) is discretized on the same mesh and solved by a Newton-Raphson iterative method. The Newton-Raphson method converges if the initial solution (ϕ^n, D^n) is close enough to the solution (ϕ^{n+1}, D^{n+1}). The main difficulty is the choice of good local time steps. On one hand, if Δt is too small a great number of Euler iterations will be required. On the other hand, if Δt is chosen too large, the number of iterations of Newton-Raphson will grow up at each Euler iteration. The selected strategy for the choice of Δt is discussed in [19].

1.3 Numerical solution of the static problem

We present in this section the algorithm used to obtain the solution of the static problem

Initialization : Choose an initial value for the potentials and the fluxes, ϕ^0 and D^0.
Time iteration loop : ϕ^n and D^n known, compute (ϕ^{n+1}, D^{n+1}) solutions of the problem

$$\begin{cases} S_1(x)\frac{\phi^{n+1}-\phi^n}{\Delta t} - div(D^{n+1}) + F(\phi^{n+1}) = 0 \text{ in } \Omega, \\ D^{n+1} = \epsilon\nabla\phi^{n+1}, \\ \frac{\partial\phi}{\partial n} = g_n \text{ on } \Gamma_n \in \partial\Omega, \\ \phi = g_d \text{ on } \Gamma_d \in \partial\Omega, \\ \phi(x,0) = \phi_0(x) \text{ in } \Omega. \end{cases} \tag{1.14}$$

(Remark : we use here local time steps, Δt depends on x.)
Stopping criterion : we consider here the criterion

$$max\left(\frac{\| \phi^{n+1} - \phi^n \|_1}{\| \phi^{n+1} \|_1}, \frac{\| D^{n+1} - D^n \|_1}{\| D^{n+1} \|_1}\right) < \epsilon.$$

Algorithm 3: Time discretization for the equilibrium problem.

$$\left\{\begin{array}{l} \text{Find } (\phi, \phi_n, \phi_p) \in \{\Omega \times \mathbb{R}\}^3 \text{ so that} \\ \\ -div(\epsilon\nabla\phi) + q[N(\phi,\phi_n) - P(\phi,\phi_p) - Dop] = 0, \\ \\ -div(q\mu_n N(\phi,\phi_n)\nabla\phi_n) + qGR(\phi,\phi_n,\phi_p) = 0, \\ \\ -div(q\mu_p P(\phi,\phi_p)\nabla\phi_p) - qGR(\phi,\phi_n,\phi_p) = 0, \\ \\ + \text{ Dirichlet-Neumann boundary conditions} \\ \phi = g,\ \phi_n = g_n,\ \phi_p = g_p \text{ on } \Gamma_D, \\ \frac{\partial\phi}{\partial n} = \frac{\partial\phi_n}{\partial n} = \frac{\partial\phi_p}{\partial n} = 0 \text{ on } \Gamma_N = \partial\Omega - \Gamma_D. \end{array}\right. \tag{1.15}$$

The mixed formulation of the static problem introduces the densities of current j_n and j_p as dual unknowns and can be written

$$\left\{\begin{array}{l} \text{Find } (\phi, D, \phi_n, j_n, \phi_p, j_p) \text{ so that} \\ \\ -div(D) + q[N(\phi,\phi_n) - P(\phi,\phi_p) - Dop] = 0, \\ \\ D = \epsilon\nabla\phi, \\ \\ -div(j_n) + qGR(\phi,\phi_n,\phi_p) = 0, \\ \\ j_n = q\mu_n N(\phi,\phi_n)\nabla\phi_n, \\ \\ -div(j_p) - qGR(\phi,\phi_n,\phi_p) = 0, \\ \\ j_p = q\mu_p P(\phi,\phi_p)\nabla\phi_p, \\ \\ +\text{Boundary conditions.} \end{array}\right. \tag{1.16}$$

Like for the equilibrium problem, we use here a stabilization technique that consists in considering the solution as the steady state solution of a time dependent problem. The time dependent problem is defined as :

$$\left\{\begin{array}{l} \text{Find } (\phi, D, \phi_n, j_n, \phi_p, j_p) \text{ so that} \\ S_1(x)\frac{\partial \phi}{\partial t} - div(D) + q[N(\phi,\phi_n) - P(\phi,\phi_p) - Dop] = 0, \\ D = \epsilon \nabla \phi, \\ S_2(x)\frac{\partial \phi_n}{\partial t} - div(j_n) + qGR(\phi,\phi_n,\phi_p) = 0, \\ j_n = q\mu_n N(\phi,\phi_n)\nabla\phi_n, \\ S_3(x)\frac{\partial \phi_p}{\partial t} - div(j_p) - qGR(\phi,\phi_n,\phi_p) = 0, \\ j_p = q\mu_p P(\phi,\phi_p)\nabla\phi_p, \\ +\text{Boundary conditions}, \\ +\text{Initial conditions}. \end{array}\right. \tag{1.17}$$

1.3.1 Time discretization by an implicit nonlinear scheme

The algorithm to solve (1.17) is based on one hand on a decoupling of the Poisson equation and of the two continuity equations and, on the other hand on a time discretization by an Euler implicit scheme. The numerical technique is described by Algorithm 4.

1.3.2 Discretization and numerical solution of the continuity equation for the electrons

The Problem (1.17) is solved using Algorithm 4. The latter algorithm is an iterative evolution process which requires the solution of the problems (1.18), (1.19) and (1.20) at each time step. Equation (1.18) is a nonlinear Poisson equation. Therefore it can be solved using the algorithms presented in Section 1.2. Equations (1.19) and (1.20) are similar and we only detail the solution of Equation (1.19). Equation (1.19) can be written

$$\left\{\begin{array}{l} -div(j_n^{\ell+1}) + G_n(\phi_n^{\ell+1}) = 0, \\ j_n^{\ell+1} = q\mu_n N(\phi^{\ell+1},\phi_n^{\ell+1})\nabla\phi_n^{\ell+1}, \\ + \text{ Boundary conditions}, \end{array}\right. \tag{1.21}$$

where $G_n(\phi_n^{\ell+1}) = S_2\frac{\phi_n^{\ell+1}-\phi_n^{\ell}}{\Delta t} + qGR(\phi^{\ell+1},\phi_n^{\ell+1},\phi_p^{\ell})$. This equation is discretized by Raviart-Thomas finite elements on the same mesh than the one used to solve the

Step 1. $\ell = 0$. This step consists in choosing an initial guess $(\phi^0, D^0, \phi_n^0, j_n^0, \phi_p^0, j_p^0)$ to start the iterative scheme.
Step 2. This step consists in solving the nonlinear Poisson equation

$$\begin{cases} S_1(x)\frac{\phi^{\ell+1}-\phi^{\ell}}{\Delta t} - div(D^{\ell+1}) + f(\phi^{\ell+1}, \phi_n^{\ell}, \phi_p^{\ell}) = 0, \\ D^{\ell+1} = \epsilon\nabla\phi^{\ell+1}, \\ \phi^{\ell+1} = g_1 \text{ on } \partial\Omega_D, \\ D^{\ell+1}.n = 0 \text{ on } \partial\Omega_N, \end{cases} \tag{1.18}$$

with $\phi^{\ell+1}$ and $D^{\ell+1}$ as unknowns and ϕ^{ℓ}, D^{ℓ}, ϕ_n^{ℓ}, j_n^{ℓ}, ϕ_p^{ℓ}, j_p^{ℓ} as data.
Step 3. This step consists in solving the continuity equation for the electrons

$$\begin{cases} S_2(x)\frac{\phi_n^{\ell+1}-\phi_n^{\ell}}{\Delta t} - div(j_n^{\ell+1}) + qGR(\phi^{\ell+1}, \phi_n^{\ell+1}, \phi_p^{\ell}) = 0, \\ j_n^{\ell+1} = q\mu_n N(\phi^{\ell+1}, \phi_n^{\ell+1})\nabla\phi_n^{\ell+1}, \\ \phi_n^{\ell+1} = g_2 \text{ on } \partial\Omega_D, \\ j_n^{\ell+1}.n = 0 \text{ on } \partial\Omega_N, \end{cases} \tag{1.19}$$

with $\phi_n^{\ell+1}$ and $D_n^{\ell+1}$ as unknowns and $\phi^{\ell+1}$, $D^{\ell+1}$, ϕ_n^{ℓ}, j_n^{ℓ}, ϕ_p^{ℓ}, j_p^{ℓ} as data.
Step 4. This step consists in solving the continuity equation for the holes

$$\begin{cases} S_3(x)\frac{\phi_p^{\ell+1}-\phi_p^{\ell}}{\Delta t} - div(j_p^{\ell+1}) - qGR(\phi^{\ell+1}, \phi_n^{\ell+1}, \phi_p^{\ell+1}) = 0, \\ j_p^{\ell+1} = q\mu_p P(\phi^{\ell+1}, \phi_p^{\ell+1})\nabla\phi_p^{\ell+1}, \\ \phi_p^{\ell+1} = g_3 \text{ on } \partial\Omega_D, \\ j_p^{\ell+1}.n = 0 \text{ on } \partial\Omega_N. \end{cases} \tag{1.20}$$

with $\phi_p^{\ell+1}$ and $D_p^{\ell+1}$ as unknowns and $\phi^{\ell+1}$, $D^{\ell+1}$, $\phi_n^{\ell+1}$, $j_n^{\ell+1}$, ϕ_p^{ℓ}, j_p^{ℓ} as data.
Step 5. Stopping criterion.
if $max\begin{pmatrix} \frac{\|\phi^{\ell+1}-\phi^{\ell}\|_1}{\|\phi^{\ell+1}\|_1}, & \frac{\|D^{\ell+1}-D^{\ell}\|_1}{\|D^{\ell+1}\|_1}, \\ \frac{\|\phi_n^{\ell+1}-\phi_n^{\ell}\|_1}{\|\phi_n^{\ell+1}\|_1}, & \frac{\|j_n^{\ell+1}-j_n^{\ell}\|_1}{\|j_n^{\ell+1}\|_1}, \\ \frac{\|\phi_p^{\ell+1}-\phi_p^{\ell}\|_1}{\|\phi_p^{\ell+1}\|_1}, & \frac{\|j_p^{\ell+1}-j_p^{\ell}\|_1}{\|j_p^{\ell+1}\|_1}) \end{pmatrix} < \epsilon$ **then**
exit,
else
$\ell \leftarrow \ell + 1$,
go to **Step 2.**
end if

Algorithm 4: Decoupling by relaxation and time discretization by Euler semi-implicit schemes.

nonlinear Poisson equation (1.18). We obtain the following nonlinear system of equations

$$\begin{pmatrix} M(\phi_n^{\ell+1}) & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} j_n^{\ell+1} \\ \phi_n^{\ell+1} \end{pmatrix} + \begin{pmatrix} 0 \\ G_n(\phi_n^{\ell+1}) \end{pmatrix} = \begin{pmatrix} b_{\Gamma_n} \\ 0 \end{pmatrix}. \tag{1.22}$$

The main difference between this system and System (1.9) obtained after the discretization of Equation (1.14) is the nonlinearity of the mass operator M with respect to the unknown $\phi_n^{\ell+1}$. In order to simplify the notations we remove from (1.22) the time suffix ℓ. Then we have to solve

$$\begin{pmatrix} M(\phi_n) & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} j_n \\ \phi_n \end{pmatrix} + \begin{pmatrix} 0 \\ G_n(\phi_n) \end{pmatrix} = \begin{pmatrix} b_{\Gamma_n} \\ 0 \end{pmatrix}. \tag{1.23}$$

Equation (1.23) can be solved by finding a zero value of the function

$$\left\{ \begin{array}{rcl} \mathcal{F}_n : \mathbb{R}^{n_t+n_e} & \longmapsto & \mathbb{R}^{n_t+n_e} \\ \begin{pmatrix} j_n \\ \phi_n \end{pmatrix} & \longmapsto & \begin{pmatrix} M(\phi_n)j_n + A\phi_n - b_{\Gamma_n} \\ \\ A^T j_n + G_n(\phi_n) \end{pmatrix}. \end{array} \right. \tag{1.24}$$

A Newton-Raphson method (see Algorithm 1) is used to compute the zero value of $\mathcal{F}_n$. The Jacobian matrix associated with the function $\mathcal{F}_n$ is

$$J_{\mathcal{F}_n} = \begin{pmatrix} M(\phi_n) & A + \frac{\partial(M(\phi_n)j_n)}{\partial\phi_n} \\ A^T & \frac{\partial Gn}{\partial\phi_n}(\phi_n) \end{pmatrix} = \begin{pmatrix} M(\phi_n) & A + H(\phi_n, j_n) \\ A^T & C_n(\phi_n) \end{pmatrix}. \tag{1.25}$$

While the Jacobian matrix (1.11) obtained from the discretization of the Poisson equation was SPD, the Jacobian matrix (1.25) is no longer symmetric due to the introduction of the term H. H is arising from the nonlinearity of M with respect to the unknown ϕ_n. Therefore the problem to solve at step k of the Newton-Raphson method is

$$\begin{pmatrix} M(\phi_n^k) & A + H(\phi_n^k, j_n^k) \\ A^T & C_n(\phi_n^k) \end{pmatrix} \begin{pmatrix} \delta j_n^{k+1} \\ \delta\phi_n^{k+1} \end{pmatrix} = \begin{pmatrix} j_n^k \\ \phi_n^k \end{pmatrix}. \tag{1.26}$$

Due to the high nonlinearity of the concentration of electrons N, these linear systems are ill conditioned. Like the submatrix C of the matrix (1.11), the submatrix C_n of (1.25) is diagonal. Then we can eliminate the potentials and solve a system reduced to the fluxes. The values of the potentials are then simply recovered by substitution (see Algorithm 2).

1.3.3 Choice of an initial solution

To compute the initial solution (ϕ^0, D^0, ϕ_n^0, j_n^0, ϕ_p^0, j_p^0) of Algorithm 4, several possibilities exist. The simplest one would be to consider all these functions as null functions. A more appropriate one would be to solve first the equilibrium problem

to obtain initial values for ϕ^0 and D^0. More generally, an idea is to use the solution of previous experiments in order to build initial values by interpolation.

A method has been implemented to improve an initial solution for the two continuity equations. Once an initial guess (ϕ_n^0, j_n^0, ϕ_p^0, j_p^0) has been chosen, an improved initial guess (ϕ_n^{0+}, j_n^{0+}, ϕ_p^{0+}, j_p^{0+}) can be computed by solving the two linear systems

$$\begin{pmatrix} M(\phi_n^0) & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} j_n^{0+} \\ \phi_n^{0+} \end{pmatrix} + \begin{pmatrix} 0 \\ G_n(\phi_n^0) \end{pmatrix} = \begin{pmatrix} b_{\Gamma_n} \\ 0 \end{pmatrix}, \tag{1.27}$$

and

$$\begin{pmatrix} M(\phi_p^0) & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} j_p^{0+} \\ \phi_p^{0+} \end{pmatrix} + \begin{pmatrix} 0 \\ G_n(\phi_p^0) \end{pmatrix} = \begin{pmatrix} b_{\Gamma_n} \\ 0 \end{pmatrix}. \tag{1.28}$$

These two linear systems are SPD and generally ill-conditioned. It is shown in [19] that this procedure reduces significantly the number of needed time steps.

Continuation technique

If the gap between the tensions applied to the bounds of the transistor is too large then the convergence of the Euler scheme may become difficult to obtain. An incremental tension strategy is used to overcome this difficulty. The simulation is separated in less difficult intermediate simulations. For example, instead of computing the static regime with a tension of 0.4 Volts on the basis in one simulation, a first simulation is performed to obtain the static regime with a tension of 0.2 Volts on the basis and the solution of this simulation is then used as a starting point for the 0.4 Volts simulation.

1.4 Conclusion

The general method used to compute the steady state of a transistor in amplification mode can be summarized in the following way. First the equilibrium problem is solved. This is the case where no tension is applied on the contacts of the device. In order to solve this problem, an artificial evolution process is introduced to make the solution of the original nonlinear problem easier. It is discretized in time by an Euler implicit scheme. At each time step, a nonlinear system of equations has to be solved. This system is solved by a Newton-Raphson iterative method. Each step of the Newton-Raphson method requires the solution of a SPD linear system. Once the equilibrium problem has been solved, a tension increment V_1 applied on the contacts of the device is chosen. The initial solution for the static problem is defined by the solution of the equilibrium problem, involving ϕ and D. Concerning the continuity equations, the initial solution is given by the solutions of the two linearized problems. This solution is used to initialize an artificial evolution process discretized in time

by an Euler scheme where the Poisson equation and the two continuity equations are decoupled. At each time step, three systems of nonlinear equations have to be solved. Each system is solved by a Newton-Raphson iterative method. Each step of the Newton-Raphson method requires the solution of a linear system. This system is SPD for the Poisson equation and unsymmetric for the continuity equations. Once the solution of the static problem has been obtained, a new tension increment V_2 and a new static problem with $V_1 + V_2$ as boundary conditions are defined. The initial solution of the Euler scheme is the previous solution computed with V_1 as boundary conditions. This initial solution is improved by solving the associated linearized problems. And so on, for tension increments V_3, V_4, ..., V_n until one has obtained the solutions for all the desired tensions. Figure 1.8 represents this procedure. A sequential Fortran implementation of this algorithm has been developed at INRIA. In Chapter 2 we address the main issues concerning how we proceed to parallelize this sequential code.

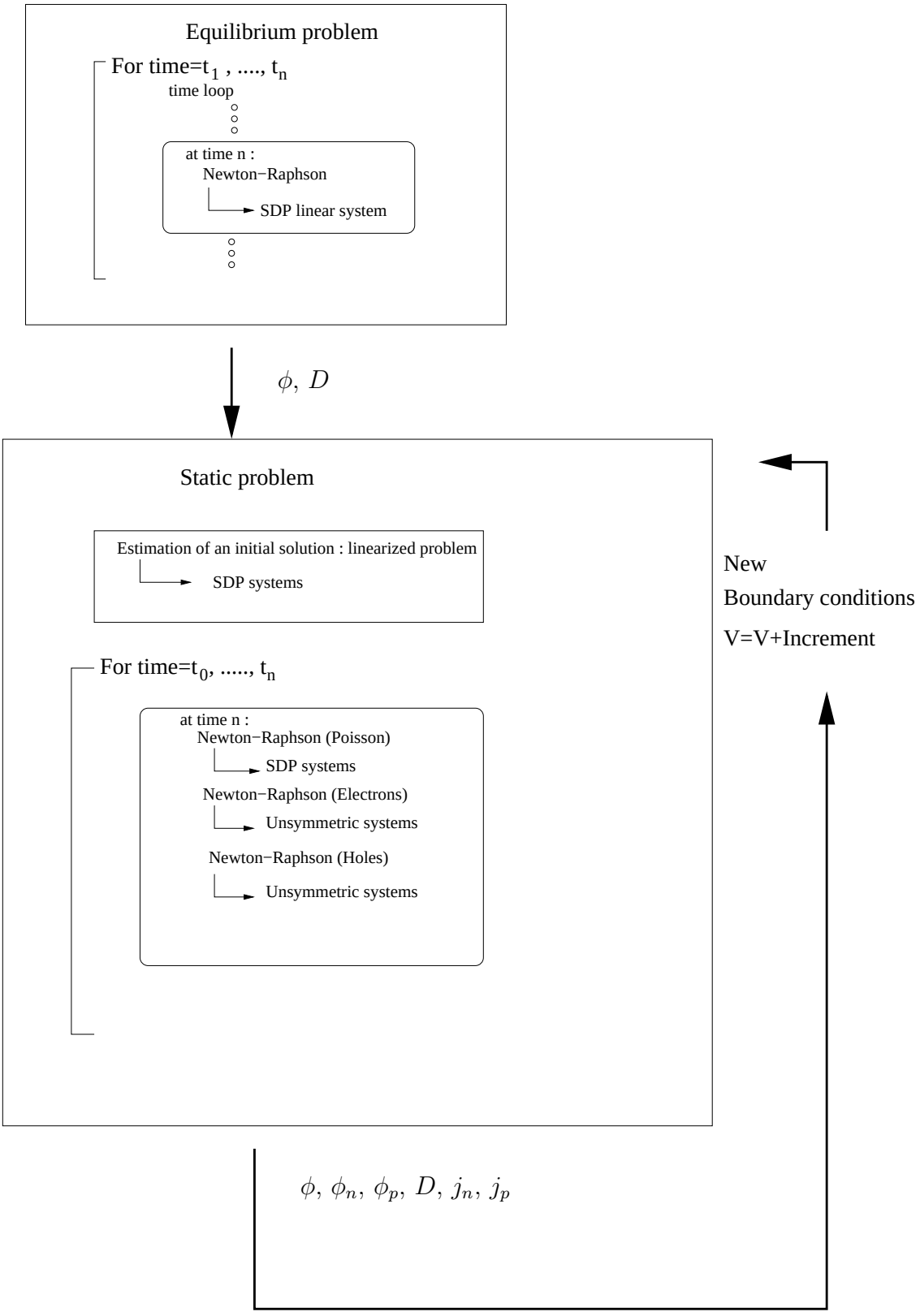

Figure 1.8: Summary of the algorithm used for the solution of the static problem.

Chapter 2

Parallelization of the finite element code

In this chapter we present the main algorithmic and software tools that we have used to parallelize the sequential code developed at INRIA. In Sections 2.1 and 2.2 we introduce the general context and describe why the main difficulty is in the parallel solution of the linear systems. In Section 2.3, we present the multifrontal method which is an algorithm for the factorization of sparse matrices. We also present the software MUMPS which is an implementation of the multifrontal algorithm for distributed memory platforms. In Section 2.4, we discuss domain decomposition methods which are methods designed to solve in parallel linear systems arising from the discretization of PDEs. We focus on substructuring algorithms and present both iterative and direct substructuring approaches.

2.1 The parallel computing framework

2.1.1 Brief overview of the parallel computing platforms

In spite of the constant growth of processor power, some physical limitations, like the speed of light, will prevent any uniprocessor computer satisfying the huge computational needs required by many of the current and future complex numerical simulations. During the last decades, many ideas and tricks have been implemented to design scientific computers with the goal of increasing the number of floating point operations per clock cycle. Such techniques have first been implemented at the processor level by pipelining the arithmetic operations, that is a very fine grain parallelism. Later a coarser grain parallelism has been exploited by putting several processors within a single computer: this was the birth of what is called today a parallel computer. In the first generation of parallel scientific computers, all the processors physically shared the central memory and accessed it through a sophisticated memory path (see Figure 2.1 for a macroscopic view of a shared memory computer). Originally called shared memory parallel computers, their main architectural weakness was the limited number of processors that can be plugged on the

memory path. To remove this physical bottleneck, the main memory has been cut into pieces, each piece attached to one processor to build a node, and all the nodes have been connected through sophisticated networks. Those latter computers were first named distributed memory platforms.

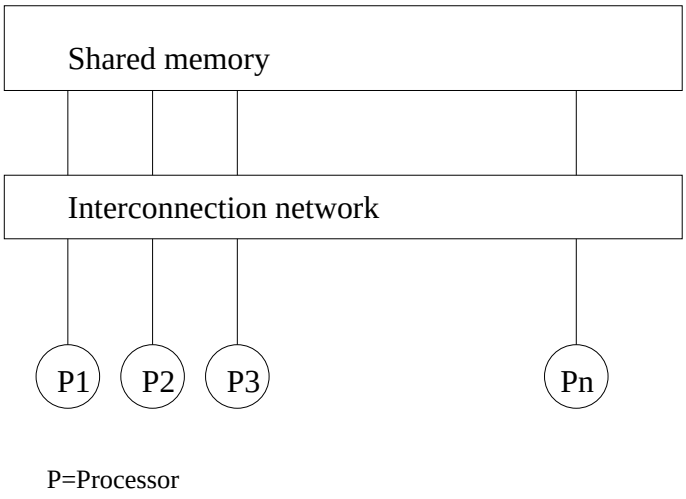

Figure 2.1: Shared memory architecture.

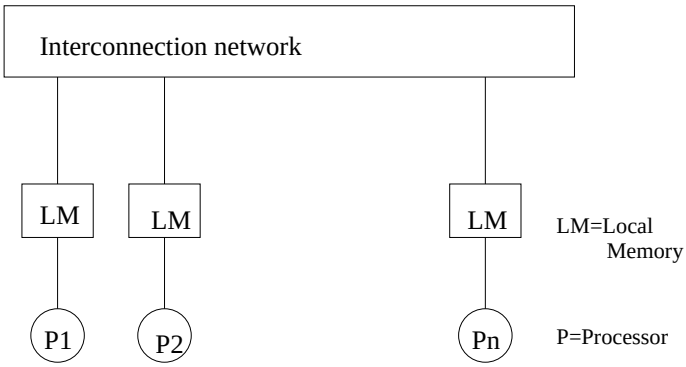

Figure 2.2: Distributed memory architectures.

This primal classification of parallel computer architectures into two main families depending on the key defined by the physical management of the memory is incomplete. Another major criterion, and more relevant for the design of parallel algorithms, is the way the memory is logically viewed and possibly shared by all the processors of the parallel machine. That is, either all the concurrent processors shared a global address space or each of them has its own private and disjoint address space. This logical point of view is not necessary correlated to the way the memory is physically implemented. In the 90's, computers like the BBN Butterfly or the Kendall Square were the first computers to have physically distributed but globally addressable memory. They were first called distributed virtual shared machines, and today such platforms are commonly called Non Uniform Memory Architecture

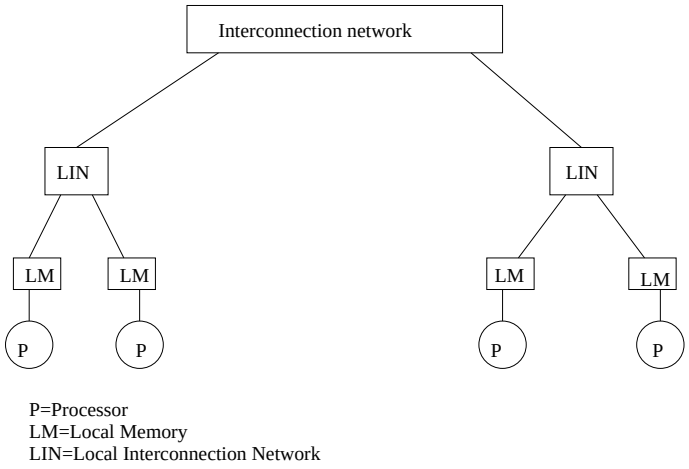

Figure 2.3: Cluster of SMP.

(NUMA). Indeed, the time to access a datum in memory depends on its location, either in the memory physically associated with the processor or a remote memory associated with another processor. By contrast, the old shared memory platforms are frequently called today Symmetric Multiprocessors, since all the processors have the same "privilege" to access any memory location. Finally because shared and distributed memory architectures exhibit some advantages, they have been combined to give birth to clusters of symmetric multiprocessors (see Figure 2.3 for a macroscopic view of a cluster of SMP). Today the most powerful installed computers are based on this hybrid architecture.

2.1.2 The parallel programming paradigms

The logical view of the memory is of prime importance as it deeply influences the programming paradigm to be selected for implementing a parallel algorithm. Today, a quasi-standard exists for global address space computers that is OpenMP [34, 88]. OpenMP is a set of compilation directives and a run time library that allows the user to parallelize an existing code incrementally and smoothly. In that case the concurrent threads communicate through the memory, by writing and reading in the shared address space. For disjoint address space computers, message passing is the tool of choice. For such platforms two main libraries are still applicable, these are the standard MPI (Message Passing Interface) [70], for intensive homogeneous computing, and PVM (Parallel Virtual Machine) [57] for heterogeneous computing. In this latter case homogeneous should be understood both for the target platform that might be a network of heterogeneous computers but more significantly for the nature of the parallel application. For instance for multi-physics simulations when people are loosely coupling existing codes. While MPI-1 was pure SPMD (Single Program Multiple Data) and then not appropriate for such heterogeneous parallel computation, MPI-2 [69] addresses MPMD (Multiple Program Multiple Data) applications but its complete and stable implementation is not yet available on all

computers. This might lead us to think that for a few more years both MPI and PVM will coexist, each devoted to its specific application area. It should be mentioned that message passing can be used on global address space platforms. Finally and similarly to what has been done on clusters of SMPs at the architecture level by mixing global address space within a group of processors, and disjoint address space between the groups, MPI and OpenMP can be mixed to express and easily manage two levels of parallelism. This combination of programming paradigms is likely to become a promising and natural alternative for developing large applications on the huge computers in the future [41, 60].

2.2 Parallelization of a PDE solver in a distributed environment

2.2.1 Mesh partitioning

The first question to address when implementing an algorithm using message passing, is the splitting of the data associated with the problem to be solved and their mapping to the different processors. In that situation the two governing constraints are:

1. load balancing, that is, decompose the data in such a way that the required computing effort on each subset will be equal. Good load balancing will minimize the idle time the processors spend in synchronizations.

2. minimizing the number, while maximizing the size, of messages to be exchanged for implementing the parallel algorithm.

For the parallel solution of PDEs the two above conditions often translate into a decomposition of the underlying mesh into sub-meshes, hopefully connected, and a static mapping of each of those subdomains onto each processor of the target computer. This pre-processing phase is performed using a graph/mesh partitioner tool like the public domain METIS [75], CHACO [72] or Scotch [90] or an in-house partitioner, which in our case is the one integrated in MODULEF. MODULEF is a Fortran library for finite elements developed at INRIA [79]. Finding the optimal partition to achieve optimal load-balancing and minimal interface constraints is an NP-hard problem. In that respect all these tools implement heuristics that exploit either geometric information or only topological information through the adjacency graph [59]. Finally, we mention that for finite-difference or finite-volume discretization the set of vertices of the mesh is partitioned, while the set of elements is usually partitioned for finite-element discretization (i.e. the dual graph). In Figure 2.4, we display a partitioning into 8 subdomains of a mesh of 5194 triangles, computed with the MODULEF mesh partitioner tool.

We should mention yhat in all our implementations, the divided problem is mapped directly on the parallel machine. That means that we associate one and

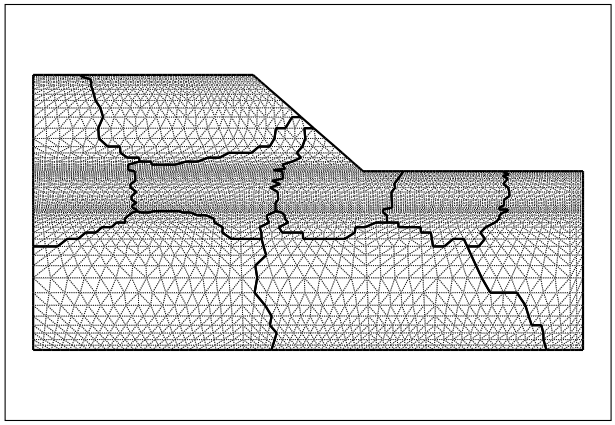

Figure 2.4: Partitioning into 8 subdomains of a mesh of 5194 triangles.

only one subdomain to one physical processor. This does not have to be the case and some domain decomposition codes allow multiple subdivisions on individual processors.

2.2.2 Parallelization of the Euler and the Newton-Raphson procedures

Once the distribution of the mesh is known, the operators are discretized locally on each subdomain in parallel. Then the numerical solution consists of an Euler scheme calling a Newton method which calls itself a linear system solver at each step. The update of the Euler and the Newton schemes are vector operations. Dot products are also required to compute residual norms. In that framework, no communication is needed to sum two vectors. For a dot product one global reduction is needed. So once the linear solver has been parallelized, the parallelization of the Euler and the Newton methods is straightforward.

2.2.3 Parallelization of the linear system solution

The most important part of the computations is spent solving linear systems. Typically, more than 90% of the computational time is spent solving linear systems when the sequential code is used. How to parallelize these linear systems in a distributed environment is the main topic of this thesis. We investigate both distributed direct methods and distributed iterative methods for the semiconductor application. Before presenting in detail these methods we recall some properties of the linear systems to solve.

Some properties of the linear systems

Let $Ax = b$ be one of the linear systems to solve

1. Size : A is a square matrix of dimension n, where n is the number of edges in the mesh; the entries of A are reals.

2. Symmetry : the matrix A is either symmetric positive definite (SPD matrix) if it is arising from the discretization of the Poisson equation or unsymmetric if is arising from the discretization of one of the two continuity equations. In the numerical unsymmetric case, the pattern remains symmetric. That means that $a_{ij} \neq 0$ if and only if $a_{ji} \neq 0$.

3. Sparsity : the unknowns for the fluxes are associated with the edges in the triangulation and we have the property that $a_{ij} \neq 0$ if and only if i and j are two edges belonging to a common triangle. One edge can be connected directly to a maximum of 4 other edges as it is shown in Figure 2.5. So the matrix A has at most 5 nonzero entries per row. Moreover, this sparsity pattern remains constant during the nonlinear iterations as it is based upon the mesh topology which remains constant.

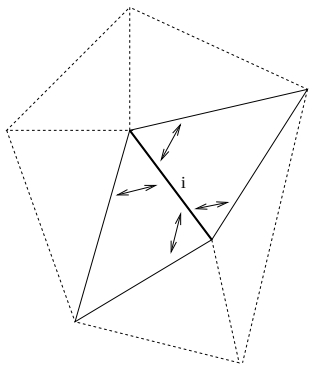

Figure 2.5: The edge i is connected directly to only the 4 neighbouring edges.

2.3 Parallel direct methods for sparse matrices

2.3.1 Introduction

When performing a LU (or LL^T) factorization in order to solve the linear system $Ax = b$, one has to take into account the sparsity of A. Usually the factors do not remain as sparse as the original matrix due to fill-in. Fill-in occurs if, during the basic operation

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{ik}^{(k)} \times a_{kj}^{(k)}}{a_{kk}^{(k)}}, \tag{2.1}$$

corresponding to the update of the reduced matrix after selection of pivot $a_{kk}^{(k)}$ at step k of the LU factorization, entry $a_{ij}^{(k+1)}$ becomes nonzero when $a_{ij}^{(k)}$ was zero.

Parallel sparse direct algorithms are designed to reduce the fill-in while keeping a maximum level of parallelism. Many algorithms have been developed. We will only detail the multifrontal approach, but we can cite supernodal approaches (see for example SuperLU [40]) and Fan-both algorithms [11]. The SuperLU and the multifrontal methods can be described by a computational tree, precomputed during a symbolic analysis phase only based on the matrix structure, whose nodes represent computations and whose edges represent transfer of data. In the case of the multifrontal method, at each node, some steps of Gaussian elimination are performed on a dense frontal matrix and the Schur complement that remains is passed for assembly at the parent node. In the case of the supernodal code the distributed memory version uses a right-looking formulation which, having computed the factorization of a block of columns corresponding to a node of the tree, then immediatly sends the data to update the block columns corresponding to ancestors in the tree [7].

2.3.2 The multifrontal method

The multifrontal method is used to compute the LU or LDL^T factorizations of a general sparse matrix. The multifrontal technique was developed by Duff and Reid [48] for computing the solution of indefinite sparse symmetric linear equations using Gaussian elimination and was then extended to solve more general unsymmetric matrices by Duff and Reid [49]. We refer to [47, 83] for a detailed description of the multifrontal technique.

The multifrontal algorithm consists of three steps : the symbolic analysis, the numerical factorization and the solution.

Symbolic analysis

This step consists of generating an ordering and data structures for the subsequent numerical factorization. The reordering is chosen so that pivoting down the diagonal in order on the resulting permuted matrix PAP^T produces much less fill-in and work than computing the factors of A by pivoting down the diagonal in the original order. This reordering is computed using only information on the matrix structure without taking into account the numerical values and so may not be stable for general matrices. However, if the matrix A is symmetric positive-definite a Cholesky factorization can be safely used. This technique of preceding the numerical factorization with a symbolic analysis can also be extended to unsymmetric systems although the numerical factorization phase must allow numerical pivoting [47]. Unfortunately, this problem is NP-complete [113], so heuristics are used. A standard ordering is the minimum degree algorithm or one of its variants [3]. Other methods are based on nested dissection algorithms [58]. But most of the state-of-the art ordering packages hybridize these methods by performing incomplete nested dissection and ordering the subgraphs associated with subtrees corresponding to the leaves of the separation tree by minimum degree. Experiments coupling the nested dissection algorithm with an approximate minimum degree algorithm show performance improvements both in term of fill-in reduction and concurrency during numerical factorization [7, 91].

The symbolic analysis does not take into account numerical values. If two matrices A_1 and A_2 have the same structure, only one analysis is needed. Particularly, in our semi-conductor simulation, we only need two analysis phases, one for the SPD systems and one for the unsymmetric systems.

Numerical factorization

All elimination operations take place within a dense submatrix, called a *frontal matrix*. The frontal matrix can be partitioned as

$$\begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix}$$

and pivots at this stage in the elimination can be chosen from within the block F_{11} only. The Schur complement $F_{22} - F_{21}F_{11}^{-1}F_{12}$ is computed and used to update later rows and columns of the matrix. We call this matrix, the *contribution block*. The overall factorization of the sparse matrix using a multifrontal scheme, can be described by an *assembly tree*, where the nodes correspond to computations of the Schur complement as just described, and the edges represent the transfer of the contribution block which is assembled (or summed) with other contribution blocks and original matrix entries at the parent node in the tree. An important aspect of the assembly tree is that operations at nodes which are not ancestors or descendents of each other are independent thus giving the possibility for obtaining parallelism from the tree. The use of dense submatrices has several advantages. It is possible to use level 3 BLAS optimized subroutines and indirect addressing is avoided. For the unsymmetric case, dynamic pivoting must be allowed during the factorization phase to ensure numerical backward stability.

Solve

In this phase, the factors computed during the factorization phase are used to compute the solution via backward and forward substitution. The assembly tree can also be used to identify parallelism during this step. When the problem is numerically difficult, a few steps of iterative refinement [47] are often performed to improve the accuracy of the solution.

2.3.3 The MUMPS software

The software MUMPS (MUltifrontal Massively Parallel Solver) is an implementation of the multifrontal technique for distributed memory environments. It was initially developed in the framework of the PARASOL Project [8]. PARASOL was an ESPRIT IV Long Term Research Project for "An Integrated Environment for Parallel Sparse Matrix Solvers". The main goal of this project was to build and test a portable library for solving large sparse systems of equations on distributed memory systems.

The software MUMPS [4, 6, 89] is written in FORTRAN 90 and uses the new functionalities of this language (modularity, dynamic memory management) to be an efficient modern code easy to use. The message passing library used is MPI. We present here the main features of this code.

Factorization of sparse symmetric positive definite matrices (LDL^T factorization), general symmetric matrices and general unsymmetric matrices (LU factorization).

Entry format for the matrices. The matrix of the linear system to be solved can be given to MUMPS in different formats. The three formats that can be used are :

- the centralized format where the matrix is stored in coordinate format [51] on one processor that is called the host,
- the distributed format where each processor involved in the solution has access to a subset of the matrix described in a coordinate format, defined in a global ordering,
- the elemental format where the matrix is described as a sum of dense elementary matrices. This latter format is natural in some finite element codes where the small dense matrices are the elementary matrices.

Parallel factorization and solve phase (uniprocessor execution also possible). The symbolic analysis phase remains sequential and is centralized on a processor designated as *the host* in the MPI communicator. The default algorithm for the ordering is an approximate minimum degree (AMD) [3] but the user can provide any other ordering. Classical threshold numerical pivoting is allowed during the numerical phase. Moreover, MUMPS can adapt to computer load variations during the numerical phase. Dynamic distributed scheduling is used to obtain this feature.

Backward error analysis. MUMPS can calculate a sparse backward error using the theory and metrics developed in [9] and it can perform iterative refinement to reduce the backward error down to machine precision if required.

Null space functionalities. MUMPS provides options for rank detection and computation of the null space basis. The dynamic pivoting strategy available in both the symmetric and unsymmetric version of MUMPS postpones all the singularities to the root. Therefore, the problem of rank detection for the original matrix is reduced to the problem of rank detection for the root matrix. At this root, rank revealing algorithms are applied [30, 35]. The null space basis is then computed using backward transformations. This latter feature is of primal interest in domain decomposition methods like BNN [84] and FETI [54].

Computation of a Schur complement. Let A be the partitioned matrix

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

where A_{11} and A_{22} are two square matrices coupled by the two rectangular matrices A_{12} and A_{21}. The matrix $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ is called the Schur complement matrix.

MUMPS can return a Schur complement to the user. The user must specify the list of indices of the Schur matrix. MUMPS then provides both factorization of the A_{11} matrix and the explicit Schur matrix S. The Schur matrix is returned as a dense matrix. The partial factorization that builds the Schur matrix can also be used to solve linear systems associated with the matrix A_{11}.

This functionality of MUMPS is critical in our implementation of domain decomposition methods as we will see in Section 2.4.

2.4 Domain decomposition methods

2.4.1 Introduction

In our case we consider domain decomposition as an algebraic technique used to solve in parallel the linear systems arising from the discretization of PDEs and based on a splitting of the domain into subdomains. These methods can be divided into two classes : overlapping and non-overlapping algorithms. The overlapping algorithms are referred to as Schwarz algorithms [97] and the non-overlapping as Schur algorithms. Schwarz methods consist in building block preconditioners for an iterative solver on the complete system. To be efficient these methods need the domains to overlap. It is complex to build this overlap in the case of unstructured meshes. Furthermore, a preliminary study [106] has shown that the additive Schwarz technique was not effective on the class of problems we are interested in. We therefore concentrate our studies on the Schur complement method. For details of the Schwarz method we refer to [32, 45, 101].

2.4.2 Schur complement method

This method is also called *substructuring*, referring to structural mechanics problems which were historically the first area of application for this method. It consists in first solving the interface problem and then the internal problems on each subdomain. A detailed and exhaustive overview of the Schur complement methods can be found in [32, 101].

Let $Au = f$ be the linear problem to solve. We assume that the domain Ω is partitioned into N non-overlapping subdomains Ω_1, ..., Ω_N with boundaries $\partial\Omega_1$, ..., $\partial\Omega_N$. Let B be the set of all indices of the discretized points which belong to the interfaces between the subdomains. Grouping the points corresponding to B

in the vector u_B and those corresponding to the interior I of the subdomains in u_I, we get the reordered problem :

$$\begin{pmatrix} A_{II} & A_{IB} \\ A_{BI} & A_{BB} \end{pmatrix} \begin{pmatrix} u_I \\ u_B \end{pmatrix} = \begin{pmatrix} f_I \\ f_B \end{pmatrix}. \tag{2.2}$$

Eliminating u_I from the second block row of (2.2) leads to the following reduced equation for u_B :

$$Su_B = f_B - A_{BI}A_{II}^{-1}f_I, \text{ where } S = A_{BB} - A_{BI}A_{II}^{-1}A_{IB} \tag{2.3}$$

is the Schur complement of the matrix A_{II} in A, and is usually referred to as the *Schur complement matrix*. The matrix S inherits from A the symmetric positive definiteness property. Algorithm 5 describes the Schur complement method.

Step 1. Reorder the unknowns so that the unknowns on the interface (B) are the last ones.

$$\begin{pmatrix} A_{II} & A_{IB} \\ A_{BI} & A_{BB} \end{pmatrix} \begin{pmatrix} u_I \\ u_B \end{pmatrix} = \begin{pmatrix} f_I \\ f_B \end{pmatrix}$$

Step 2. Solve the problem on the interface

$$Su_B = f_B - A_{BI}A_{II}^{-1}f_I, \text{ with } S = A_{BB} - A_{BI}A_{II}^{-1}A_{IB}$$

Step 3. Solve the problem for the interior unknowns

$$A_{II}u_I = f_I - A_{IB}u_B$$

Algorithm 5: Algorithm of the Schur complement method.

We define Γ_i the internal frontier of the subdomain Ω_i as $\Gamma_i = \partial\Omega_i \setminus \partial\Omega$ and the whole interface Γ as $\Gamma = \cup\Gamma_i$. Let $R_{\Gamma_i} : \Gamma \to \Gamma_i$ be the canonical pointwise restriction which maps full vectors defined on Γ into vectors defined on Γ_i, and let $R_{\Gamma_i}^T : \Gamma_i \to \Gamma$ be its transpose. For a matrix A arising from a finite-element discretization, the Schur complement matrix (2.3) can also be written as

$$S = \sum_{i=1}^{N} R_{\Gamma_i}^T S^{(i)} R_{\Gamma_i}, \tag{2.4}$$

where

$$S^{(i)} = A_{\Gamma_i}^{(i)} - A_{\Gamma_i i}A_{ii}^{-1}A_{i\Gamma_i} \tag{2.5}$$

is referred to as the local Schur complement associated with the subdomain Ω_i. $S^{(i)}$ involves submatrices from the local matrix $A^{(i)}$, defined by

$$A^{(i)} = \begin{pmatrix} A_{ii} & A_{i\Gamma_i} \\ A_{\Gamma_i i} & A_{\Gamma_i}^{(i)} \end{pmatrix}, \tag{2.6}$$

where $A^{(i)}$ is the local discretization of the problem on the subdomain Ω_i. We can also denote by u_i the unknowns corresponding to the internal edges of the subdomain Ω_i and f_i its associated right-hand side.

When implemented on a distributed environment, the matrix S is usually not fully assembled. We solve the Schur complement system in parallel by an iterative or a direct distributed method.

2.4.3 Iterative substructuring

Krylov subspace methods

Krylov methods are iterative methods based on the projection onto a subspace $\kappa(A, v, j) = Span\{v, Av, A^2v, \ldots, A^{j-1}v\}$ called a Krylov subspace associated with v and A where A is the matrix of the linear system to solve and v is usually the starting residual. For a general presentation of Krylov methods we refer to [68, 95].

In the symmetric positive definite case, the method of choice is the conjugate gradient algorithm [73]. The conjugate gradient has two remarkable properties. The first advantage is that it minimizes the A-norm of the forward error on $\kappa(A, v, j)$. The second advantage is that it is based on a simple triple recursion and so it only requires the storage of three vectors.

In the general unsymmetric case it is not possible to combine these two advantages [52]. For example the GMRES algorithm [96] minimizes the 2-norm of the residual on $\kappa(A, v, j)$ but is not based on a short recurrence and requires storage for k vectors at step k of the iterative process. On the contrary, the BiCGStab algorithm [107] is based on short recurrences but the iterate $x^{(i)}$ does not satisfy any optimal criterion on $\kappa(A, v, j)$.

These algorithms when implemented in numerical libraries for parallel computations only require from the user the implementation of three computational kernels :

- the dot product calculation,
- the matrix-vector product,
- applying the preconditioner to a vector.

Preconditioning techniques will be developed in detail in Chapter 3. We propose now two ways of computing the matrix-vector product with the Schur complement. These two possibilities define two algorithms : implicit iterative substructuring and explicit iterative substructuring.

Implicit iterative substructuring

For this approach the matrices A_{ii} are factorized by a sparse direct solver independently on each subdomain, consequently in parallel on a distributed computer. If A is symmetric positive definite we use a Cholesky factorization to obtain

$$A_{ii} = L_i L_i^T,$$

where L_i is the lower triangular factor. If A_{ii} is unsymmetric we use a LU algorithm to obtain

$$A_{ii} = L_i U_i,$$

where L_i is the lower triangular factor and U_i the upper triangular factor. These factorizations are computed by using uniprocessor version of MUMPS concurrently on each subdomain.

In an implicit iterative substructuring method, described by Algorithm 6, the factors of A_{ii} are used to compute the local matrix-vector products for the local Schur complement (2.5). This is done via a sequence of sparse linear algebra computations, namely a sparse matrix-vector product by $A_{i\Gamma_i}$, then sparse forward/backward substitution using the computed factors of A_{ii}, and finally a sparse matrix-vector product by $A_{\Gamma_i i}$.

Step 1. Each processor computes the local discretization matrix $A^{(i)}$ associated with its subdomain.
All the processors compute a factorization of the internal problem $A_{ii} = L_i U_i$ (or $L_i L_i^T$ in the symmetric positive definite case).
Step 2. The interface problem $Su_B = g$ is solved by an iterative Krylov solver. The computation of the *matrix-vector product*

$$y \leftarrow Sx$$

is done in two steps :
Step a) is completely parallel and does not need any communication between the processors.

$$\begin{cases} \forall i \in \{1, \ldots, n\} \\ y_i \leftarrow A_{i\Gamma_i} R_{\Gamma_i} x_i \\ y_i \leftarrow A_{ii}^{-1} y_i, \text{ computed by the solution of two triangular systems} \\ \quad \text{with the factors computed at Step 1.} \\ y_i \leftarrow A_{\Gamma_i}^{(i)} x_i - A_{\Gamma_i i} y_i \end{cases}$$

Step b) needs some exchange of informations between neighbouring subdomain

$$y \leftarrow \Sigma_{i=1}^{n} R_{\Gamma_i}^T y_i$$

Step 3. The internal problem is solved in parallel without any communications on each subdomain. On each subdomain the linear system

$$A_{ii} u_i = f_i - A_{i\Gamma_i} R_{\Gamma_i} u_B$$

is solved via forward/backward substitution using the factors computed at Step 1.

Algorithm 6: Implicit iterative substructuring.

Explicit iterative substructuring

In the case of explicit iterative substructuring the local Schur complement matrices $S^{(i)}$ are computed explicitly using the Schur complement computation feature of MUMPS (see Section 2.3.3) concurrently on each processor. On the contrary, in the implicit case, we only know how to compute the matrix-vector products with these local Schur complements using forward/backward substitutions using the factors associated with the local Dirichlet problem. Explicit iterative substructuring is described by Algorithm 7.

Step 1. Each processor computes the local discretization matrix $A^{(i)}$ associated with its subdomain.
All the processors compute a factorization of the internal problem $A_{ii} = L_i U_i$ (or $L_i L_i^T$) and generate the local Schur complement matrix $S^{(i)}$.
Step 2. The interface problem $Su_B = g$ is solved by an iterative Krylov solver. The computation of the *matrix-vector product*

$$y \leftarrow Sx$$

is done in two steps :
Step a) is completely parallel and does not need any communication between the processors.

$$\begin{cases} \forall i \in \{1, \ldots, n\} \\ y_i \leftarrow R_{\Gamma_i} x_i \\ y_i \leftarrow S^{(i)} y_i \end{cases}$$

Step b) needs some exchange of informations between neighbouring subdomains.

$$y \leftarrow \Sigma_{i=1}^{n} R_{\Gamma_i}^T y_i$$

Step 3. The internal problem is solved in parallel without any communications on each subdomain. On each subdomain the linear system

$$A_{ii} u_i = f_i - A_{i\Gamma_i} R_{\Gamma_i} u_B$$

is solved via forward/backward substitution using the factors computed at Step 1.

Algorithm 7: Explicit iterative substructuring.

There are several advantages to the explicit algorithm. In the implicit case, the core of the matrix-vector product needs two sparse triangular solves on the internal unknowns of each subdomain. In the explicit case, it consists of a call to DGEMV, the dense level 2 BLAS matrix-vector subroutine on each subdomain on a block of size the number of unknowns on the interface of the subdomain.

The other advantage is that the explicit algorithm will allow us to build more efficient preconditioners for the Schur complement system as presented in Chapter 3

because we have access to the entries of the local Schur complement matrices $S^{(i)}$.

There are also some drawbacks to this method. First, the factorization step (Step 1 of Algorithm 6 and Algorithm 7) is longer as we have more operations to perform to make the updates on the whole matrix to compute the Schur complement. This method also implies additional storage to hold the local Schur complement as a dense matrix. When the size of the interface is small relative to the number of internal unknowns, the explicit method will be more efficient than the implicit one. This is usually the case for 2D problems like the one we are treating. For 3D problems, the storage/computation of the local Schur complement might not be affordable. Furthermore, even if the factorization step is longer, only a small number of Krylov iterations is needed to make the explicit method better than the implicit one. A performance comparison of these two approaches is presented in Chapter 4.

2.4.4 Direct substructuring

Step 2 of Algorithm 7 consists in solving the interface problem by a Krylov iterative method. Direct substructuring consists in replacing the Krylov solver by a parallel direct method.

The Schur complement matrix is unassembled and distributed over all the different processors as the local Schur complement matrices. The interface system can be solved by a distributed sparse direct linear solver like MUMPS (see Section 2.3.3). The method is described by Algorithm 8.

Step 1 Each processor has access to the local discretization matrix $A^{(i)}$.
All the processors compute in parallel and without communication a factorization of the internal problem $A_{ii} = L_i U_i$ (or $L_i L_i^T$) and the local Schur complement matrix $S^{(i)}$.
Step 2 The Schur complement system $Su_B = g$ is solved by MUMPS used as a parallel solver with distributed matrix entries (the matrix S is viewed as a set of local Schur complement matrices).
Step 3 : The internal problem is solved in parallel without any communications on each subdomain. On each subdomain the linear system

$$A_{ii} u_i = f_i - A_{i\Gamma_i} R_{\Gamma_i} u_B$$

is solved via forward/backward substitution using the factors computed at Step 1.

Algorithm 8: Direct substructuring.

It would be possible to obtain an algorithm equivalent to Algorithm 8 by only using a single instance of MUMPS on the complete matrix. This would consist in providing MUMPS with an ordering composed by two steps. The first step would partition the graph of the complete matrix into subgraphs, each of the subgraphs would be associated with one subdomain generated by the mesh partitioner. Then on each subgraph an AMD ordering would be applied. The overall ordering would

result from a combination of nested dissection [58] (to define the subgraphs) and AMD (within each subgraph). If one uses this ordering instead of AMD on the complete graph of the matrix (as it is done by default in MUMPS), one will obtain a sequence of numerical operations for the factorization similar to those defined by Algorithm 8 (except for some slight variations that may occur due to a different dynamic pivoting during the factorization phase of unsymmetric matrices).

From a software point of view there are several differences between direct substructuring and an application of MUMPS to the matrix A distributed among the processors. In direct substructuring, the analysis phase is distributed for the internal problems but during the factorization phase load balancing is no longer possible and numerical pivoting is limited to the local subdomains.

Chapter 3

Preconditioned iterative methods for the Schur complement

In this Chapter we present numerical techniques for solving the Schur complement system using iterative substructuring methods. In Section 3.2 we introduce the one-level preconditioners, while the two-level approaches are described in the next section. In Section 3.4 we present some scaling techniques that we consider in a preprocessing phase for the Schur complement system. Finally in Section 3.5 we describe the stopping criterion implemented in the packages of iterative linear solvers.

3.1 Introduction

In iterative substructuring (see Section 2.4.3) the interface problem is solved using a Krylov solver. The possible weakness of iterative methods is their potential lack of robustness compared with direct solvers. However both the efficiency and the robustness can be improved by using *preconditioning techniques*. Preconditioning consists in transforming the original linear system into one which has the same solution; but it is expected that the transformed linear system is easier to solve.

Let us first consider the SPD situation. Let S be a SPD Schur matrix. An upper bound of the rate of convergence of the conjugate gradient method, when solving the linear system $Su = g$, depends on the condition number $\kappa(S)$ of the matrix that is defined by the ratio $\lambda_{max}/\lambda_{min}$ where λ_{max}, λ_{min}, denotes the largest, the smallest respectively, eigenvalue of S. The idea of preconditioning is then to replace the linear system

$$Su = g \tag{3.1}$$

by the equivalent linear system

$$MSu = Mg, \tag{3.2}$$

where M is a non singular matrix such that $\kappa(MS) < \kappa(S)$. The ideal but somehow conflicting features of M would be that M is an approximation of S^{-1}, M is not

expensive to compute and to store and the matrix-vector product Mv is easy to compute. Another constraint for SPD linear systems is that the preconditioner M should be SPD. The preconditioning defined by (3.2) is called left preconditioning due to its location with respect to the original matrix. The operator MS is no longer self-adjoint for the Euclidean inner product but is still self-adjoint for the M-inner product and therefore it is possible to define a preconditioned conjugate gradient algorithm (see for instance [95]). Another way to preserve the symmetry is to use the classical conjugate gradient method but on the system

$$LSL^T v = Lg,\ u = L^T v,$$

where $M = LL^T$ is defined in a factorized form, that is L is the Cholesky factor of M. This preconditioning is called split preconditioning and is mathematically equivalent to left preconditioning. Finally, it is also possible to define a right preconditioning technique which consists in solving

$$SMv = g,\ u = Mv.$$

The right preconditioned conjugate gradient method with the M^{-1}-inner product is also mathematically equivalent to the left preconditioned conjugate gradient with the M-inner product [95]. As these three preconditioning techniques are mathematically equivalent we have only considered the classical case of left preconditioning in our numerical experiments.

For unsymmetric matrices, we can also define a left, a right and split preconditioning, based for instance on the LU decomposition of the preconditioner. Unlike the conjugate gradient case, the iterates are different for these three preconditioning techniques when used with GMRES. In the unsymmetric case we do not have any convergence bound based on the condition number of the matrix, but some arguments exist [50] for diagonalizable matrices that indicate the bad convergence effect of the smallest eigenvalues.

Preconditioning theory for SPD problems often refers to spectral properties of the matrix to be preconditioned. The linear systems arising in the semi-conductor simulation code are too complex to enable to get any a priori idea of the spectral properties of the Schur complement matrices we encounter during the calculation. First, the discretized problem corresponding to the PDE is nonlinear and the linear systems that we are solving result from a linearization within a Newton process. Moreover, the flux equations are treated after having algebraically eliminated the potential terms; consequently these equations do not correspond anymore to the original PDE. So all the convergence bounds known in the elliptic case for the preconditioners described in Sections 3.2 and 3.3 are not guaranteed for our semi-conductor problems.

In Section 3.2 and 3.3 we describe some preconditioners for the Schur complement that apply both to SPD and non-symmetric matrices. In Section 3.4 we present some scaling techniques to reduce the magnitude gap between the coefficients of the Schur complement matrix. Finally, in Section 3.5 we introduce the stopping criterion used for the Krylov solvers.

3.2 Local preconditioners for the Schur complement

3.2.1 Neumann-Neumann Preconditioner

This local preconditioner is based on the local Schur complement matrices $S^{(i)}$ and was originally proposed in an analytic form in [22] and further studied in [39, 103]. It can be formulated algebraically as

$$M_{NN} = \sum_{i=1}^{N} R_{\Gamma_i}^T D_i^T (S^{(i)})^+ D_i R_{\Gamma_i}, \tag{3.3}$$

where $(S^{(i)})^+$ denotes the pseudo-inverse of the local Schur complement $S^{(i)}$ that might be singular. This is for instance the case for the internal subdomains for diffusion equations. The matrices D_i are diagonal matrices and define a partition of unity, i.e., $\sum_{i=1}^{N} R_{\Gamma_i} D_i = I$.

In the framework of that study, we use the implementation of this preconditioner developed at Parallab (University of Bergen) [15]. This package only address SPD linear systems and the unsymmetric version proposed by [2] is an ongoing work at Parallab.

3.2.2 Block preconditioners

Generalities

The preconditioners presented in this section have been initially proposed in [26, 27, 28]. In order to describe these preconditioners, we need to first define a partition of B, the set of edges of the discretization belonging to the interface between the subdomains. Let U be the algebraic space of vectors where the Schur complement is defined and $(U_i)_{i=1,p}$ a set of subspaces of U such that

$$U = U_1 + U_2 + \cdots + U_p.$$

Let R_i be the canonical pointwise restriction from U to U_i. Its transpose extends grid functions in U_i by zero to the rest of U. Using the above notation, we can define a wide class of block preconditioners by:

$$M_{loc} = \sum_{i=1}^{p} R_i^T M_i^{-1} R_i, \tag{3.4}$$

where

$$M_i = R_i S R_i^T. \tag{3.5}$$

Remark 1 : If the operator R_i^T is of full rank and if S is symmetric and positive definite, then the matrices M_i, defined in Equation (3.5) are SPD. Consequently

M_{loc} defined in Equation (3.4) is also SPD.

Remark 2 : If $U = U_1 \oplus .... \oplus U_n$, then M_{loc} is a block Jacobi preconditioner. Otherwise, M_{loc} is a block diagonal preconditioner with an overlap between the blocks as $U_i \cap U_j \neq \emptyset$. In this case, the preconditioner can be viewed as an algebraic additive Schwarz preconditioner for the Schur complement.

The preconditioners are requested to be efficient on parallel distributed memory platforms. Therefore, we do mainly consider subspaces U_i that involve information mainly stored in the local memory of the processors; that is information associated with only one subdomain and its closest neighbours. We present two decompositions of U :

1. each common interface $E_k = \partial\Omega_i \cap \partial\Omega_j$ between two subdomains of the decomposition giving rise to the edge preconditioner;
2. each interface Γ_i of the subdomains giving the subdomain preconditioner.

Block Jacobi preconditioner

We define the *common interface* E_i between subdomain Ω_j and subdomain Ω_l as the set of edges belonging to $(\partial\Omega_j \cap \partial\Omega_l)$. The set B can be partitioned into m *common interfaces* $E_i, i \in \{1, \ldots, m\}$, $B = (\bigcup_{i=1}^{m} E_i)$.

For each common interface E_i we define $R_i \equiv R_{E_i}$ as the standard restriction from B to E_i. Its transpose extends vectors in E_i by zero to the rest of the interface. Thus, $M_i = R_{E_i} S R_{E_i}^T = S_{ii}$. Using the above notation we define the following local preconditioner by

$$M_{bJ} = \sum_{E_i} R_{E_i}^T S_{ii}^{-1} R_{E_i}. \tag{3.6}$$

This preconditioner aims at capturing the interaction between neighbouring edges within the same common interface between two subdomains. This preconditioner is the straightforward block Jacobi that is well-known to be efficiently parallelizable.

Subdomain based preconditioner

In this alternative [27], we try to exploit all the information available on each subdomain and we associate each subspace U_i with the entire boundary Γ_i of subdomain Ω_i. Here, we have $R_i \equiv R_{\Gamma_i}$. The local matrix $M_i = \bar{S}^{(i)}$ is called the *assembled local Schur complement* and corresponds to the restriction of the complete Schur matrix to the interface of the subdomain Ω_i. This splitting $(U_i)_i$ is not a direct sum of the space U and we have introduced some overlap between the blocks defining

the subdomain preconditioner M_{AS}. This preconditioner can be written as:

$$M_{AS} = \sum_{i=1}^{N} R_{\Gamma_i}^T (\bar{S}^{(i)})^{-1} R_{\Gamma_i} \ . \tag{3.7}$$

We refer to this preconditioner as the M_{AS} preconditioner because it can be viewed as an *Additive Schwarz* preconditioner for the Schur complement system. One advantage of using the assembled local Schur complements instead of the local Schur complements (like in the Neumann-Neumann case) is that in the SPD case the assembled Schur complements cannot be singular (as S is not singular).

Implementation remarks

In the case of implicit iterative substructuring algorithms (see Section 2.4.3), the M_{bJ} preconditioner can be built using the probing [31] technique which requires multiple matrix-vector products by S. Unfortunately the probing approach cannot be applied to the M_{AS} preconditioner. The only way to recover the assembled local Schur complement would be to apply the Schur complement matrix to each vector of the canonical basis; this would be too computationally expensive and consequently unpractical for real computation.

In the case of explicit iterative substructuring (see Section 2.4.3) the local Schur complement matrices are explicitly known. A natural alternative to the probing technique is then simply to sum the explicitly computed $S^{(i)}$ to build the local assembled Schur matrices $\bar{S}^{(i)}$. This operation can be done using only one message exchange between each neighbouring subdomains Ω_i and Ω_j sharing the interface E_{ij}. The data communicated is the local Schur complement matrices restricted to the edge E_{ij} that is a $(n_{ij} \times n_{ij})$ matrix where n_{ij} is the number of unknowns along the edge E_{ij}.

Once the assembled local Schur complements $\bar{S}^{(i)}$ have been computed it is easy to build either the M_{bJ} preconditioner or the M_{AS} preconditioner. The construction of the M_{bJ} preconditioner is cheaper as we only need to factorize the diagonal blocks of $\bar{S}^{(i)}$ corresponding to the interfaces between two subdomains. M_{AS} requires the factorization of the complete assembled local Schur complement. The application of the M_{bJ} preconditioner is also cheaper. For the sake of simplicity in our implementation we choose to redundantly factorize the diagonal block associated with each edge E_{ij} on the processor dealing with Ω_i and the one in charge of Ω_j. The first advantage is the simplicity for the implementation as we do not have to logically assign an interface to a subdomain; secondly this avoid to implement a communication after the forward/backward substitution when the preconditioner is applied. This latter communication step is implemented for M_{AS}.

3.3 Two-level preconditioners for the Schur complement

3.3.1 Motivations for two-level algorithms

The Green's functions associated with elliptic partial differential equations are global. Consequently the solution at any point depends on the solution everywhere in the domain. Therefore, for solving the systems arising from the discretization of these equations, we have to provide a mechanism to represent this global coupling/behaviour.

Various preconditioners, that have appeared in the eighties and nineties, have suggested different ways for constructing the global coupling mechanism, referred to as the coarse-space components, and for combining them with local preconditioners. Although the local block preconditioners proposed in Section 3.2 introduce some exchanges of information, these exchanges remain local to the neighbouring edges or subdomains and introduce no global coupling mechanism. This mechanism is necessary to prevent an increase in the number of iterations when the number of subdomains is increased.

In this Section we present the well known Balanced Neumann-Neumann [84] preconditioner as well as variants of the BPS [23] preconditioners described in [28].

3.3.2 Balanced Neumann-Neumann preconditioner

The balanced Neumann-Neumann preconditioner is a two-level extension of the Neumann-Neumann preconditioner presented in Section 3.2.1. This two-level preconditioner was first introduced in [84]. It can be formulated as

$$M^{-1}S = P + (I-P)M_{NN}S(I-P), \quad \text{and} \quad M_{NN} = \sum_{i=1}^{N} R_i^T D_i^T (S^{(i)})^{-1} D_i R_i.$$

Here, M_{NN} is the one-level Neumann-Neumann preconditioner (see Section 3.2.1). P denotes the S-orthogonal projection onto the coarse space defined by $Span\{\sum_{i=1}^{N} R_i^T D_i^T Z_i\}$ where Z_i contains at least the null-space of $S^{(i)}$ if any.

For our numerical experiments, we used a software package developed by Parallab (University of Bergen). One of the features of that software is its ability to construct the coarse space from local subspaces Z_i that are spanned by the eigenvectors associated with the smallest (in magnitude) eigenvalues of the local Schur complement matrices $S^{(i)}$; that consequently contains the null space of $S^{(i)}$ if any. We refer to [15] for the complete description of the preconditioner and its implementation.

We denote this preconditioner as $M_{BNN(k)}$ where k is the number of eigenvectors of each local Schur complements that are computed to build the coarse component of the preconditioner (therefore it can be considered as the number of degrees of

freedom per subdomain in the preconditioner). The implementation of the balancing Neumann-Neumann preconditioner that we use is only available for the SPD matrices. An unsymmetric version is under development at Parallab.

3.3.3 Coarse space components for local block preconditioners

General context

The preconditioners presented now are closely related to the BPS preconditioner [23], although we consider different coarse spaces to construct their coarse components. The class of two-level preconditioners that we define now can be described as follows:

$$M = M_{loc} + R_0^T (R_0 S R_0^T)^{-1} R_0$$

where R_0 is a restriction operator from the subspace U (where the Schur complement is defined) to a subspace U_0 of U called the coarse space and where M_{loc} is one of the local block preconditioners presented in Section 3.2.2.

The coarse space operator is defined by the Galerkin formula:

$$A_0 = R_0 S R_0^T,$$

represents in some way the Schur complement on the coarse space U_0. The global coupling mechanism is introduced by the coarse component of the preconditioner which can thus be defined as $M_{global} = R_0^T A_0^{-1} R_0$.

The coarse space preconditioners will only differ in the choice of the coarse space U_0 and the interpolation operator R_0^T. Similarly to the Neumann-Neumann and Balancing Neumann-Neumann preconditioner, R_0^T must be a partition of unity in U in the sense that

$$R_0^T \mathbf{1} = \mathbf{1}, \tag{3.8}$$

where the symbol $\mathbf{1}$ denotes the vectors of all 1's that have different size in the right and left hand side of (3.8).

Further references and numerical tests on this class of two level preconditioners can be found in [27, 28, 63, 108]. In our study we consider two different coarse space: the subdomain-based coarse preconditioner and the edge-based coarse preconditioner.

Subdomain based coarse space

With this coarse space we associate one degree of freedom with each subdomain. Let B be the set of edges belonging to the interface Γ between the subdomains. Let Ω_k be a subdomain and $\partial\Omega_k$ its boundary. Let Z_k be a vector defined on B and $Z_k(i)$ its i-th component. Then, the subdomain-based coarse space U_0 can be defined as

$$U_0 = \text{span}[Z_k : k = 1, \ldots, N], \text{ where } \quad Z_k(i) = \begin{cases} 1, & \text{if } i \in \partial\Omega_k \cap B \text{ and} \\ 0, & \text{otherwise.} \end{cases}$$

The considered restriction operator R_0 returns for each subdomain $(\Omega_i)_{i=1,N-1}$ the half-sum of the values at all the edges on the boundary of this subdomain.

If we associate to this coarse space the M_{AS} local preconditioner as defined in Section 3.2.2 we obtain a preconditioner we refer to as M_{AS-sub}.

Edge based coarse space

We refine the coarse space based on the subdomains and we introduce one degree of freedom per interface between two neighbouring subdomains, that is, when $\partial\Omega_i \cap \partial\Omega_j \neq \emptyset$. Let $E_k = \partial\Omega_i \cap \partial\Omega_j$ be the interface between subdomain Ω_i and Ω_j. Let Z_k be a vector defined on B and $Z_k(i)$ its i-th component. Let m_e denotes the number of common interfaces $E_i \subset B$, then, the edge based coarse space U_0 can be defined as:

$$U_0 = \text{span}[Z_k : k = 1, \dots, m_e], \text{ where } \quad Z_k(i) = \begin{cases} 1 & i \in E_k, \\ 0 & \text{otherwise.} \end{cases}$$

The set of vectors $\mathcal{B} = \{Z_1, Z_2, \dots, Z_{m_e}\}$ forms a basis for the subspace U_0, as these vectors span U_0 by construction and they are linearly independent. The considered restriction operator R_0 returns for each edge the sum of the values at all the edges on the interface between two neighbouring subdomains.

If we associate to this coarse space, the M_{AS} local preconditioner as defined in Section 3.2.2, we obtain a preconditioner we refer to as $M_{AS-edge}$.

3.4 Scaling techniques for the Schur complement

The dynamic of the computed quantities during the simulation is very high and leads to huge variations in the coefficients of the linear systems. Consequently this large variations also appear in the associated Schur complement systems and causes trouble to the convergence of the iterative scheme. In order to tackle this problem, we describe a set of scaling techniques that have been implemented and experienced. All those techniques are relatively easy to implement for scaling the Schur complement system when the local Schur complement are built explicitly.

We consider the solution of

$$Su = g, \tag{3.9}$$

and denote by (s_{ij}) the entries of S.

3.4.1 Diagonal scaling for the Schur complement

The symmetric diagonal scaling of (3.9) consists in solving

$$DSDv = Dg, \; u = Dv$$

where $D = diag((\sqrt{|s_{ii}|})^{-1}))$. When the original matrix S is symmetric, by construction, the diagonal scaling preserves this property as well as the positive definiteness, if S is.

3.4.2 Row and column scaling

When using a row scaling, each entries of a row in the original matrix is divided by the norm of that row. Different norms, such as the infinity-norm or the 1-norm, may be considered, depending on the strategy one wishes to develop. Here we consider the case of the infinity-norm. The system (3.9) is replaced by

$$D_r S u = D_r g$$

where $D_r = diag((\parallel S(i,:) \parallel_\infty)^{-1})$. Similarly we define the column scaling by replacing the original system (3.9) by

$$S D_c v = g,\ u = D_c v$$

with $D_c = diag((\parallel S(:,i) \parallel_\infty)^{-1})$. One drawback of these two latter methods is that they do not preserve the symmetry and cannot be applied for SPD systems.

3.4.3 Iterative row-column scaling

This algorithm has been proposed in [94]. It is an iterative procedure that scales asymptotically the infinity norm of both the rows and the columns to 1. Let D_r, D_c, denotes the row scaling matrix, column scaling matrix respectively, as described in the previous section. We define the scaling *rcs(1)* as

$$D_r S D_c v = D_r g,\ u = D_c v.$$

This procedure can be applied recursively to define the *rcs(2)* scaling as described by Algorithm 9. This procedure can be further iterated to define a *rcs(k)* scaling. If the matrix S is SPD, then the linear systems scaled using *rcs(1)* or *rcs(2)* remains SPD.

3.4.4 Relationship between the scalings on A and scalings on S

Let S denote the Schur complement matrix associated with the original matrix A. Instead of scaling the Schur complement system, it is also possible to scale the original matrix A before computing the local Schur complement matrices. We first consider the symmetric diagonal scaling for A meaning that the system $Ax = b$ is replaced by the $DADy = Db,\ x = Dy$ where D is the scaling matrix computed from the diagonal entries of A. If we order first the internal edges and then the ones on the interface we obtain

$$\begin{pmatrix} A_{II} & A_{I\Gamma} \\ A_{\Gamma I} & A_{\Gamma\Gamma} \end{pmatrix}, \tag{3.10}$$

as described in Section 2.4.2. Reordering in a consistent manner the diagonal scaling matrix leads to

$$\begin{pmatrix} D_I & 0 \\ 0 & D_\Gamma \end{pmatrix} \begin{pmatrix} A_{II} & A_{I\Gamma} \\ A_{\Gamma I} & A_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} D_I & 0 \\ 0 & D_\Gamma \end{pmatrix} = \begin{pmatrix} D_I A_{II} D_I & D_I A_{I\Gamma} D_\Gamma \\ D_\Gamma A_{\Gamma I} D_I & D_\Gamma A_{\Gamma\Gamma} D_\Gamma \end{pmatrix}. \tag{3.11}$$

$$S^{(1)} = S.$$

Let $D_r^{(1)}$ and $D_c^{(1)}$ the classical row and column scaling matrices for $S^{(1)}$.

$$S^{(2)} = D_r^{(1)} S^{(1)} D_c^{(1)}.$$

Let $D_r^{(2)}$ and $D_c^{(2)}$ the classical row and column scaling matrices for $S^{(2)}$.

Solve the linear system

$$D_r^{(2)} S^{(2)} D_c^{(2)} v = D_r^{(2)} * D_r^{(1)} g.$$

Compute the solution of the initial system

$$u = D_c^{(2)} * D_c^{(1)} v.$$

Algorithm 9: Iterative row column scaling of level 2 (*rcs(2)*) for the Schur complement system.

Eliminating the internal scaled equations we obtain

$$S_{scaled} = D_\Gamma A_{\Gamma\Gamma} D_\Gamma - D_\Gamma A_{\Gamma i} D_I (D_I A_{II} D_I)^{-1} D_I A_{I\Gamma} D_\Gamma = D_\Gamma S D_\Gamma$$

where S is the Schur system associated with the unscaled matrix A. This observation is also true for the row, the column and the *rcs(k)* scalings. This indicates that scaling the original matrix leads to scale the Schur complement S using entries of A. For instance for the symmetric diagonal scaling the Schur complement is scaled using the diagonal entries of $A_{\Gamma\Gamma}$ that might differ significantly from the diagonal entries of S. In that respect, only using a scaling on A might be inappropriate.

3.5 Stopping criterion for the linear iterative solvers

3.5.1 Backward error analysis

In this section, we recall the main ideas that govern the definition of suitable stopping criterion for iterative schemes [10]. These ideas are based on the backward error analysis introduced in [111]. For a detailed and up to date overview of this topic we refer to [29] in which we find the following explanations.

The essence of backward error analysis is to associate the exact and the finite-precision computations in a common framework by means of the following principle:

Consider the computed solution $\tilde{x}$ as the exact solution of a nearby problem.

This idea turns out to be much more powerful than it first appears:

i) It permits us to ignore the details of the computer arithmetic: the errors made during the course of the computation are interpreted in terms of equivalent perturbations to the given problem, and the computed quantities are *exact* for the perturbed problem.

ii) One advantage is that rounding errors are put on the same footing as errors in the original data, which, in case of PDEs, can be introduced by the discretization methods. The effect of uncertainty in data has usually to be considered in any case.

Such an error analysis is referred to as backward error analysis because the errors are *reflected back* into the original problem.

Let $\tilde{x} = \tilde{G}(y)$ be the computed solution for the problem (P) $F(x) = y$. The backward error measures the minimal distance of (P) to the set of perturbed problems that have exact solution $\tilde{x}$. Such a notion requires the specification of the admissible perturbations of (P). For example, in the case of linear solvers for the system $Ax = b$, the class of admissible perturbations of the data $Q = \{A, b\}$ and some norm $\| \, . \, \|$ has to be defined.

With these assumptions, we can get an estimation of the error on the computed solution through the first order bound:

$$\text{Forward Error} \;\leq\; \text{Condition Number} \;\times \text{Backward Error.} \tag{3.12}$$

Backward error analysis thus permits us to separate the error bound (3.12) into the product of

i) *condition number*, which depends on the equation $F(x) = y$ only,

ii) *backward error*, which depends on the algorithm and the arithmetic of the computer.

The condition number is imposed by the problem, and it is the aim of software developers to propose algorithms which provide a backward error of the order of machine precision. Unfortunately this is not always possible.

In the following Sections we examine the particular case of Krylov solvers for linear systems and the case of direct solvers for linear systems.

3.5.2 Krylov solvers

In iterative schemes the residual, and sometimes its norm, is available directly in the algorithm. For instance in CG, the residual is updated thanks to a simple recurrence and for GMRES, the 2-norm of the preconditioned residual is given for "free" by the update of the QR factorization of the Hessenberg matrix [96]. For those reasons, it

is convenient to use a stopping criterion based on a normwise backward error. For the solution of (3.9) it is defined by

$$\begin{aligned} \eta &= inf\,\{\omega;\ \| \Delta g \|_2 \leq \omega \| g \|_2 \text{ and } S\tilde{u} = g + \Delta g\} \\ &= \frac{\| S\tilde{u} - g \|_2}{\| g \|_2} \\ &= \frac{\| r \|_2}{\| g \|_2}. \end{aligned} \tag{3.13}$$

The iterative scheme is stopped when

$$\frac{\| r_n \|_2}{\| g \|_2} < \epsilon, \tag{3.14}$$

where ϵ is a threshold defined by the user and r_n the residual associated with the current iterate. It should be noticed that a common used stopping criterion based on the reduction of the residual norm

$$\frac{\| r \|_2}{\| r_0 \|_2} < \epsilon,$$

where r_0 is the initial residual, reduces to (3.14) if the initial guess is set to the null vector. The package we use for CG [55] and for GMRES [56] implement a normwise backward error stopping criterion. When GMRES is preconditioned with a left preconditioner, i.e. $MSu = Mg$, the 2-norm of the preconditioned residual given by the algorithm is used to define the stopping criterion, that is

$$\frac{\| Mr_n \|_2}{\| Mg \|_2} < \epsilon. \tag{3.15}$$

This stopping criterion is a backward error for the preconditioned system and not for the original system. With right preconditioner, the 2-norm of the preconditioned residual coincides with the residual of the original matrix. Consequently the backward error of the preconditioned system matches the one of the original system.

For preconditioned CG, the unpreconditioned residual is given by a short recurrence and the backward error associated with the original problem is easy to compute. One should also mention that for CG, there exists a technique [67] that enables one to get a cheap estimation of the A-norm of the forward error. This estimator will only be available in the next release of the package used in this work.

3.5.3 Direct solvers

In the case of direct solvers, the quality of the solution has to be estimated only after the forward/backward substitutions (and possibly after each of the few iterative

refinement steps). For those techniques the appropriate criterion [9] is the sparse componentwise backward error η defined by

$$\eta = inf\left\{\omega, \forall i, \forall j, |\delta A_{ij}| < \omega |A_{ij}|; |\delta b_i| < \omega |b_i|; (A + \delta A)\tilde{x} = (b + \delta b)\right\}. \quad (3.16)$$

The value of η can be computed by

$$\eta = max_i \frac{|b - A\tilde{x}|_i}{(|b| + |A||\tilde{x}|)_i}, \quad (3.17)$$

where the modulus sign on a vector or a matrix indicates the vector or the matrix obtained by replacing all entries by their moduli. The MUMPS software is able to calculate an estimate of the sparse backward error [89] using the theory and metrics developed in [9].

3.5.4 Embedded iterations

In the case of the semiconductor application, the linear solvers are embedded in nonlinear loops, consisting in Newton methods, embedded themselves in a semi-implicit time scheme (see Chapter 1).

Although theoretical results exist to analyse the influence of the linear solution on the Newton methods [76] they might be difficult to implement in a simulation code. To illustrate the difficulty to analyse the behaviour of embedded iterations, we can mention the a priori simple case of embedded linear solvers presented in [20, 21, 66] that nevertheless remains for the main part an open question in linear algebra.

Precisely analysing the behaviour of the embedded iterations for the semiconductor simulation code is very complex and out of the scope of this work. Nevertheless some numerical experiments will be given in Section 4.1.

Chapter 4

Numerical results and performance measurements

In this chapter, we present the numerical behaviour and performance measurements of the parallel semiconductor device simulation code. More precisely, we report on the behaviour of the parallel direct and iterative linear solvers presented in Chapters 2 and 3. For the sake of simplicity of exposure, we choose to keep separate the description of the numerical behaviour of the algorithms and their parallel performance on the selected parallel platform. In Section 4.1, we focus on the numerical behaviour of iterative substructuring algorithms. In Section 4.2 we compare, in terms of parallel computational time, iterative and direct solvers. In Section 4.3 we give some complementary information on some aspects that were not addressed in Sections 4.1 and 4.2.

4.1 Numerical behaviour of iterative substructuring algorithms

In this section, we focus only on the numerical behaviour of iterative substructuring algorithms. Because the overall performance is not only governed by the convergence behaviour, the performance and time measurements are analysed in Section 4.2.

In Section 4.1.1, we present the test cases selected to perform our experiments. In Section 4.1.3, we demonstrate the decisive influence of both scaling and preconditioning on the overall numerical behaviour of the methods. Then, in Section 4.1.4, we present some experiments that illustrate how the accuracy required for the Krylov solvers may modify the convergence of the nonlinear solvers. Finally, in Section 4.1.5, we investigate the numerical scalability of the preconditioners for the Schur complement systems.

4.1.1 Description of the test cases

We present the test cases selected to illustrate the numerical behaviour of our parallel linear solvers. The experiments described in this section have been performed on three meshes denoted by *Mesh S* (*S* stands for small), *Mesh M* (*M* stands for medium) and *Mesh L* (*L* stands for large). These three meshes are defined on the same geometry as that of the heterojunction transistor depicted in Figure 1.7. *Mesh S* has 102944 triangles, and the associated linear systems have 154892 unknowns. *Mesh M* has 237499 triangles, and the associated linear systems have 356701 unknowns. Finally, *Mesh L*, the biggest test case, has 806098 triangles, and the associated linear systems have 1214758 unknowns.

The main characteristics of *Mesh S*, *Mesh M* and *Mesh L* decomposed into 8, 16 or 32 subdomains are summarized in Tables 4.1, 4.2 and 4.3.

Size of the mesh	102944 triangles		
Size of the linear systems	154892		
Number of subdomains	8	16	32
Size of Schur matrix	958	1607	2446
Smallest size of a local Schur	110	119	98
Largest size of a local Schur	336	274	198

Table 4.1: Characteristics of *Mesh S* decomposed into 8, 16 or 32 subdomains.

Size of the mesh	237499 triangles		
Size of the linear systems	356701		
Number of subdomains	8	16	32
Size of Schur matrix	1607	2273	3606
Smallest size of a local Schur	242	139	94
Largest size of a local Schur	602	470	345

Table 4.2: Characteristics of *Mesh M* decomposed into 8, 16 or 32 subdomains.

Size of the mesh	806 098 triangles
Size of the linear systems	1 214 758
Number of subdomains	32
Size of Schur matrix	5180
Smallest size of a local Schur	115
Largest size of a local Schur	646

Table 4.3: Characteristics of *Mesh L* decomposed into 32 subdomains.

In all the simulations presented in this chapter, the boundary conditions of the problem are the same and correspond to the computation of one tension increment of 0.2 Volts. The stopping criteria for the Newton and the Euler schemes are also constant for all the simulations, and their values are both set to 10^{-7}.

The free parameter for those simulations is the choice of the method for solving the linear systems. One can choose either a direct or an iterative method. Two choices are possible for the direct method, the direct substructuring algorithm or the use of MUMPS on the complete matrix given in distributed input format. The main parameters we play with for the iterative schemes are the choice of the Krylov solvers, the required accuracy for the Krylov solvers, the preconditioner and the scaling. The number of possible combinations is too large to make an exhaustive parametric study. We only vary the ones that we found the most sensitive.

It is important to underline the fact that the linear solvers studied are embedded in a nonlinear process. Therefore, changing the linear solver might change the nonlinear path and consequently the entries of the linear systems solved might differ slightly and depend on the selected linear solver.

4.1.2 A remark on the construction of the right-hand side of the Schur system

Let

$$Su = g$$

be the Schur system to be solved. With the same notation as in Section 2.4, we have

$$g = f_B - A_{BI} A_{II}^{-1} f_I. \tag{4.1}$$

In (4.1), f_B is the restriction of the right hand-side of the original system to the interface Γ between the subdomains. Practically, f_B is distributed among the n subdomains as

$$f_B = \sum_{i=1}^{n} R_i^T f_B^{(i)}.$$

Because of the finite-element discretization, the magnitude of one entry in $f_B^{(i)}$ may be very different from the corresponding one in f_B. For example the value 0 may be decomposed into -1 and $+1$ between two neighbouring subdomains. In finite-precision arithmetic the vector

$$g_1 = \sum_{i=1}^{n} R_i^T (f_B^{(i)} - A_{\Gamma_i i} A_{ii}^{-1} f_i) \tag{4.2}$$

and the vector

$$g_2 = \sum_{i=1}^{n} R_i^T f_B^{(i)} - \sum_{i=1}^{n} R_i^T A_{\Gamma_i i} A_{ii}^{-1} f_i \tag{4.3}$$

can be different. In practice, we have observed that in (4.2) some values of $f_B^{(i)}$ cancel values of $A_{\Gamma_i i} A_{ii}^{-1} f_i$.

The formula (4.2) seems to be the most natural to compute the right-hand side of the Schur system as it only requires one communication between two neighbouring subdomains while (4.3) requires communication of two vectors. Unfortunately, it

took us some time to realize that numerically (4.2) leads to the wrong results as the steady state of the simulation is never obtained when it is used due to the cancellation described above. This remark illustrates the fact that the numerical behaviour of an algorithm cannot be completely decoupled from its software implementation. Therefore, even if we do not enter into the details of the implementation, cautious software development remains an important part of the work.

4.1.3 Choice of the scaling and the preconditioner

In the first part below, we first present the experiments performed and only report on the observed results. Those results are discussed further in the next paragraph entitled "Comments on the results".

Experiments and results

The two simulations, denoted by *Simulation 1* and *Simulation 2*, are considered to illustrate the influence of the scaling and the preconditioning. *Simulation 1* is performed on *Mesh S* decomposed into 16 subdomains (see Table 4.1) and *Simulation 2* is performed on *Mesh M* decomposed into 16 subdomains (see Table 4.2). The stopping criterion for the Krylov solver is set to 10^{-13}. The SPD Schur systems are solved using CG iterations. The unsymmetric Schur systems are solved by GMRES iterations without restart and with a right preconditioner.

The free parameters are the choice of the preconditioner and the choice of the scaling. The three possibilities selected for preconditioning are : no preconditioner (denoted by *none*), the local block diagonal preconditioner (denoted by M_{bJ}) and the preconditioner based upon the subdomains (denoted by M_{AS}). The preconditioners M_{bJ} and M_{AS} are presented in Section 3.2.

The different scalings are presented in Section 3.4. If not explicitly stated we apply the same scaling to the SPD and the unsymmetric linear systems for the simulations. We distinguish the following cases : no scaling (denoted by *no*), symmetric diagonal scaling on the original matrix (denoted by *diag on A*) or one of the five scalings on the Schur complement matrix S described in Section 3.4. They are denoted by *diag*, *max row*, *max col*, *rcs(1)* and *rcs(2)*. The scalings *diag*, *rcs(1)* and *rcs(2)* preserve the symmetry and the positive definitness of a matrix. So they can be applied to SPD as well as unsymmetric systems. The scalings *max col* and *max row* possibly destroy the symmetry of a matrix. They can only be applied on unsymmetric systems. During a simulation, they are used simultaneously with *diag* scaling for the SPD systems involved in the simulation.

The main parameters of *Simulation 1* and *Simulation 2* are summarized in Table 4.4. In Table 4.5 we report on experiments that illustrate the influence of the choice of the scaling and the preconditioner on the behaviour of the nonlinear Newton iterations. This table displays the total number of Newton steps required to obtain the steady state. The "$\times$" symbol indicates that the nonlinear scheme does not converge after 300 Newton steps. For the sake of completness, the number of

	Simulation 1	*Simulation 2*
Frozen parameters		
Mesh	Mesh S	Mesh M
Number of subdomains	16	16
Max number of Krylov iterations	200	200
SPD solver	CG	CG
Unsymmetric solver	GMRES	GMRES
GMRES preconditioning technique	right	right
ϵ_{Krylov}	10^{-13}	10^{-13}
Free parameters		
Scaling	No, diag on A, diag, max row, max col, rcs(1) or rcs(2)	
Preconditioning	None, M_{bJ} or M_{AS}	

Table 4.4: Parameters of *Simulation 1* and *Simulation 2*.

Newton steps needed to obtain the steady state with a direct substructuring method is also reported.

Simulation 1

	no	diag on A	diag	max row	max col	rcs(1)	rcs(2)
none	×	×	×	×	×	×	×
M_{bJ}	×	×	164	164	×	164	164
M_{AS}	×	×	164	164	×	164	164
Direct method	159						

Simulation 2

	no	diag on A	diag	max row	max col	rcs(1)	rcs(2)
none	×	×	×	×	×	×	×
M_{bJ}	×	×	179	179	×	179	182
M_{AS}	×	×	176	180	×	174	174
Direct method	173						

Table 4.5: Number of Newton steps during *Simulation 1* and *Simulation 2* varying the preconditioner and the scaling strategy. × means that the nonlinear scheme does not converge.

When the steady state is obtained, it is possible to compare the behaviour of the linear solvers. Table 4.6 displays the average number of CG iterations (GMRES iterations) needed to solve each SPD linear system (each unsymmetric linear system respectively) during the simulation. These results are discussed in the following subsection.

Simulation 1

	diag		max row		rcs(1)		rcs(2)	
	CG	GMRES	CG	GMRES	CG	GMRES	CG	GMRES
M_{bJ}	63	44	63	60	59	44	59	44
M_{AS}	27	21	27	50	27	21	27	21

Simulation 2

	diag		max row		rcs(1)		rcs(2)	
	CG	GMRES	CG	GMRES	CG	GMRES	CG	GMRES
M_{bJ}	43	40	43	41	43	40	46	40
M_{AS}	29	28	29	56	29	28	29	28

Table 4.6: Average number of Krylov iterations to solve each Schur system during *Simulation 1* and *Simulation 2* varying the preconditioner and the scaling strategy. CG is used for SPD systems and full GMRES is used for unsymmetric systems.

$dim(S_1)$	=	1607
σ_{min}	=	0.0055
σ_{max}	=	$2.9552 \cdot 10^{+25}$
$cond(S_1)$	=	$+\infty$
$rank(S_1)$	=	411
$max(max\ per\ row)$	=	$2.9282 \cdot 10^{+25}$
$min(max\ per\ row)$	=	0.0180
$dim(\overline{S_1})$	=	1607
σ_{min}	=	$5.9940 \cdot 10^{-04}$
σ_{max}	=	34.5309
$cond(\overline{S_1})$	=	$5.76 \cdot 10^{+04}$
$rank(\overline{S_1})$	=	1607
$max(max\ per\ row)$	=	1.0
$min(max\ per\ row)$	=	1.0

Table 4.7: Some characteristics of a Schur complement matrix S_1 taken from *Simulation 1*. $\overline{S_1}$ is S_1 with diagonal scaling.

Comments on the results

Table 4.5 shows that a combination of a preconditioner for S and a scaling on S is needed to ensure the convergence of the nonlinear scheme. The numerical difficulty of the problems considered explains the necessity of the combination of scaling and preconditioning. We have extracted from *Simulation 1* one matrix A_1 which corresponds to the discretization of the transport equation for the holes. Let S_1 be its associated Schur complement matrix, σ_{min} the minimum singular value of S_1 and σ_{max} its maximum singular value. To measure the bad row scaling of S_1, we have computed the maximum element per row, and then the maximum of the maximum per row and the minimum of the maximum per row. Then we denote by $\overline{S_1}$ the matrix corresponding to S_1 after having applied the diagonal scaling. We show, in Table 4.7, different parameters computed on both matrices using Matlab.

Actually, the very bad conditioning of S_1 (in our case detected by Matlab as rank defficient) can also be illustrated on the following simple 2×2 matrix

$$\begin{pmatrix} 1 & 0 \\ 0 & 10^{20} \end{pmatrix}$$

that is very badly scaled. The bad conditioning disappears when a simple diagonal scaling is applied. Furthermore, on the example of Table 4.7, the gap between the maximum and the minimum of the maximum per row has vanished.

As expected, the diagonal scaling on A does not give good results (see Section 3.4). We illustrate this on the matrices A_1 and S_1 of Table 4.7. We denote by $|B|$ the matrix whose entries are the absolute values of those of the matrix B. Let

d_S be the vector of diagonal entries of $|S_1|$. Let d_Γ be the vector of diagonal entries of the interface block $|A_{\Gamma\Gamma}|$ of the matrix $|A_1|$. We define the relative difference between the i^{th} component of d_S and d_Γ as the ratio

$$e_r(i) = \frac{|d_S(i) - d_\Gamma(i)|}{max(d_S(i), d_\Gamma(i))}.$$

The maximum relative difference found on this example is 0.9997 corresponding to a value of $1.7823 \cdot 10^3$ in the Schur complement matrix and $1.4908 \cdot 10^6$ in the original matrix. The minimum one is 0.6863 corresponding to a value of $1.9788 \cdot 10^{14}$ in the Schur complement matrix and $6.3069 \cdot 10^{14}$ in the original matrix. So the difference between the two vectors is large in this case. It explains the poor results observed when we scale A compared to those obtained when scaling S. We do not expect better results with a row or a column scaling on A. Moreover, these scalings are more complex to implement in a parallel distributed environment as they need additional communication.

Now we compare, in Tables 4.5 and 4.6, the three scalings that preserve the symmetry of S. Compared with the results of *diag*, the gains obtained with $r$cs(1) or $r$cs(2) are not significant. Because they require additional computation and communication, these two scalings have not been considered in the following experiments.

Concerning the *max row* and *max col* scalings, that do not preserve symmetry, we notice experimentally that column scaling is not efficient. The nonlinear convergence is lost and it is impossible to obtain the steady state. On the contrary, row scaling enables the convergence of the nonlinear scheme on all the experiments we have performed. We do not have any explanation for this phenomenon.

To conclude these comparisons, we compare diagonal scaling and row scaling. Experimentally, we observe that for GMRES row scaling is less efficient than diagonal scaling. In Table 4.6, we can see the number of GMRES iterations is larger when row scaling is used than when diagonal scaling is used. This is particularly due to the fact that a few linear systems do not converge to the required accuracy; however this non convergence does not destroy the nonlinear convergence.

Motivated by the results of this section, for all the other experiments reported in this Chapter, a diagonal scaling is applied to the Schur complement for the SPD and the unsymmetric linear systems.

For the sake of completness, we would like to point out the fact that the largest entries of the Schur complement matrices are often on the diagonal. From *Simulation 1* where 95 unsymmetric Schur complement systems have been solved : only 25 have more than 1 % of their largest entries per row not being the diagonal element; among these 25 , 20 have more than 10 % and among these 20, 11 have between 30 % and 40 % of their largest entries in each row outside the diagonal.

However, after the scaling, the Schur complement matrices remain difficult to solve by unpreconditioned Krylov iterations. That is why a good preconditioner has

to be combined with the scaling to ensure the nonlinear convergence. Concerning the comparison between the two local preconditioners, we see that the average number of Krylov iterations is larger with the M_{bJ} preconditioner than with the M_{AS} preconditioner. If M_{AS} is used on unscaled matrices, the linear systems still converge to the required accuracy, but the nonlinear convergence is lost. Therefore, scaling is the key parameter to ensure the robustness of the complete numerical method.

Finally, we also notice that, in any case, the direct substructuring algorithm requires less Newton steps than the iterative substructuring algorithms to obtain the steady state solution. This tends to indicate that the better accuracy provided by the direct method better helps the nonlinear convergence.

In order to emphasize how crucial the scaling is for the nonlinear convergence, we mention that without any scaling and with the M_{AS} preconditioner, the nonlinear scheme does not converge while each linear system converges in normwise backward error to the 10^{-13} accuracy.

4.1.4 Influence of the accuracy of the linear solver

Because the linear solvers are embedded in a nonlinear scheme, the accuracy required for the linear solvers may influence the convergence of the Newton method. In the simulations denoted by *Simulation 3* and by *Simulation 4*, the free parameters are the stopping criterion for the Krylov solvers and the number of subdomains. The main parameters of these two simulations are summarized in Table 4.8.

	Simulation 3	*Simulation 4*
Frozen parameters		
Mesh	Mesh S	Mesh M
Max Krylov its.	200	200
SPD systems	CG	CG
Unsymmetric systems	GMRES	GMRES
GMRES precond.	right	right
Scaling	diagonal	diagonal
Preconditioner	M_{AS}	M_{AS}
Free parameters		
Linear solver stopping criterion	$10^{-5}, \ldots, 10^{-15}$	
Number of subdomains	8, 16 or 32	

Table 4.8: Parameters of *Simulation 3* and *Simulation 4*.

In Table 4.9, we depict the number of Newton steps needed to obtain the steady state for simulations *Simulation 3* and *Simulation 4*. Looking at the rows of Table 4.9, we see that the number of Newton steps increases when the threshold used for the stopping criterion of the Krylov solver is relaxed. With $\epsilon_{Krylov} = 10^{-5}$ and 16 or 32 subdomains the nonlinear scheme does not converge any more. This

Simulation 3

	ϵ_{Krylov}					
Nb of Subdomains	10^{-5}	10^{-7}	10^{-9}	10^{-11}	10^{-13}	10^{-15}
8	193	179	165	164	164	164
16	×	165	164	164	164	159
32	×	189	170	166	159	162

Simulation 4

	ϵ_{Krylov}					
Nb of subdomains	10^{-5}	10^{-7}	10^{-9}	10^{-11}	10^{-13}	10^{-15}
8	180	179	179	179	174	173
16	×	214	182	176	176	176
32	×	190	186	182	177	177

Table 4.9: Number of Newton steps needed to obtain the steady state for *Simulation 3* and *Simulation 4*. × means that the nonlinear scheme does not converge.

degradation can be explained by the inequality :

$$\| \text{ Forward error } \| < \text{ Condition number} \times \text{ Backward error.} \tag{4.4}$$

In Inequality (4.4) the forward error is the distance between the computed solution and the exact solution of the problem in a certain norm $\| \, . \, \|$, the condition number is the quantity that indicates the sensitivity of the problem to perturbations on its data measured in the same norm $\| \, . \, \|$ and the backward error is the distance between the problem to solve and the problem solved by the algorithm [29, 112]. As explained in Section 3.5, the stopping criterion for the Krylov solvers is based on backward error analysis. When the stopping criterion for the Krylov solver is relaxed, the backward error for the Schur complement system increases and the forward error on the solution at the interface might increase as well. The degradation of the solution accuracy at the interface propagates to the whole domain, and finally perturbs the convergence of the Newton and the Euler schemes.

In the columns of Table 4.9, for the same problem and with a constant required accuracy, the number of Newton steps generally increases with the number of subdomains. When the number of subdomains of the decomposition is increased, the size of the interface becomes larger and generally the condition number of the Schur complement increases as well. Inequality (4.4) indicates that again the forward error might increase. For example, for $\epsilon_{Krylov} = 10^{-5}$ in *Simulation 3* and in *Simulation 4*, the steady state is obtained for a decomposition in 8 subdomains but is not obtained for a decomposition in 16 or 32 subdomains. However Inequality (4.4) only gives an *upper bound* on the norm of the forward error that might not be sharp. For example, for *Simulation 4* with an accuracy of 10^{-7}, we can observe the opposite phenomenon. The number of Newton steps is smaller in the case of a decomposition into 32 subdomains than in the case of a decomposition into 16 subdomains.

If the accuracy required for the Krylov solver is relaxed, then the number of

Newton steps increases. At the same time, the number of iterations of the Krylov solvers decreases. So, each Newton step becomes less expensive and the overall simulation might become less computationally expensive. We further investigate this aspect in Section 4.2. Finally, we refer to [76] for more information on inexact Newton solvers.

4.1.5 Numerical scalability of the preconditioners

We now illustrate the numerical behaviour of the Krylov solvers when the number of subdomains increases. The preconditioners tested are the two local block preconditioners M_{bJ} and M_{AS} and the two two-level preconditioners M_{AS-sub} and $M_{AS-edge}$. These preconditioners were presented in Sections 3.2 and 3.3. The numerical experiments are performed on the meshes *Mesh S* and *Mesh M* decomposed into 8 or 16 subdomains using *Simulation 5*. The characteristics of this simulation are summarized in Table 4.10.

Frozen parameters	
Scaling	diagonal on S
ϵ_{Krylov}	10^{-11}
Max Krylov its.	200
SPD Systems	CG
Unsymmetric Systems	GMRES
Free parameters	
Mesh	Mesh S or Mesh M
Number of subdomains	8 or 16
Preconditioner	M_{bJ}, M_{AS}, M_{AS-sub} or $M_{AS-edge}$

Table 4.10: Parameters of *Simulation 5*.

Table 4.11 presents the number of Newton steps needed to obtain the steady state for *Simulation 5*. The results obtained with a direct method are also given as a reference. We see that, except in one case, the choice of the preconditioner does not really influence the nonlinear convergence. The only significant degradation of the nonlinear convergence is obtained with the preconditioner M_{AS-sub} on *Mesh M* decomposed in 16 subdomains. We also see that, in any case, direct methods always enable the fastest nonlinear convergence.

Table 4.12 presents the average number of GMRES iterations needed to solve each unsymmetric system in *Simulation 5*. With the two local preconditioners M_{AS} and M_{bJ}, the number of GMRES iterations increases when we double the number of subdomains. This increase could be explained by the absence of a global coupling mechanism as described in Section 3.3. However, the two coarse grid correction mechanisms introduced in M_{AS-sub} and $M_{AS-edge}$ are not efficient for this application. Even worse, they deteriorate the convergence of GMRES and sometimes the convergence of the nonlinear schemes (see Table 4.11).

	Mesh S 8 subdomains	Mesh S 16 subdomains	Mesh M 8 subdomains	Mesh M 16 subdomains
M_{bJ}	164	164	179	182
M_{AS}	164	164	179	176
M_{AS-sub}	166	166	182	193
$M_{AS-edge}$	166	166	182	182
Direct method	159	159	173	173

Table 4.11: Number of Newton steps needed to obtain the steady state in the case of *Simulation 5*.

	Mesh S 8 subdomains	Mesh S 16 subdomains	Mesh M 8 subdomains	Mesh M 16 subdomains
M_{bJ}	16	36	19	34
M_{AS}	10	18	14	24
M_{AS-sub}	20	34	23	44
$M_{AS-edge}$	18	29	22	38

Table 4.12: Average number of GMRES iterations needed to solve an unsymmetric linear system in the case of *Simulation 5*.

The SPD linear systems arising from the discretization of the semiconductor problem can be split into two different sets. The first set corresponds to the SPD systems arising from the discretization of the Poisson equation for the electrostatic potential. The second set corresponds to the SPD systems used to compute an initial state for the Euler schemes which are linearizations of the two transport equations (see Section 1.3.3). Table 4.13 presents the average number of CG iterations needed to compute the solution of each SPD linear system of the first set (also referred to as *Poisson systems*) and of the second set (also referred to as *Init systems*). The preconditioners are M_{bJ}, M_{AS}, M_{AS-sub}, $M_{AS-edge}$ and Balanced Neumann-Neumann (described in Section 3.3.2). The local Neumann-Neumann preconditioner is denoted by M_{NN} and $M_{BNN(k)}$ denotes the Balanced Neumann-Neumann preconditioner with a coarse space built using k degrees of freedom per subdomain.

In Table 4.13, for the M_{bJ} preconditioner the number of iterations of CG is almost twice as large when the number of subdomains is doubled; for the *Init* systems, a growth factor between 3 and 4 can even be observed. With the M_{AS} preconditioner, the number of iterations is smaller than with the M_{bJ} preconditioner. When the number of subdomains is doubled, an increase in the number of iterations can still be observed for *Poisson* or *Init* systems. In this latter situation, the factor is between 1.5 and 2.

Similarly to the unsymmetric case, the two two-level preconditioners M_{AS-sub} and $M_{AS-edge}$ are not efficient and even deteriorate the convergence of CG.

The local Neumann-Neumann preconditioner gives results comparable to the results obtained with the M_{AS} preconditioner for *Poisson* systems but is less efficient

Poisson systems

	Mesh S 8 subdomains	Mesh S 16 subdomains	Mesh M 8 subdomains	Mesh M 16 subdomains
M_{bJ}	22	38	17	35
M_{AS}	16	24	12	23
M_{AS-sub}	23	32	17	31
$M_{AS-edge}$	22	31	17	30
M_{NN}	17	22	15	22
$M_{BNN(1)}$	16	19	14	21
$M_{BNN(3)}$	15	18	14	19

Init systems

	Mesh S 8 subdomains	Mesh S 16 subdomains	Mesh M 8 subdomains	Mesh M 16 subdomains
M_{bJ}	18	79	14	46
M_{AS}	10	19	12	28
M_{AS-sub}	23	60	38	121
$M_{AS-edge}$	22	52	36	105
M_{NN}	25	50	34	72
$M_{BNN(1)}$	27	51	33	68
$M_{BNN(3)}$	27	36	27	48

Table 4.13: Average number of CG iterations needed to solve one SPD linear system of the *Poisson* or the *Init* type in the case of *Simulation 5*.

for *Init* systems. We can still observe a significant increase in the iteration number when the number of subdomains is doubled. Two-level preconditioners based on M_{NN} are $M_{BNN(1)}$ and $M_{BNN(3)}$. The preconditioner $M_{BNN(1)}$ has a coarse space with one degree of freedom per subdomain and $M_{BNN(3)}$ has a coarse space with three degrees of freedom per subdomain. We can observe an improvement of the behaviour of M_{NN} when $M_{BNN(1)}$ and $M_{BNN(3)}$ are used (except for *Init* systems with a decomposition into 8 or 16 subdomains of *Mesh S*). This improvement is almost negligeable for *Poisson* systems but is noticeable for *Init* systems especially when using $M_{BNN(3)}$.

Whatever the two-level preconditioner is, a significant increase in the number of iterations of CG when the number of subdomains is doubled can be observed in Table 4.13. For the *Init* systems, the local preconditioner M_{AS} is more efficient than the two-level preconditioner $M_{BNN(3)}$. Regarding the *Poisson* systems, $M_{BNN(3)}$ is numerically more efficient than M_{AS}. For $M_{BNN(3)}$, three eigenvectors associated with the smallest eigenvalues in modulus have to be computed for all the local Schur complement matrices. These calculations may become prohibitive concerning computational time. *Mesh S* decomposed into 16 subdomains is the simulation where the gap between M_{AS} and M_{BNN} is the largest. However, in this case the average time needed to solve a *Poisson* Schur system is roughly 10% longer when $M_{BNN(1)}$ or $M_{BNN(3)}$ is used than when M_{AS} is used even if the number of iterations is less important (24 with M_{AS}, 19 with $M_{BNN(1)}$ and 18 with $M_{BNN(3)}$). More details concerning the comparison between M_{AS} and M_{BNN} for *Poisson* systems can be found in [61].

To conclude, we would like to underline the fact that neither the two-level block preconditioners presented in Section 3.3.3, nor the Balanced Neumann-Neumann algorithm presented in Section 3.3.2 have been able to ensure the numerical scalability of the Krylov solvers in the case of our semiconductor application. In [28], a set of experiments shows that in the case of an anisotropic problem, the effect of the coarse grid component may only be visible for a large number of subdomains. We may remark that our experiments are similar. Our meshes present anisotropy combined with the high variation of the functions N and P involved in system (1.2) and displayed in Figure 1.6. Furthermore, the number of subdomains considered remains relatively small (≤ 16) and this fact may hide the effect of the coarse grid similarly to what is reported in [28].

4.1.6 Conclusion

These experiments highlight that the semiconductor device modelling problems are numerically difficult to solve. When iterative substructuring methods are used, the combination of scaling and preconditioning is needed to solve efficiently the Schur complement systems in such a way that enables the nonlinear solver to converge. The numerical complexity of embedded iterations is also illustrated by these experiments. The results show that the convergence of the Newton methods and of the Euler scheme can be very sensitive to the choice of the parameters governing the linear

solver. In particular, the experiments revealed that preconditioning is essential to make the inner linear systems converge and that scaling is essential to ensure the convergence of the outer nonlinear schemes.

Moreover, one can see that the transition from model problems to complex applications is not an easy task. The application of multi-level preconditioners was not conclusive. Coarse grid mechanisms that were demonstrated efficient for recognized difficult elliptic problems [28] even deteriorate the convergence of the Krylov solvers in our application. The well known Balanced Neumann-Neumann preconditioner did not show its usual robustness and efficiency for this application compared to its performance on difficult problems arising in structural analysis [104, 105].

Finally, in the case of iterative substructuring, the combination of the M_{AS} local block preconditioner and of the diagonal scaling on the Schur complement has been identified as the more robust from a numerical point of view.

4.2 Performance of iterative substructuring and direct solvers

This section is devoted to the presentation and analysis of the parallel performance of the implemented linear solvers. In Section 4.2.1, we compare on a model problem the computational cost of the explicit and the implicit iterative substructuring algorithms. From Section 4.2.2 to 4.2.5, we compare for selected semiconductor applications the performance of the parallel iterative and direct solvers presented in Chapters 2 and 3. In Section 4.2.2, we present the selected test cases and the influence of the hardware and software environments on time measurements. Then we present the results observed for iterative substructuring in Section 4.2.3 and for parallel direct methods in Section 4.2.4. Finally, we make the comparison between iterative and direct solvers in Section 4.2.5.

4.2.1 Implicit versus explicit iterative substructuring

The difference between these two methods lies in the fact that, in one case, the local Schur complements are not computed (implicit case) while, in the other case, they are explicitly formed (explicit case). These two methods are described in Section 2.4.

We illustrate the differences between these two methods in terms of computing effort on a model problem. This model problem consists of the discretized Poisson equation on a regular rectangular grid. The equation is the following

$$\begin{cases} -\dfrac{\partial^2 u}{\partial x^2} - \dfrac{\partial^2 u}{\partial y^2} = f \text{ in } \Omega \subset \mathbb{R}^2, \\ u = g \text{ on } \partial\Omega, \end{cases} \tag{4.5}$$

where the domain Ω is a rectangle discretized by a uniform grid and decomposed in regular boxes (see Figure 4.1). Using a Schur complement technique, the problem

is then reduced to the solution of a symmetric positive definite linear system by CG iterations on the interface variables. Each CG iteration requires a matrix-vector product that can be done explicitly if the local Schur complement matrices are explicitly computed or is performed implicitly otherwise. This 2D problem is similar to semiconductor problems concerning the ratio between the number of unknowns belonging to the interface and the number of unknowns belonging to the interior of the subdomains as both problems are two-dimensional.

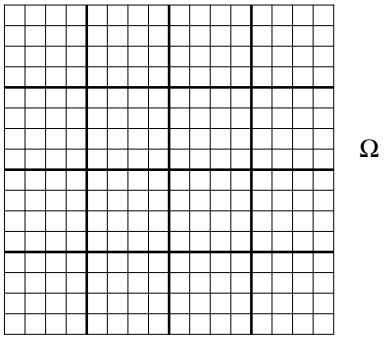

Figure 4.1: Regular discretization and decomposition into regular subdomains of a rectangular domain Ω (4×4 decomposition).

In Table 4.14, we present the elapsed time required by MUMPS for the factorization of the internal subproblems, for the factorization and construction of the local Schur matrices, in the implicit and the explicit case respectively. In the explicit case, the local Schur complements are computed in addition to the factorizations of the internal subproblems. As expected, we see that this time is longer in the explicit

	Size of the subdomains							
Number of	100		200		400		800	
subdomains	Impl	Expl	Impl	Expl	Impl	Expl	Impl	Expl
2×2 decomp.	0.35	0.38	1.69	1.97	9.23	11.74	61.49	113.3
2×4 decomp.	0.34	0.43	1.74	2.66	9.41	14.62	67.73	113.8
4×4 decomp.	0.38	0.43	1.94	3.39	10.19	18.38	70.36	119.23

Table 4.14: Elapsed time (in seconds) for the factorization of the local Dirichlet problems in the case of implicit or explicit iterative substructuring algorithms.

situation. It is due to the additional cost for building the local Schur complement during the factorization. This additional cost remains small for the smaller test case (4 subdomains of size 100×100) but grows up to 40% for the biggest one (16 subdomains of size 800×800). In the case of a decomposition into 16 subdomains, the local Schur complements corresponding to internal subdomains (that means without any element belonging to $\partial\Omega$) are bigger than the local Schur complements associated with subdomains which intercept the boundary $\partial\Omega$. So the cost of the factorization is larger for a decomposition into 16 subdomains (i.e. 4×4) than for a decomposition

into 4 subdomains (i.e. 2×2) with the same domain size. This effect is less visible for subdomains of size 800×800 because in this case the ratio between the number of interface points and the number of internal points is smaller than for smaller subdomains. The time required by MUMPS in the implicit situation is less sensitive to the number of subdomains and actually should remain constant as the size of the subdomains remains constant (i.e. reading Table 4.14 column by column).

In Table 4.15, we present the elapsed time needed to perform one matrix-vector product by the Schur complement in the implicit and the explicit case. Using the

	Size of the subdomains							
Number of	100		200		400		800	
subdomains	Impl	Expl	Impl	Expl	Impl	Expl	Impl	Expl
2×2	61	0.58	280	1.5	1270	12	6320	81
2×4	55	0.71	280	6.0	1430	36	7570	190
4×4	70	1.8	370	13	1590	71	7610	460

Table 4.15: Elapsed time (in milliseconds) to compute a matrix-vector product by the Schur complement in the implicit and explicit case.

explicit method reduces the time spent in the matrix-vector product by a factor of between 15 and 25 compared to the implicit scheme. In the implicit case, two matrix-vector products by the sparse coupling blocks and a forward/backward substitution using the sparse factors of the internal problem are needed locally on each subdomain to compute the matrix-vector product by the local Schur complement. The matrix-vector product by the complete Schur matrix is then obtained by a communication phase between neighbouring subdomains. In the explicit case this communication phase is still the same, but the computation of the local matrix-vector product by the local Schur complement is obtained by a simple call to a level 2 BLAS subroutine that implements a dense matrix-vector product. In this latter case, the number of floating point operations is smaller and the access to the memory is more regular (i.e. dense versus sparse calculation), this explains the large reduction of computing time observed when using the explicit matrix-vector product.

Except during the factorization and the matrix-vector product, the operations performed by CG iterations using either the implicit or the explicit algorithms are identical. It is then enough to consider the computational time of these two kernels to evaluate their possible advantages when used in Krylov iterations. In order to make this comparison, we define the amortization ratio k which is defined to be the number of matrix-vector products needed to compensate the additional cost of the computation of the local Schur matrices when the explicit method is used. The values of the amortization ratio for a decomposition into 4, 8 or 16 subdomains of dimensions 100, 200, 400 and 800 are depicted in Table 4.16. In this table, we see that less than 10 matrix-vector products are generally enough for the explicit algorithm to be more efficient than the implicit one. That means that when CG requires more than 10 iterations to converge the implementation based on the explicit computation

Number of subdomains	Size of the subdomains 100	200	400	800
4	8	2	2	9
8	2	4	4	7
16	6	5	6	7

Table 4.16: Amortization ratio to compensate the explicit Schur complement computations by the gain in matrix-vector products.

of the Schur complement outperforms the one based on the implicit calculation. This observation is still valid for unsymmetric systems, solved for instance using GMRES iterations as GMRES only requires one matrix-vector product per iteration.

Furthermore, we have not considered the problem of preconditioning. As described in Section 3.2.2, the preconditioners that we have studied are easier to build when the explicit method is used. The main drawback of the explicit case is the additional memory cost due to the storage of the dense local Schur complement matrices. However for 2D computations this extra cost remains affordable.

In all the experiments on semiconductor devices, we only consider iterative substructuring methods that implement the explicit computation of the local Schur complement matrices.

4.2.2 Description of the test examples

Computer environment

In this section, we present timing and performance measurements for the methods presented in the previous chapters and sections. All these experiments have been performed on a SGI O2000 NUMA platform installed at CINES (Centre Informatique National de l'Enseignement Supérieur) or Parallab. Time measurements on this class of architecture may fluctuate between two identical runs as the mapping of the data in memory can vary from one run to another. A relative variation of about 10 % in the timing should be taken into account.

Frozen parameters and notations

In order to investigate the parallel efficiency of the linear solver implementation we consider three simulations denoted by *Medium 8*, *Medium 16* and *Large 32*. These simulations are based on the meshes *Mesh M* and *Mesh L* defined in Section 4.1. In simulation *Medium 8*, *Mesh M* is decomposed into 8 subdomains and in *Medium 16*, it is decomposed into 16 subdomains (see Tables 4.1 and 4.2). In *Large 32*, *Mesh L* is decomposed into 32 subdomains (see Table 4.3). The main parameters of these simulations are summarized in Tables 4.17, 4.18 and 4.19. The methods used to solve the linear systems are iterative substructuring, direct substructuring or the multifrontal method with the input matrix distributed.

General parameters	
Mesh	Mesh M
Nb of subdomains	8
Parameters for iterative substructuring	
ϵ_{Krylov}	10^{-11}
Max Krylov its.	200
SPD systems	CG
Unsymmetric systems	GMRES
Scaling	Diagonal on S
Preconditioner	M_{bJ} or M_{AS}

Table 4.17: Parameters of the simulation *Medium 8*.

General parameters	
Mesh	Mesh M
Nb of subdomains	16
Parameters for iterative substructuring	
ϵ_{Krylov}	10^{-11}
Max Krylov its.	200
SPD systems	CG
Unsymmetric systems	GMRES
Scaling	Diagonal on S
Preconditioner	M_{bJ} or M_{AS}

Table 4.18: Parameters of the simulation *Medium 16*.

General parameters	
Mesh	Mesh L
Nb of subdomains	32
Parameters for iterative substructuring	
ϵ_{Krylov}	10^{-11}
Max Krylov its.	200
SPD systems	CG
Unsymmetric systems	GMRES
Scaling	Diagonal on S
Preconditioner	M_{bJ} or M_{AS}

Table 4.19: Parameters of the simulation *Large 32*.

For the iterative method, symmetric diagonal scaling is applied to the complete Schur complement system. The preconditioners used are the two local block preconditioners M_{AS} and M_{bJ}. The choice of the preconditioner defines two iterative substructuring algorithms. These two algorithms are referred to as M_{bJ} and M_{AS} by reference to the preconditioner they use. Concerning direct substructuring, this is an implementation of the algorithm described in Section 2.4.4 and we refer to it as *Dss*. The distributed multifrontal algorithm is mainly a call to the software MUMPS (see Section 2.3.3) with the distributed entries option. We refer to this method as *Sparse Direct*.

A remark on the MUMPS symbolic analysis

If the method selected is *Sparse Direct*, the computation of the solution is performed in three steps : the symbolic analysis, the numerical factorization and the solve (see Section 2.3). The sparsity pattern is the same for all the matrices because the mesh does not change during the simulation. Therefore, only two symbolic analysis phases are needed, one for all the SPD systems and one for all the unsymmetric systems (see Section 2.3). Practically, due to an unresolved memory leak problem either in MUMPS or in the coupling between MUMPS and the semiconductor device simulation code, the data structure of MUMPS must be cleaned between two Newton iterations. Therefore, the symbolic analysis phase has to be done for each linear system. This requirement is also valid for each MUMPS instance used in the substructuring algorithms. The elapsed time that would have been obtained if only two symbolic analysis were performed can easily be extrapolated from the results. We have chosen to give the extrapolated results to allow a fair comparison between iterative and direct methods.

4.2.3 Results observed with the iterative substructuring algorithms

In this part, we present a comparison between the two iterative substructuring methods M_{AS} and M_{bJ} on simulations *Medium 8*, *Medium 16* and *Large 32*.

The measured quantities are either numerical quantities without dimension or elapsed time measurements in seconds. The numerical quantities are the total number of Newton steps, that can be decomposed into the number of steps that involve SPD linear systems and the number of steps that involve unsymmetric systems, the average number of CG iterations needed to solve the SPD Schur systems and the average number of full GMRES iterations needed to solve the unsymmetric Schur systems. Concerning time measurements, we first measure the total elapsed time needed to perform the simulation (denoted by *Total simul*) and the total elapsed time spent in the linear solvers during a simulation (denoted by *Total* $Ax = b$). The results are presented in the Tables 4.20, 4.21 and 4.22.

From Tables 4.20 and 4.21 it is difficult to tell whether M_{bJ} or M_{AS} is the best. The number of Newton steps is almost the same. The average number of Krylov

	M_{bJ}	M_{AS}
Newton steps	179	179
SPD systems	73	73
Unsymmetric systems	106	106
Iter CG	15	12
Iter GMRES	19	14
Global Times		
Total $Ax = b$	976	921
Total simul	1361	1295

Table 4.20: Results for *Medium 8* with M_{bJ} and M_{AS}.

	M_{bJ}	M_{AS}
Newton steps	182	176
SPD systems	76	70
Unsym systems	106	106
Iter CG	38	24
Iter GMRES	34	24
Global times		
Total $Ax = b$	582	582
Total simul	788	809

Table 4.21: Results for *Medium 16* with M_{bJ} and M_{AS}.

	M_{bJ}	M_{AS}
Newton steps	228	175
SPD systems	76	76
Unsymmetric systems	152	99
iter CG	68	32
iter GMRES	94	34
Global times		
Total $Ax = b$	2433	1286
Total simul	2892	1654

Table 4.22: Results for *Large 32* with M_{bJ} and M_{AS}.

iterations per linear system is larger with M_{bJ} than with M_{AS}. So the time spent in matrix-vector products and in dot products is larger in the case of M_{bJ}. On the other hand, the cost of construction and application of the preconditioner is larger for M_{AS}. For *Medium 16* the average time needed to build the M_{AS} preconditioner is 0.26 seconds while the average time needed to build M_{bJ} is 0.05 seconds. The average time needed to apply the preconditioner M_{bJ} is 2.0 milliseconds while the average time needed for M_{AS} is 8.3 milliseconds. Finally, if we take into account the variability in the time measurements, we can say that for simulations *Medium 8* and *Medium 16*, M_{AS} and M_{bJ} are equivalent. M_{AS} is more expensive but enables a faster convergence of the linear solvers than M_{bJ} that is cheaper in computation but less numerically efficient.

However, if we turn to *Large 32*, some differences appear. With M_{bJ} the number of Newton steps is larger than with M_{AS}, about 30 % more iterations. Moreover, the linear convergence is also deteriorated. Concerning the average number of Krylov iterations needed to solve a linear system, the gap between M_{bJ} and M_{AS} is much larger than in the case of *Medium 8* and *Medium 16*. The larger number of matrix-vector products is no longer mitigated by the cheaper construction and application of the M_{bJ} preconditioner. So the preconditioner M_{AS} is more robust than the preconditioner M_{bJ} and leads to better results. Therefore, in all the following experiments, if it is not stated explicitly, the preconditoner used is M_{AS}.

We display, in Table 4.23, the average elapsed time spent in sparse computations and in solving iteratively the Schur system with the M_{AS} preconditioner. By sparse computations we mean the symbolic analysis, the numerical factorization and the solve using MUMPS sequentially but concurrently on each subdomain. The symbolic analysis can be reused from one Newton step to another. In Table 4.23 we display the time spent in the analysis for only one Newton step. We point out that the total time spent in the symbolic analysis during a simulation becomes almost negligible if the number of Newton steps is large as is usually the case in a complete semiconductor simulation. For the iterative solution of the Schur complement system, we also take into account the construction of the M_{AS} preconditioner.

	Medium 8	*Medium 16*	*Large 32*
Symbolic analysis	0.82	0.38	0.83
Factorization	2.67	1.43	2.87
Solve	0.40	0.29	0.58
Iterative solution of the Schur complement system	0.89	0.77	2.13

Table 4.23: Average time per linear system for the sparse computations on the subdomains and in the iterative solution of the Schur system preconditioned by M_{AS} and using explicit matrix-vector products.

We can see, in Table 4.23, that the time spent in sparse computations is at least twice as large as the time spent in solving iteratively the Schur complement

system. We recall that the symbolic analysis phase can be reused from one Newton step to another and that its influence on the total simulation time decreases if the number of Newton steps increases. Nevertheless, the factorization phase is longer than the solution of the Schur system. The factorization of the internal problems has to be computed for each Newton step. The high cost of the factorization can be explained by the choice of the explicit iterative substructuring method. In this case, the matrix-vector products are very efficient and the relatively large number of Krylov iterations justifies the choice of the explicit computation of the local Schur complements. In simulation *Medium 8*, the linear systems have 356701 unknowns for a decomposition into 8 subdomains and an interface with 1607 unknowns. We can make a comparison with the model problem used in Section 4.2.1 to justify the choice of the explicit iterative substructuring. With 8 subdomains of size 200×200 we have a model problem with 320000 unknowns and an interface of 2000 unknowns. Therefore these two problems are of comparable dimensions. We can see in Table 4.16 that, in the latter case, only 4 iterations of the Krylov solver were enough to make the explicit algorithm outperform the implicit one. Now, if we look at the results of Table 4.20 we see that the average number of CG iterations needed to solve each SPD Schur system is 12 and the average number of GMRES iterations needed to solve each unsymmetric Schur system is 14. Therefore we can justify the use of the explicit algorithm for *Medium 8*. A similar approach can also justify the choice of the explicit algorithm for *Medium 16* and *Large 32*.

We recall that the number of iterations of the Krylov solvers increases when the number of subdomains is increased. For example, between *Medium 8* and *Medium 16*, the number of subdomains grows from 8 to 16 and the average number of Krylov iterations increases from 12 to 24 for CG and from 14 to 24 for GMRES (see Tables 4.20 and 4.21). Furthermore, the size of the Schur complement matrix also increases from 1607 to 2273 (see Tables 4.1 and 4.2). Paradoxically, we can see, in Table 4.23, that the average time to solve a Schur complement system decreases slightly. This can be explained by the fact that the subdomains become smaller. Consequently, the good scalability of the construction and application of the preconditioner as well as the parallel efficiency of the matrix-vector product lead to faster solutions. In Table 4.24, we display the average elapsed measured times for the construction and the application of the preconditioner and of the matrix-vector product for *Medium 8* and *Medium 16*. We see that the times needed for these three computational kernels decrease when the number of subdomains grows from 8 to 16.

	Medium 8	*Medium 16*
Build Precond	490	260
Matvec product	114	75
Apply Precond	159	83

Table 4.24: Average elapsed time (in milliseconds) for the construction of the M_{AS} preconditioner, for its application and for the matrix-vector product.

In Table 4.25, we present the average weights (in percentages) of these three main computational kernels of the Krylov solver for the solution of one Schur complement system with the M_{AS} preconditioner. We also take into account the percentage of time spent in the dot product procedure. We only neglect SAXPY-like operations that are extremely cheap. In *Medium 8* the number of Krylov iterations is relatively small (see Table 4.20). In this case the cost of the construction of the preconditioner dominates the solution time. When the number of iterations increase (*Medium 16* or *Large 32*) the cost of the iterative solution becomes the most time consuming part of the computation. The time spent in the dot product is relatively large due to the choice of the orthogonalization scheme selected in GMRES (we will investigate this in more detail later in Section 4.3).

	Medium 8	*Medium 16*	*Large 32*
Build Precond	55	34	27
Matvec products	17	23	24
Dot products	4	17	22
Apply Precond	24	26	27

Table 4.25: Percentage of time spent in the construction of the M_{AS} preconditioner, in its application, in the matrix-vector product and in the dot product during the iterative solution of one Schur system.

In Section 4.1.4, we have seen that if the accuracy required for the Krylov solvers is relaxed then the number of Newton steps increases. At the same time the number of Krylov iterations needed to solve each Schur complement system decreases. In this section, we present results that indicate that the time spent in the sparse factorization of the internal subdomains is longer than the iterative solution of the Schur complement. Therefore if the accuracy is relaxed and if the number of Newton steps increases, the overall simulation time will generally increase. On the other hand, if the accuracy required is too small, useless Krylov iterations will be performed. So the difficulty is to find a good trade-off between a fast convergence of the linear solvers and a fast convergence of the nonlinear schemes.

The comparison of the preconditioners M_{bJ} and M_{AS} in Tables 4.20, 4.21 and 4.22 illustrates again the importance of the robustness of the iterative linear solver. Even if M_{AS} and M_{bJ} are equivalent in many cases concerning computational time, the choice of M_{AS} ensures better condition number for the preconditioned matrix and therefore a better stability of the nonlinear schemes. In that respect M_{AS} is much more robust than M_{bJ}.

4.2.4 Results observed with parallel direct methods

In this section, we analyse and compare the results obtained with the two parallel direct methods *Dss* and *Sparse Direct* for the simulations *Medium 8*, *Medium 16* and *Large 32*.

The number of Newton steps is decomposed into the steps that involve SPD linear systems and the steps that involve unsymmetric linear systems. The elapsed times (in seconds) measured are the time needed to perform the simulation (denoted by *Total simul*) and the total time spent in the linear solvers (denoted by *Total* $Ax = b$). Then, the direct solution of one linear system can be split into three steps : the symbolic analysis, the numerical factorization and the solve. We measure the average time per linear system spent in these three steps (denoted by *Mumps symbolic analysis*, *Mumps factorization* and *Mumps solve*). These results are presented in Tables 4.26, 4.27 and 4.28. For *Dss*, we accumulate the symbolic analysis time spent for the internal subproblems and for the Schur complement given as a distributed matrix to MUMPS. A similar approach has been followed for the factorization and the solve times. For the sake of completness, in Table 4.29, we divide these accumulated times between what is spent for the internal subproblems (denoted by *Internal*) and for the Schur (denoted by *Schur*).

	Dss	*Sparse Direct*
Newton steps	173	173
SPD systems	67	67
Unsymmetric systems	106	106
Global times		
Total $Ax = b$	1272	1291
Total simul	1651	1701
Average times per linear system		
Mumps symbolic analysis	1.76	11.60
Mumps factorization	5.76	4.65
Mumps Solve	0.51	2.35

Table 4.26: Results of parallel direct methods for *Medium 8*.

	Dss	*Sparse Direct*
Newton steps	173	173
SPD systems	67	67
Unsymmetric systems	106	106
Global times		
Total $Ax = b$	941	1022
Total simul	1140	1228
Average times per linear system		
Mumps symbolic analysis	1.20	12.36
Mumps factorization	4.58	3.48
Mumps solve	0.31	2.12

Table 4.27: Results of parallel direct methods for *Medium 16*.

	Dss	*Sparse Direct*
Newton steps	166	166
SPD systems	68	68
Unsymmetric systems	98	98
Global times		
Total $Ax = b$	2110	3620
Total simul	2527	3995
Average times per linear system		
Mumps symbolic analysis	3.75	46.25
Mumps factorization	10.50	11.71
Mumps solve	0.6	9.22

Table 4.28: Results of parallel direct methods for *Large 32*.

	Medium 8		*Medium 16*		*Large 32*	
	Internal	Schur	Internal	Schur	Internal	Schur
Symbolic analysis	0.90	0.86	0.38	0.82	0.8	2.95
Factorization	2.92	2.84	1.35	3.23	3.30	7.20
Solve	0.47	0.04	0.24	0.07	0.48	0.12

Table 4.29: For *Dss*, elapsed times (in seconds) for the internal subproblems and the interface problem (Schur complement system).

MUMPS symbolic analysis. For the three test cases, the symbolic analysis time is higher with *Sparse Direct* than with *Dss*. In the case of *Sparse Direct*, the symbolic analysis is completely sequential in the current implementation of MUMPS [6]. In the case of *Dss*, the part of the symbolic analysis associated with the internal subproblems is performed concurrently on each subdomain. Only the symbolic analysis for the Schur complement remains centralized on one processor (again due to the current implementation of MUMPS). The symbolic analysis can be reused from one Newton step to another. Therefore the final gain will depend on the total number of Newton steps required by the simulation.

MUMPS Factorization. We can see in Tables 4.26 and 4.27 that the factorization time is longer for *Dss* than for *Sparse Direct* on *Medium 8* and *Medium 16*. On *Large 32*, it is the opposite (see Table 4.28). Nevertheless, in the three cases, the times are of the same magnitude. In Table 4.30, we display the total number of million of floating-point operations (flops) performed during the factorization for an SPD or an unsymmetric system. We can see that, in any case, the number of flops is larger with *Dss* than with *Sparse Direct*. For *Sparse Direct*, the ordering on which the factorization is based is an approximate minimum degree algorithm (AMD). For *Dss*, it is a combination of nested dissection-like (corresponding to the

	Medium 8		*Medium 16*		*Large 32*	
	Dss	*Sp. Direct*	*Dss*	*Sp. Direct*	*Dss*	*Sp. Direct*
SPD system	1062	814	1014	821	2939	2455
Unsymmetric system	2074	1581	1980	1595	5872	4901

Table 4.30: Total number of million floating-point operations performed during the factorization phase.

mesh partitioning) and minimum degree within the subdomains and at the root on the complete Schur complement (see Section 2.4.4). Usually, the nested dissection based orderings lead to a factorization with less floating point operations [5]. But the opposite phenomenon has been noticed for linear systems arising from an uniform discretization on a mesh with an unbalanced aspect ratio (that is a mesh which is much longer along one direction) [13]. Actually, the meshes used for semiconductor device simulations present some geometric anisotropy that induces an unbalanced aspect ratio in the graph of the associated matrix. This observation on our application might explain this particular behaviour of the *Sparse direct* algorithm.

The number of operations performed during the factorization is not necessarily the key parameter. It may not give a good indication of the factorization time. We see that, for *Large 32*, the factorization time is slightly longer with *Sparse Direct* even if the number of flops is smaller than with *Dss*. Another key aspect is the memory size required by the factors. In the case of *Dss*, it corresponds to the sum of the factors associated with the subdomains, the factors associated with the complete Schur matrix and the storage of the dense local Schur complement matrices (computed by *MUMPS* in the first step of the *Dss* method). These quantities are given in Megabytes and are shown in Table 4.31. We see that *Sparse Direct* requires

	Medium 8		*Medium 16*		*Large 32*	
	Dss	*Sp. Direct*	*Dss*	*Sp. Direct*	*Dss*	*Sp. Direct*
SPD system	117	91	117	91	356	288
Unsymmetric system	174	146	173	146	531	459

Table 4.31: Number of Megabytes occupied by the factors.

less memory than *Dss*. In our experiments, we have never exceeded the memory capacity of the SGI 02000 platform even for the biggest test case. So the difference of a few Megabytes per processor that can be observed in Table 4.31 is affordable as *Dss* is slightly faster than *Sparse Direct* (for the global simulation time).

These results show a slight advantage for *Sparse Direct* concerning the factorization phase.

MUMPS Solve. For the three simulations, the time spent in the solve is larger with *Sparse Direct* than with *Dss*. In Table 4.28, for the biggest test case, there is a factor of more than 15. The ordering induced by *Dss* gives better results for the solve phase than the *AMD* ordering. The choice of the ordering influences the

performance of the direct solvers. A nested dissection ordering option is currently under development for the next release of the MUMPS software. We have tested a preliminary version on some examples. We should mention that we have not been able to use this version for semiconductor problems because the distributed entries option was not compatible yet with the nested dissection ordering option. In Table 4.32 we display the elapsed times for the symbolic analysis, factorization and solve phase depending on the ordering for a model problem. This model problem is a 2D Poisson operator with a 5-point discretization. The size of the matrix is 300000 and the number of nonzeros is 1493800. We use 8 processors. These results illustrate the influence of the ordering on each step of MUMPS. The factorization and the solve phase are faster when a nested dissection ordering is used. It can

	AMD ordering	ND ordering
Symbolic analysis	5.48	4.07
Factorization	4.07	1.77
Solve	2.47	0.75

Table 4.32: Influence of the ordering for MUMPS on a model problem. Elapsed times in seconds for the symbolic analysis, the factorization, the solve.

be expected that the next release of MUMPS with the nested dissection ordering option will reduce the gap between the two direct approaches *Dss* and *Sparse Direct* for semiconductor applications.

Conclusion. Concerning the global simulation time, *Dss* is more efficient than *Sparse Direct* in the three simulations. For *Medium 8* and *Medium 16* the difference is small. There is a gain of approximately 3 % for *Large 32* and 8 % for *Medium 16*. If we take into account the uncertainty on time measurements, we can say that these two methods are equivalent. For *Large 32* the significant gain on the solve phase leads to a significant gain (about 37 %) on the global simulation time. As already mentioned, this gap might decrease in the future when the new release of MUMPS will support other orderings than AMD.

4.2.5 Comparison between iterative and direct substructuring algorithms

We now compare iterative and direct solvers on the three simulations : *Medium 8*, *Medium 16* and *Large 32*. In the case of iterative substructuring, the method that gives the most satisfactory results is M_{AS}. In the case of parallel direct solvers, the results of Section 4.2.4 indicate that *Dss* is more attractive than *Sparse Direct* for this application. Moreover, the choice of *Dss* makes the comparison easier with iterative substructuring. The only difference between M_{AS} and *Dss* is the way the complete Schur system is solved. In the first case, the Schur systems are solved by a distributed iterative Krylov solver and in the second case by a distributed direct multifrontal solver. In order to compare M_{AS} and *Dss* we synthesize some results of

Sections 4.2.3 and 4.2.4 in Table 4.33. These results are the total number of Newton steps to perform a simulation, the total elapsed time spent in solving linear systems, the elapsed time required to perform the simulation and the average elapsed time needed to solve one Schur complement system during the simulation.

	Medium 8		*Medium 16*		*Large 32*	
	M_{AS}	Dss	M_{AS}	Dss	M_{AS}	Dss
Newton steps	179	173	176	173	175	166
Total $Ax = b$	921	1272	582	941	1286	2110
Total simul	1295	1651	809	1140	1654	2527
Average Time for $Su = g$	0.89	2.89	0.77	3.31	2.13	7.34

Table 4.33: Comparison of iterative substructuring and direct substructuring. Total number of Newton steps, total elapsed time (in seconds) spent in the linear solvers and average elapsed time (in seconds) for solving one Schur complement system.

We can see that *Dss* requires less Newton steps than M_{AS} to obtain the steady state. On the other hand, the total time spent in the linear solver is larger with *Dss* than with M_{AS}. For our applications, this is due to the fact that solving the distributed Schur complement with an iterative solver is much more efficient than solving it with a distributed direct solver.

In Table 4.34 we compare the scalability of direct and iterative substructuring when the number of subdomains of the decomposition is doubled. We can see that for the two methods the speed up remains lower than two. Concerning iterative methods this can be explained by the increase in the number of Krylov iterations (see Section 4.2.3). In the case of direct substructuring, the speed-up is good for the internal problem but not for the interface problem (see Table 4.29). The interface problem is not sparse and large enough to ensure a good speed-up.

	Medium 8	*Medium 16*	*Speed up*
M_{AS}	1295	809	1.6
Dss	1701	1140	1.5

Table 4.34: Scalability of direct and iterative substructuring methods.

Concerning memory requirements, we do not have to store the factors of the Schur matrix but we have to store the preconditioner M_{AS} which requires the same memory as the local Schur complements. For M_{AS}, we consider the memory space occupied by the factors of the internal subproblems, the local Schur complements and the preconditioner. For *Dss*, we consider the memory space occupied by the factors of the internal subproblems, the Schur complement matrix S and the factors of S. These quantities are presented in Table 4.35.

One can see that the difference between the two methods are not significant. Actually, the main part of the memory used by the two methods is required by the

SPD system

	Medium 8		*Medium 16*		*Large 32*	
	M_{AS}	Dss	M_{AS}	Dss	M_{AS}	Dss
Internal Factors	99		96		304	
Local Schur complements	11		11		30	
M_{AS} Preconditioner	11	×	11	×	30	×
Factors of S	×	7	×	10	×	22
Total	121	117	118	117	364	356

Unsymmetric system

	Medium 8		*Medium 16*		*Large 32*	
	AS	Dss	AS	Dss	AS	Dss
Internal Factors	152		148		468	
Local Schur complements	11		11		30	
M_{AS} Preconditioner	11	×	11	×	30	×
Factors of S	×	10	×	14	×	33
Total	175	174	171	173	528	531

Table 4.35: Comparison of the memory space (in megabytes) used by iterative or direct substructuring.

factors of the internal subproblems and by the local Schur complements. We see that the size occupied by M_{AS} is larger than the size occupied by the factors of S in the SPD case. This is due to the fact that for software reasons we do not exploit the symmetry of M_{AS}. In the unsymmetric case the size occupied by the factors of S is larger than M_{AS} for *Large 32* and *Medium 16* but not for *Medium 8*. We can make two remarks. First, the size of the factors of S remains relatively small compared to the size of the local Schur complement matrices. So in our test cases there is not a lot of fill-in when factorizing the complete Schur matrix. Secondly, the M_{AS} preconditioner is expensive in terms of memory cost. In Section 5.1, we present a sparsification procedure that enables us to reduce the memory required by the M_{AS} preconditioner.

We would like to conclude by emphasizing that, in our context, iterative solvers are more efficient than direct ones but the tuning of the parameters that govern their behaviour might be difficult. In Section 4.1, we have presented several numerical difficulties encountered during the development of iterative solvers. Some others are presented in Section 4.3. On the other hand, the use of MUMPS as a black box direct solver is the least efficient method concerning computational time but it was the easiest to interface with the semiconductor code. The direct substructuring algorithm requires more effort but future versions of MUMPS will integrate new features that should reduce the gap between the two direct approaches in terms of performance (i.e. parallel elapsed time).

4.3 A posteriori justification of some choices

For the sake of simplicity of the presentation, some choices made in Sections 4.1 and 4.2 have not been argued. These choices are : the choice of GMRES as a Krylov solver for the unsymmetric systems (Section 4.3.1), the location of the preconditioner (right and left) for GMRES (Section 4.3.2), the choice of the orthogonalization scheme within GMRES (Section 4.3.3) and the use of GMRES without restart (Section 4.3.4).

4.3.1 Selection of GMRES as unsymmetric Krylov solver

Let

$$Su = g$$

be the Schur complement system to be solved. We have selected GMRES as the Krylov solver to solve the unsymmetric Schur complement systems. Other possibilities as BiCGStab also exist (see Section 2.4.3). We have compared on *Simulation 6* the behaviour of the two algorithms. The main parameters of this simulation are summarized in Table 4.36. The implementation of BiCGStab is a variant of the rou-

Mesh	Mesh L
Number of subdomains	32
ϵ_{Krylov}	Variable
Max It Krylov	200
SPD systems	Conjugate Gradient
Unsymmetric systems	GMRES or BiCGStab
Preconditioner	M_{AS}
Scaling	Diagonal on S

Table 4.36: Parameters of *Simulation 6*.

tine MI06 from the "Harwell Subroutine Library" [74]. We consider two stopping criteria for BiCGStab. The first one is

$$\frac{\| r_n \|}{\| g \|} \tag{4.6}$$

where r_n is the residual computed by the internal recurrences of BiCGStab at step n. The second one is

$$\frac{\| Su_n - g \|}{\| g \|} \tag{4.7}$$

where $Su_n - g$ is the true residual at step n. In exact arithmetic (4.6) and (4.7) are identical but in finite precision arithmetic they might differ. The stopping criterion for GMRES is (4.7). For BiCGStab we performed experiments with either (4.6) or (4.7). For software reasons, if (4.7) is chosen the algorithm will be very slow because the true residual is computed at each step.

Krylov solver	GMRES	BiCGStab	BiCGStab
ϵ_{Krylov}	10^{-11}	10^{-11}	10^{-15}
Stopping criterion	True residual	True residual	Internal residual

Newton steps	175	×	190
Iter GMRES/BiCGStab	34	×	70
Maximum be	10^{-11}	10^{-11}	10^{-12}
Total simul	1798	×	2764

Table 4.37: Number of Newton steps, average number of Krylov iterations to solve each unsymmetric Schur system, maximum normwise backward error for each computed solution and total elapsed time (in seconds) for the simulation in three tests associated to *Simulation 6*. × = the nonlinear scheme is not converging.

In Table 4.37 we compare on *Simulation 6* BiCGStab and GMRES. In a first test we use GMRES with ϵ_{Krylov} set to 10^{-11} and a stopping criterion based on (4.7) (denoted by *True residual*). We have also computed in output of GMRES the normwise backward error

$$be = \frac{\| S\tilde{u} - g \|}{\| g \|}$$

where $\tilde{u}$ is the approximate solution computed by the Krylov solver. We display in Table 4.37 the largest observed value of *be* (denoted by *Maximum be*) during the simulation. Then in the second test (reported in the second column of Table 4.37) we use BiCGStab with ϵ_{Krylov} set to 10^{-11} and a stopping criterion based on (4.7). We can see that in this case the steady state is not obtained even though the largest observed value of *be* is also 10^{-11}. In a third test we use BiCGStab with ϵ_{Krylov} set to 10^{-15} and the stopping criterion based on (4.6) (denoted by *Internal residual*). In this case the steady state is reached but the number of Newton steps is larger than for GMRES with ϵ_{Krylov} set to 10^{-11}. The number Krylov iterations is also larger and some linear systems do not converge to the required accuracy. The largest observed value of *be* is only of 10^{-12} while the required accuracy was 10^{-15}. Therefore we can observe a difference between the internal residual (4.6) and the true residual (4.7). Moreover BiCGStab requires two matrix-vector products per iteration while GMRES only requires one. Therefore the total simulation time is larger with BiCGStab than with GMRES. These results justify our choice of GMRES for solving the unsymmetric systems in our application.

We point out that these results also raise the problem of the relevance of the stopping criterion based on normwise backward error. For a requested accuracy of 10^{-11} in term of normwise backward error, the nonlinear scheme is converging with GMRES but not with BiCGStab. Somehow, the quantity $\frac{\| r \|}{\| g \|}$ does not measure precisely the quality of the solution provided to the nonlinear scheme, using a component-wise backward error might be an alternative but its computation at each iteration is prohibitive as its cost is higher than a matrix-vector product.

4.3.2 Right versus left preconditioning for GMRES

As explained in Chapter 3, there are three possible locations for the preconditioner for GMRES which define three preconditioning techniques. These techniques are left preconditioning, right preconditioning and split preconditioning. In our case, split preconditioning is not possible because our preconditioner is not expressed in a factorized form. Then we have to choose between left and right preconditioning. This choice may have an influence on convergence and has also an effect on the stopping criterion in the package used [56]. Let

$$Su = g \tag{4.8}$$

be the Schur complement system to be solved. In the case of right preconditioned GMRES, the stopping criterion is

$$\frac{\| r_n \|}{\| g \|} \tag{4.9}$$

where $r_n = Su_n - g$ is the true residual at step n of the Krylov method. In the case of left preconditioned GMRES, the stopping criterion becomes

$$\frac{\| Mr_n \|}{\| Mg \|} \tag{4.10}$$

where M is the preconditioner associated with the Schur complement.

Table 4.38 presents, for right and left preconditioning, the number of Newton steps and the average number of GMRES iterations for four different simulations. These simulations are carried out on *Mesh S* and *Mesh M* decomposed in 8 or 16 subdomains. The preconditioner is M_{AS} and ϵ_{Krylov} is set to 10^{-11}. It can be seen that the nonlinear convergence is affected by the preconditioning technique only for *Mesh S* decomposed in 16 subdomains. In this case left preconditioning deteriorates the nonlinear convergence of Newton methods for the transport equations.

In the case of *Mesh M*, the nonlinear convergence and the linear convergence are not affected by the location of the preconditioner. But in the case of *Mesh S*, left preconditioning deteriorates the linear convergence in the sense that for some linear systems the backward error normwise of the preconditioned system is not able to attain the required precision.

These results justify the fact that all the iterative substructuring experiments in Sections 4.1 and 4.2 have been performed with a right preconditioned GMRES method. Intuitively, we can argue that the quantity that really influences the Newton solvers is the unpreconditioned backward error defined by (4.9) and not the preconditioned one defined by (4.10).

4.3.3 Choice of the orthogonalization scheme in GMRES

The GMRES algorithm is based on the construction of an orthogonal basis for the Krylov subspace $\mathcal{K}_n = span\{v, Av, A^2v, \ldots, A^{n-1}v\}$. At step n of the algorithm,

Mesh S, 8 subdomains

	Left	Right
Newton steps	178	178
iter GMRES	34	10

Mesh M, 8 subdomains

	Left	Right
Newton steps	179	179
iter GMRES	14	14

Mesh S, 16 subdomains

	Left	Right
Newton steps	191	183
iter GMRES	55	18

Mesh M, 16 subdomains

	Left	Right
Newton steps	176	176
iter GMRES	24	24

Table 4.38: Influence of right and left preconditioning on the number of Newton steps and on the number of GMRES iterations for simulations on *Mesh S* and *Mesh M* decomposed in 8 or 16 subdomains.

the vector $A^{n-1}v$ is orthogonalized against the basis of the Krylov subspace $\mathcal{K}_{n-1}$ previously computed.

Usually, the method used is the "Modified Gram-Schmidt" (MGS) orthogonalization scheme [93]. Unfortunately, a loss of orthogonality may happened in finite precision arithmetic. This loss of orthogonality can be compensated by an iteration of the orthogonalization scheme [14]. The resulting algorithm is called "Iterative Modified Gram-Schmidt"(IMGS). The main drawback of IMGS is an increased number of dot products. The "Classical Gram-Schmidt" algorithm (CGS) is easy to implement and can be more efficiently implemented on parallel distributed computers. It is numerically less accurate than IMGS. But CGS can also be iterated (ICGS). It is then as numerically efficient as IMGS. The ICGS orthogonalization is especially interesting in a distributed environment because the number of global reductions needed to make the orthogonalization is smaller [99]. We have tested these four orthogonalizations on a simulation based on a decomposition of *Mesh M* in 8 subdomains. These experiments have been performed on a Compaq Alpha server because it is a more stable platform for accurate time measurements. Table 4.39 presents, for these four orthogonalization schemes, the number of Newton steps, the average number of GMRES iterations, the average time spent to solve a linear system and the average time spent in the orthogonalization scheme for the solution of each unsymmetric linear system. These results confirm the instability of CGS. It is

	Orthogonalization			
	IMGS	ICGS	CGS	MGS
Newton steps	179	179	216	179
Iter GMRES	14	14	84	14
Time $Ax = b$	2.50	2.46	3.23	2.44
Time dot products	0.028	0.018	0.294	0.020

Table 4.39: Comparison of different orthogonalization schemes for GMRES in the case of a simulation based on *Mesh M* decomposed in 8 subdomains.

the only method that deteriorates the nonlinear convergence. This is also another indication of the ill-conditioning of the linear systems that we solve.

On this test MGS is stable enough and IMGS is not needed. Concerning the computational time spent in the dot product procedure, ICGS is the less expensive and IMGS is the most expensive. However the gain is not significative compared with the time needed to solve a linear system. In all the tests of Sections 4.1 and 4.2 where GMRES is used, the orthogonalization has been performed using the IMGS scheme because it is expected to be the most stable. We notice that this choice was also made by [109] for similar reason. This choice is not optimal but conservative and more secure. In addition the extra cost is negligible compared with the other calculations involved in the rest of the simulation.

4.3.4 Restart for GMRES

One drawback of GMRES compared to BiCGStab is that at iteration k, the k vectors of the Arnoldi basis have to be stored. The "restarting" technique enables to reduce the amount of memory required as full GMRES becomes prohibitive for large problems [96].

In the case of iterative substructuring, only the interface problem is solved by GMRES. The storage requirement for vectors defined on the interface is negligible compared to the memory needed to store the factors of the internal problems and the dense local Schur complement matrices. Furthermore, the use of the restart may slow-down the convergence of GMRES and therefore the nonlinear convergence. For example, if we run *Large 32* (see Tables 4.19 and 4.22) with a restart of 10 or 20 for GMRES, then a lot of linear systems do not converge any more and the steady state can no longer be obtained. For these reasons we have discarded the use of the restart in our experiments.

Chapter 5

Prospectives

In this chapter, we present some new ideas that might deserve future research investigations. In Section 5.1, we present a sparsification procedure to relax the memory requirement to store the local block preconditioners used in iterative substructuring algorithms. In Section 5.2, we present a preliminary study of a two-level spectral preconditioner that exhibits promising behaviour on difficult problems. In Section 5.3, we introduce the possibility to use parallel solvers on the subdomains that leads to parallel implementations with two level of parallelism.

5.1 Sparsified block preconditioners

The memory requirement to store the M_{AS} preconditioner is quite large (see Section 4.2.5). With the notation of Section 3.2.2, we recall that the preconditioner M_{AS} is

$$M_{AS} = \sum_{i=1}^{N} R_{\Gamma_i}^T (\bar{S}^{(i)})^{-1} R_{\Gamma_i} \text{ with } \bar{S}^{(i)} = R_{\Gamma_i} S R_{\Gamma_i}^T. \tag{5.1}$$

A possible alternative to get a cheaper preconditioner is to consider a sparse approximation for $\bar{S}^{(i)}$ in (5.1) which may result in a saving of memory to store the preconditioner and saving of computation to factorize and apply the preconditioner. This approximation $\hat{S}^{(i)}$ can be constructed by dropping the elements of $\bar{S}^{(i)}$ that are smaller than a given threshold. More precisely the following dropping strategy can be applied on each matrix $\bar{S}^{(i)}$:

$$\hat{s}_{kl} = \begin{cases} 0 & \text{if } \bar{s}_{kl} \leq \eta(|\bar{s}_{kk}| + |\bar{s}_{ll}|) \\ \bar{s}_{kl} & \text{else} \end{cases} \tag{5.2}$$

Then the resulting sparsified blocks are factorized using a sparse direct solver (MUMPS in our case) instead of a dense direct solver (LAPACK in our implementation of the subdomain based preconditioner). We notice that the strategy defined by (5.2) preserves the symmetry but has also some advantages for an implementation in a distributed memory environment as it does not require any additional

communication compared to the construction of M_{AS}. This is not the case if one decide to consider the maximum per row for the denominator (instead of the diagonal element). Experimentally, in semiconductor simulations, the largest entries are most often on the diagonal but it happens sometimes not be there.

The threshold η is set to 0.01. We compare the M_{AS} preconditioner and its sparsified version (denoted by $M_{sp(AS,\eta)}$) on simulation *Large 32* (see Table 4.19). The threshold η is set to 0.01. The sparsified preconditioner is only used for the unsymmetric systems. In the SPD case, the sparsification procedure can lead to indefinite symmetric systems as the matrices are not $M-$matrices [27], consequently the sparsification procedure might not preserve the positive definiteness property. The results are given in Table 5.1. The sparsity ratio is the percentage of elements of

	M_{AS}	$M_{sp(AS,0.01)}$
Newton steps	175	176
Iter CG	32	32
Iter GMRES	34	35
Sparsity ratio ratio (max)	$\times$	47 %
Sparsity ratio (min)	$\times$	1 %
Cumulative times		
Total $Ax = b$	1430	1806
Total simul	1798	2210
Average time per system		
Build Precond	0.58	1.27
Apply Precond	0.57	0.87
Total $Su = g$	2.13	3.38

Table 5.1: Results of M_{AS} and of the sparsified preconditioner $M_{sp(AS,0.01)}$ for *Large 32*.

the Schur matrix that are kept after the smallest entries have been dropped. In this table we display the maximum and minimum values of this sparsity ratio during the numerical simulation. We can observe that the sparsity ratio is quite low while the linear and the nonlinear convergences are only slightly perturbed. Paradoxically, the times of the construction and the application of the preconditioner are higher with the sparsified version. This is due to the small sizes of the local Schur complements that vary between 115 and 646 (see Table 4.3). In this case, the LAPACK dense solver performs better than a sparse solver like MUMPS. We notice that this technique may be of interest for bigger problems (possibly 3D problems). Furthermore, these tests reveal that a lot of the information contained in the preconditioner is not meaningful to accelerate the Krylov solver.

5.2 Spectral two-level preconditioners

5.2.1 Motivation and general presentation

When solving the linear system $Ax = b$ with a Krylov method, the smallest eigenvalues of the matrix A often slow down the convergence. In the SPD case, this is clearly highlighted by the bound on the rate of convergence of the Conjugate Gradient method (CG) given by

$$||e^{(k)}||_A \leq (\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1})^k ||e^{(0)}||_A, \tag{5.3}$$

where $e^{(k)} = x^* - x^{(k)}$ denotes the forward error associated with the iterate at step k and $\kappa(A) = \frac{\lambda_{max}}{\lambda_{min}}$ denotes the condition number. From this bound it can be said that enlarging the smallest eigenvalues would improve the convergence rate of CG. Consequently if the smallest eigenvalues of A could be somehow "removed" the convergence of CG will be improved. Similarly for unsymmetric systems arguments exist to explain the bad effect of the smallest eigenvalues on the rate of convergence of the unsymmetric Krylov solver [12, 50, 87]. To cure this, several techniques have been proposed in the last few years, mainly to improve the convergence of GMRES. In [87], it is proposed to add a basis of the invariant space associated with the smallest eigenvalues to the Krylov basis generated by GMRES. Another approach based on a low rank update of the preconditioner for GMRES was proposed by [12, 50]. They consider the orthogonal complement of the invariant subspace associated with the smallest eigenvalues to build a low rank update of the preconditioned system. Finally, in [78] a preconditioner for GMRES based on a sequence of rank-one updates is proposed that involves the left and right smallest eigenvectors. In our work, we consider an explicit eigencomputation which makes the preconditioner independent of the Krylov solver used in the actual solution of the linear system.

We first present our technique for unsymmetric linear systems and then derive a variant for symmetric and SPD matrices. We consider the solution of the linear system

$$Ax = b, \tag{5.4}$$

where A is a $n \times n$ unsymmetric non singular matrix, x and b are vectors of size n. The linear system is solved using a preconditioned Krylov solver and we denote by M_1 the left preconditioner, meaning that we solve

$$M_1 Ax = M_1 b. \tag{5.5}$$

We assume that the preconditioned matrix $M_1 A$ is diagonalizable, that is:

$$M_1 A = V \Lambda V^{-1}, \tag{5.6}$$

with $\Lambda = diag(\lambda_i)$, where $|\lambda_1| \leq \ldots \leq |\lambda_n|$ are the eigenvalues and $V = (v_i)$ the associated right eigenvectors. We denote by $U = (u_i)$ the associated left eigenvectors;

we then have $U^H V = diag(u_i^H v_i)$, with $u_i^H v_i \neq 0,\ \forall i$ [112]. Let V_ε be the set of right eigenvectors associated with the set of eigenvalues λ_i such that $|\lambda_i| \leq \varepsilon$. Similarly we define by U_ε the corresponding subset of left eigenvectors.

Theorem 1 *Let*

$$A_c = U_\varepsilon^H M_1 A V_\varepsilon,$$

$$M_c = V_\varepsilon A_c^{-1} U_\varepsilon^H M_1$$

and

$$M = M_1 + M_c.$$

Then MA is diagonalizable and we have $MA = V diag(\eta_i) V^{-1}$ with

$$\begin{cases} \eta_i = \lambda_i & \text{if} \quad |\lambda_i| > \varepsilon, \\ \eta_i = 1 + \lambda_i & \text{if} \quad |\lambda_i| \leq \varepsilon. \end{cases}$$

Proof

We first remark that $A_c = diag(\lambda_i u_i^H v_i)$ with $|\lambda_i| \leq \varepsilon$ and then A_c is non singular.
Let $V = (V_\varepsilon V_{\bar{\varepsilon}})$, where $V_{\bar{\varepsilon}}$ is the set of $(n-k)$ right eigenvectors associated with eigenvalues $|\lambda_i| > \varepsilon$.
Let $D_\varepsilon = diag(\lambda_i)$ with $|\lambda_i| \leq \varepsilon$ and $D_{\bar{\varepsilon}} = diag(\lambda_j)$ with $|\lambda_j| > \varepsilon$.

$$\begin{aligned} MAV_\varepsilon &= M_1 A V_\varepsilon + V_\varepsilon A_c^{-1} U_\varepsilon^H M_1 A V_\varepsilon \\ &= V_\varepsilon D_\varepsilon + V_\varepsilon I_k \\ &= V_\varepsilon (D_\varepsilon + I_k) \end{aligned}$$

where I_k denotes the $(k \times k)$ identity matrix.

$$\begin{aligned} MAV_{\bar{\varepsilon}} &= M_1 A V_{\bar{\varepsilon}} + V_\varepsilon A_c^{-1} U_\varepsilon^H M_1 A V_{\bar{\varepsilon}} \\ &= V_{\bar{\varepsilon}} D_{\bar{\varepsilon}} + V_\varepsilon A_c^{-1} U_\varepsilon^H V_{\bar{\varepsilon}} D_{\bar{\varepsilon}} \\ &= V_{\bar{\varepsilon}} D_{\bar{\varepsilon}} \qquad \text{as } U_\varepsilon^H V_{\bar{\varepsilon}} = 0. \end{aligned}$$

We then have

$$MAV = V \begin{pmatrix} D_\varepsilon + I_k & 0 \\ 0 & D_{\bar{\varepsilon}} \end{pmatrix}.$$

■

Theorem 2 *Let W be such that*

$$\tilde{A}_c = W^H A V_\varepsilon \ \text{ has full rank,}$$

$$\tilde{M}_c = V_\varepsilon \tilde{A}_c^{-1} W^H$$

and

$$\tilde{M} = M_1 + \tilde{M}_c.$$

Then $\tilde{M}A$ is similar to a matrix whose eigenvalues are

$$\begin{cases} \eta_i = \lambda_i & \text{if} \quad |\lambda_i| > \varepsilon, \\ \eta_i = 1 + \lambda_i & \text{if} \quad |\lambda_i| \leq \varepsilon. \end{cases}$$

Proof

With the same notation as for Theorem 1 we have:

$$\begin{aligned}
\tilde{M}AV_\varepsilon &= M_1AV_\varepsilon + V_\varepsilon A_c^{-1}W^HAV_\varepsilon \\
&= V_\varepsilon D_\varepsilon + V_\varepsilon I_k \\
&= V_\varepsilon(D_\varepsilon + I_k) \\
\tilde{M}AV_{\bar{\varepsilon}} &= M_1AV_{\bar{\varepsilon}} + V_\varepsilon A_c^{-1}W^HAV_{\bar{\varepsilon}} \\
&= V_{\bar{\varepsilon}}D_{\bar{\varepsilon}} + V_\varepsilon C \qquad \text{with } C = A_c^{-1}W^HAV_{\bar{\varepsilon}} \\
&= \begin{pmatrix} V_\varepsilon V_{\bar{\varepsilon}} \end{pmatrix} \begin{pmatrix} C \\ D_{\bar{\varepsilon}} \end{pmatrix}
\end{aligned}$$

We then have

$$\tilde{M}AV = V \begin{pmatrix} D_\varepsilon + I_k & C \\ 0 & D_{\bar{\varepsilon}} \end{pmatrix}.$$

■

For right preconditioning, that is $AM_1y = b$, similar results hold.

Lemma 1 *Let*

$$A_c = U_\varepsilon^H AM_1V_\varepsilon,$$

$$M_c = M_1V_\varepsilon A_c^{-1}U_\varepsilon^H$$

and

$$M = M_1 + M_c.$$

Then AM is diagonalizable and we have $AM = V\,diag(\eta_i)V^{-1}$ with

$$\begin{cases} \eta_i = \lambda_i & \text{if} \quad |\lambda_i| > \varepsilon, \\ \eta_i = 1 + \lambda_i & \text{if} \quad |\lambda_i| \leq \varepsilon. \end{cases}$$

Lemma 2 *Let W be such that*

$$\tilde{A}_c = W^HAMV_\varepsilon \text{ has full rank,}$$

$$\tilde{M}_c = M_1V_\varepsilon\tilde{A}_c^{-1}W^H$$

and

$$\tilde{M} = M_1 + \tilde{M}_c.$$

Then $A\tilde{M}$ is similar to a matrix whose eigenvalues are

$$\begin{cases} \eta_i = \lambda_i & \text{if} \quad |\lambda_i| > \varepsilon, \\ \eta_i = 1 + \lambda_i & \text{if} \quad |\lambda_i| \leq \varepsilon. \end{cases}$$

In the SPD case we can propose the following theorem

Theorem 3 *If A and M_1 are SPD, then*

$$M_1A \text{ is diagonalizable,}$$

and

$$\tilde{A}_c = V_\varepsilon^H A V_\varepsilon \text{ is SPD.}$$

The preconditioner defined by

$$\tilde{M} = M_1 + \tilde{M}_c, \text{ with } \tilde{M}_c = V_\varepsilon \tilde{A}_c^{-1} V_\varepsilon^H$$

is SPD and $\tilde{M}A$ is similar to a matrix whose eigenvalues are

$$\begin{cases} \eta_i = \lambda_i & \text{if} \quad |\lambda_i| > \varepsilon, \\ \eta_i = 1 + \lambda_i & \text{if} \quad |\lambda_i| \leq \varepsilon. \end{cases}$$

Proof

The matrix $M_1 A$ is similar to the matrix $M_1^{\frac{1}{2}} A M_1^{\frac{1}{2}}$ which is similar to a diagonal matrix as it is a real symmetric matrix. Therefore the matrix $M_1 A$ is diagonalizable.

By construction $\tilde{A}_c$ is symmetric, let us show that it is positive definite. V_ε is a $n \times k$ matrix. Let $z \in \mathbb{R}^k,\ z \neq 0$.

$$\begin{aligned} < \tilde{A}_c z, z > &= < V_\varepsilon^H A V_\varepsilon z, z > \\ &= < A V_\varepsilon z, V_\varepsilon z > \end{aligned}$$

$V_\varepsilon z \neq 0$ because V_ε has full rank. Therefore

$< \tilde{A}_c z, z > \ > \ 0$ as A is a SPD matrix.

Therefore $\tilde{A}_c$ is a SPD matrix and consequently has full rank.

Let $x \in \mathbb{R}^n,\ x \neq 0$.

$$\begin{aligned} < \tilde{M}_c x, x > &= < V_\varepsilon \tilde{A}_c^{-1} V_\varepsilon^H x, x > \\ &= < \tilde{A}_c^{-1} V_\varepsilon^H x, V_\varepsilon^H x > \\ &\geq 0 \text{ as } \tilde{A}_c \text{ is a SPD matrix.} \end{aligned}$$

Therefore $\tilde{M}_c$ is a positive semi-definite matrix and $\tilde{M} = M_1 + \tilde{M}_c$ is a SPD matrix as M_1 is SPD.

Concerning the distribution of the eigenvalues of $\tilde{M}A$ the proof of Theorem 2 remains valid here.

■

This technique of two-level spectral preconditioning has also been applied in combination with sparse inverse preconditioners to solve dense linear systems in electromagnetism applications [25]. In the following section we present its application to iterative substructuring algorithms.

5.2.2 Application to iterative substructuring algorithms

We consider the case of the Schur complement system

$$Su = g$$

preconditioned by the local block preconditioner M_{AS} defined in Section 3.2.2. We can remark that the two-level preconditioners presented in Section 3.3.3 has a similar form as the one of the two-level spectral preconditioners presented in Section 5.2.1. The difference resides in the choice of the vectors that span the coarse space. These vectors are given a priori in 3.2.2 while they span the subspace associated with the smallest eigenvalues of the M_{AS} matrix here.

We select few linear systems from a semiconductor simulation in order to investigate the numerical behaviour of the spectral two-level preconditioners for Schur complement systems. We assemble the complete Schur matrices as well as the associated M_{AS}. Then we perform experiments with Matlab on these matrices. We select 8 Schur matrices denoted by $S_{i,\ i=1,\ldots,8}$. and the associated M_{AS} preconditioners denoted by M_{ASi}. The main characteristics of these matrices are displayed in Table 5.2. These characteristics are the symmetry of the Schur complement matrix, its size and the number of subdomains of the decomposition on which it is defined. The linear systems are arising from the discretization of the continuity equation for

	Equation	Symmetry	Number of subdomains	Size
S_1	Init	SPD	8	958
S_2	Init	SPD	8	958
S_3	Init	SPD	16	1607
S_4	Init	SPD	16	1607
S_5	Electrons	Unsymmetric	8	958
S_6	Holes	Unsymmetric	8	958
S_7	Holes	Unsymmetric	16	1607
S_8	Electrons	Unsymmetric	16	1607

Table 5.2: Characteristics of the Schur complement matrices selected.

the electrons or for the holes and of their linearizations (*Init* systems). Symmetric diagonal scaling has been applied in all the cases.

Numerical experiments with SPD Schur complement matrices

Following the theory developed in Section 5.2.1, we consider for the four SPD test cases the two-level spectral preconditioner defined by

$$M = M_{AS} + M_C \tag{5.7}$$

and

$$M_C = V_k (V_k^H S V_k)^{-1} V_k^H, \tag{5.8}$$

where V_k is a set of k right eigenvectors associated with the k smallest eigenvalues in modulus of $M_{AS}S$. The value of the parameter k defines the size of the coarse space. In Table 5.3, we display the number of CG iterations needed to solve the Schur complement system

$$Su = g$$

	Schur matrix			
Size of the coarse space k	S_1	S_2	S_3	S_4
0	15	17	40	50
1	14	16	37	42
2	13	15	32	37
3	12	14	30	34
4	11	13	28	30
5	11	13	27	27
6	11	12	25	26
7	11	12	24	24
8	11	12	23	23
9	11	12	23	23
10	11	12	23	23

Table 5.3: Number of CG iterations needed to solve an SPD Schur complement system in the case of two-level spectral preconditioning.

preconditioned by (5.7). The value of ϵ_{Krylov} is set to 10^{-11}.

We can see that for a decomposition into 8 subdomains the number of CG iterations decreases quite slowly when the size of the coarse space increases. For the test cases related to a decomposition into 16 subdomains the gain is more significant. With a coarse space of dimension 3 we reduce the number of CG iterations of more than 30 % in the case of S_4.

Numerical experiments with unsymmetric Schur complement matrices

We consider left preconditioning for the unsymmetric Schur complement systems. We refer to the two-level spectral preconditioners defined in Theorems 1 and 2 of Section 5.2.1. Let V_k be a set of k right eigenvectors corresponding to the k smallest eigenvalues in modulus of SM_{AS}. Let U_k be a set of k left eigenvectors corresponding to the smallest eigenvalues in modulus of SM_{AS}. In Lemma 2, we replace A by S, M_1 by M_{AS} and V_ε by V_k. Then we set $W = M_{AS}U_k$ or $W = V_k$. The two resulting coarse space components for the local preconditioner M_{AS} are denoted by M_{C1} (associated with $M_{AS}U_k$) and M_{C2} (associated with V_k). We may remark that we can have pairs of conjugate complex eigenvalues. For a practical implementation, it is possible to consider the real and the imaginary part of one of the eigenvector instead of two complex eigenvectors. In this case, the diagonal matrix $diag(\eta_i)$ considered in Theorem 1 is replaced by a block diagonal matrix with real blocks of size 2×2 or 1×1.

In Table 5.4, we display the number of full GMRES iterations needed to solve each unsymmetric Schur complement systems. For each test case, we consider the two coarse space components. The value of ϵ_{Krylov} is set to 10^{-11}. In Table 5.5, we consider the same experiments with the BiCGStab solver.

As expected, we see that there is almost no difference between M_{C1} and M_{C2}

	S_5		S_6		S_7		S_8	
k	M_{C1}	M_{C2}	M_{C1}	M_{C2}	M_{C1}	M_{C2}	M_{C1}	M_{C2}
0	9	9	18	18	10	10	36	36
1	9	9	16	16	11	11	33	33
2	9	9	15	15	11	11	30	30
3	9	9	14	14	11	11	28	28
4	9	9	14	14	10	10	27	26
5	9	9	13	13	11	11	25	25
6	9	9	12	12	10	10	24	23
7	9	9	12	12	11	11	23	23
8	10	10	12	12	11	11	22	22
9	10	10	12	12	10	10	22	21
10	10	10	12	12	10	10	22	21

Table 5.4: Number of GMRES iterations needed to solve each unsymmetric Schur complement in the case of two-level spectral preconditioning (k is the dimension of the coarse space).

	S_5		S_6		S_7		S_8	
k	M_{C1}	M_{C2}	M_{C1}	M_{C2}	M_{C1}	M_{C2}	M_{C1}	M_{C2}
0	6	6	11	11	7	7	24	24
1	6	6	10	10	7	7	20	20
2	6	6	9	9	7	7	17	17
3	6	6	9	9	6	6	16	16
4	6	6	8	8	6	6	16	16
5	6	6	8	8	6	6	14	14
6	6	6	7	7	6	6	13	13
7	6	6	7	7	7	7	13	13
8	6	6	7	7	6	6	13	12
9	6	6	7	7	6	6	12	12
10	6	6	7	7	6	6	12	13

Table 5.5: Number of BiCGStab iterations needed to solve each unsymmetric Schur complement in the case of two-level spectral preconditioning (k is the dimension of the coarse space).

(except a variation of one iteration for some cases for S_8). BiCGStab requires less iterations than GMRES but we recall that, in the context of our semiconductor applications, BiCGStab is less robust than GMRES (see Section 4.3.1) and that BiCGStab requires two-matrix vector products per iteration while GMRES only requires one.

Without any coarse space, S_5 and S_7 are easier to solve than S_6 and S_8. The spectral two-level preconditioning has no effect on S_5 and S_7. For S_6, the iteration number decreases slowly when the size of the coarse space increases. For S_8, which is the most difficult problem, the spectral two-level preconditioner is numerically efficient. To illustrate the difference between S_7 and S_8 we have plotted the spectra of $S_7 M_{AS7}$ and of $S_7(M_{AS7} + M_{C1})$ (for a coarse space of dimension 3) on Figure 5.2; and we have plot the spectra of $S_8 M_{AS8}$ and of $S_8(M_{AS8} + M_{C1})$ (for a coarse space of dimension 3) on Figure 5.1. We can see that both for S_7 and S_8, most of the

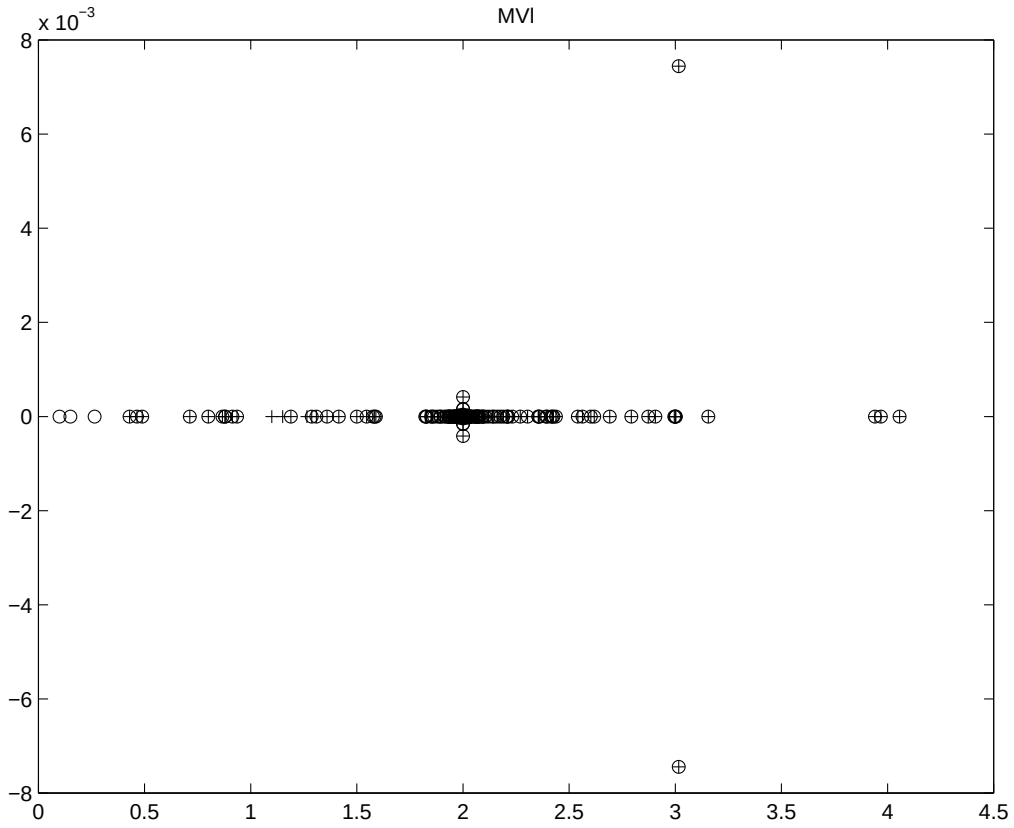

Figure 5.1: 'o' represent the eigenvalues of $S_8 M_{AS8}$; '+' represent the eigenvalues of $S_8(M_{AS8} + M_{C1})$ with a coarse space of dimension 3.

eigenvalues are clustered around 2. This can be explained by the distributed nature of the preconditioner M_{AS} which is locally an approximation of $2 \cdot S^{-1}$. We see that for S_7 and S_8, the three smallest eigenvalues are effectively shifted by one. For S_8, the three smallest eigenvalues are close to zero while for S_7 they are closer to two than to zero. This indicates why two-level spectral preconditioning is efficient in the

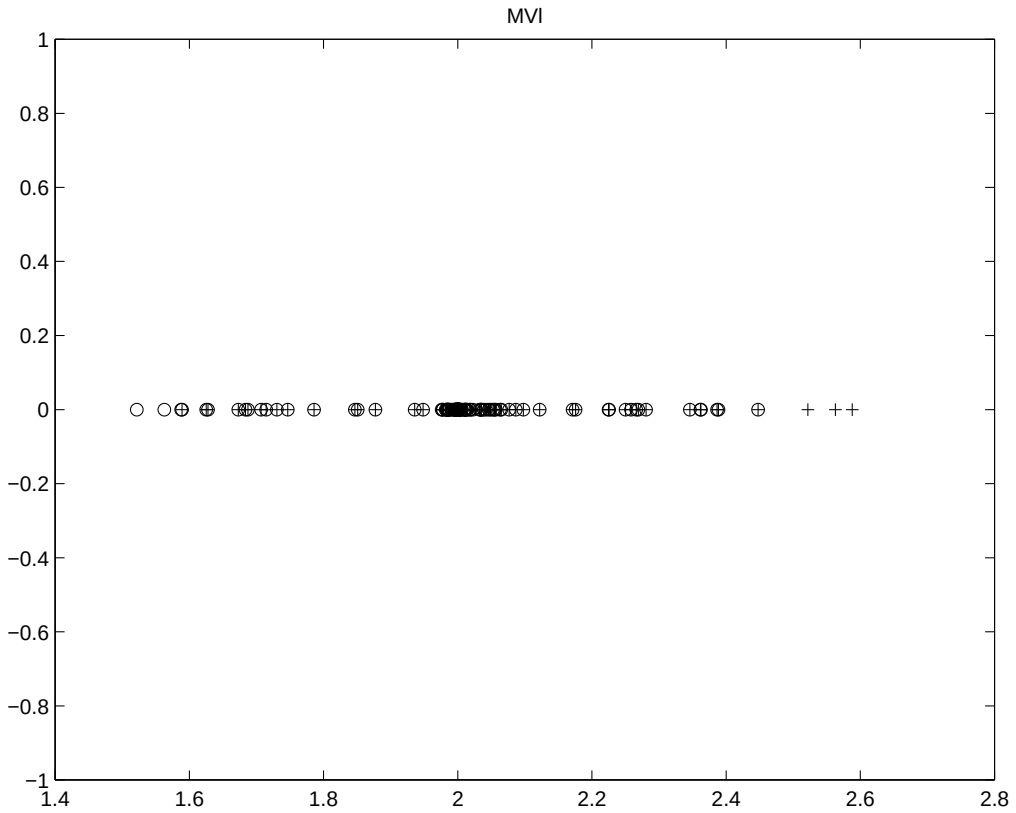

Figure 5.2: 'o' represent the eigenvalues of $S_7 M_{AS7}$; '+' represent the eigenvalues of $S_7(M_{AS7} + M_{C1})$ with a coarse space of dimension 3.

case of S_8 but not in the case of S_7.

Comparison with other two-level preconditioners

We can make a qualitative comparison with the other two-level preconditioners described in Chapter 3 and tested in Chapter 4. Actually, the eight test matrices $S_1, \ldots, S_8$ have been extracted from a simulation on a decomposition into 8 or 16 subdomains of the mesh *Mesh S* presented in Section 4.1.1. Tables 4.12 and 4.13 present the numerical behaviour of the two-level preconditioners M_{AS-sub}, $M_{AS-edge}$ and $M_{BNN(3)}$ for the same simulation.

In the unsymmetric case, the two-level preconditioners $M_{AS-edge}$ and M_{AS-sub} are not efficient and even worse deteriorate the convergence of GMRES preconditioned by M_{AS}. The results of Table 5.4 show that two-level spectral preconditioning improves the numerical behaviour of M_{AS} when the problem is relatively difficult (S_6 and S_8) and hopefully does not deteriorate it when the problem is easier (S_5 and S_7).

Concerning the SPD case, the four test matrices $S_1, \ldots, S_4$ are *Init systems* as defined in Section 4.1.5. We see in Table 4.13 that for this class of systems M_{AS} without any coarse space is more efficient than $M_{AS-edge}$, M_{AS-sub}, $M_{BNN(1)}$ or $M_{BNN(3)}$. On the contrary, we see in Table 5.3 that spectral two-level preconditioning always improves the numerical behaviour of M_{AS}. Therefore $M_{AS-spectral}$ is numerically the most efficient preconditioner tested for the matrices S_i.

These preliminary results are promising, but the implementation of the method in the semiconductor code may rise new difficulties. The eigenvectors will be computed using an iterative method that will increase the numerical complexity of the complete solver. The accuracy required for the computation of the eigenvectors and their number are new parameters that will have to be tuned carefully to ensure the efficiency and the robustness of the numerical method.

5.3 Implementation exploiting two levels of parallelism

We have seen in Section 4.1.5 that the number of iterations of the Krylov solvers required to solve each Schur complement system increases with a factor of approximatively two when the number of subdomains of the decomposition is twice bigger. In Section 4.2.3 we have illustrated that the explicit iterative substructuring method allows a relatively large number of Krylov iterations. We have seen that the cost of the increase of the iteration number is not penalizing for the overall simulation time. Due to the limitation of our computer resources, we have not made extensive tests with more than 32 processors. In the future, we may run larger simulations on a larger number of processors. If the number of Krylov iterations becomes too large, the iterative solution of the Schur system may not be affordable any more. Especially, we could mention that the dot product calculation is a well-known bot-

tleneck of the Krylov solvers on a large number of processors as they require global reductions.

The main motivation for two-level preconditioners is to mitigate the increase in the number of iterations of the Krylov solver when the number of subdomains increases. Unfortunately, the two-level preconditioners presented in Chapter 3 are not able to alleviate the number of iterations and even worse often deteriorate the convergence of the Krylov solver in the context of our semiconductor applications (see the results of Section 4.1.5). In the previous section, we have presented a two-level spectral preconditioner that may correct this drawback. However a software alternative might also be proposed that uses parallel solvers on the subdomains. This leads to algorithms that exploit two levels of parallelism. The latter approach has some advantages as, so far, no efficient two-level preconditioner has been identified. Two level of parallelism would enable us to use more processors without increasing linearly the number of subdomains and consequently without deteriorating the numerical behaviour of the associated iterative linear solvers. Practically, an instance of MUMPS with a few processors could be used concurrently on each subdomain to perform the numerical factorization of the internal subproblems. The Schur complement feature of MUMPS is available for distributed matrices but the dense Schur complement computed is currently centralized on the "host". A future release of MUMPS may integrate a distributed Schur complement as output. In this case the dense matrix-vector product with the local Schur complements could be performed using the ScaLAPACK library [18]. This second level of parallelism has not been implemented yet. The fact that one subdomain corresponds to one processor allows a simple implementation of the communications between two subdomains. Therefore, the use of parallel solvers on the subdomains would imply to modify the data structures that describe the communication pattern of the existing code. We should also mention that one difficulty will be to decide when to switch from one level of parallelism to two levels of parallelism as this criterion should include numerical and software ingredients.

Conclusion

The main objective of this study was to test modern parallel numerical linear algebra algorithms in the context of a complex physical application. From a performance point of view the parallelization of the code based on state of the art parallel linear solvers is successful. A simulation that requires 24 hours of computational time with the initial code only requires a few minutes to be completed using the parallel code. This tremendous reduction of elapsed time is only partially due to the use of parallel computers but mainly induced by the use of efficient modern techniques for solving large sparse linear systems. Moreover the new parallel code allows computations that are out of reach for the sequential code but are essential to validate the mathematical model of the semiconductors. A. Marrocco from INRIA has performed a set of experiments that shows that it is possible to obtain a solution almost independent from the mesh if this one is sufficiently refined. These experiments have been carried out using the iterative substructuring method that is M_{AS} with diagonal scaling on S, on meshes with more than 800 000 elements decomposed into 16 subdomains [86].

It is important to notice that in that work we have implemented the iterative solvers directly in the simulation code. This approach rises new difficulties and finding preconditioners to make converging the Krylov solver appears not to be sufficient to ensure the numerical robustness of the complete numerical scheme. For instance, we can cite the crucial influence of a combination of scaling and preconditioning and the importance of the choice of the stopping criterion threshold for the Krylov solvers. We would not have found such difficulties if we had only run experiments on few isolated linear systems extracted from the simulation, as it is sometimes convenient to do for comparing different preconditioners.

We want to emphasize that in this work we tried to make a fair comparison between iterative and direct solvers from the numerical, the performance and the software development point of view. The parallel direct software MUMPS is easy to interface with the semiconductor device simulation code. The implementation of direct substructuring methods requires additional effort but remains relatively straightforward. This simplicity of use is one of the strength of direct solvers. Numerically, the main advantage of direct methods is their robustness. In all our test cases, direct methods give better results than iterative ones concerning the convergence of the Euler and the Newton nonlinear schemes. Nevertheless, when iterative solvers are well tuned this difference almost vanishes. Moreover, they outperform

direct solvers concerning the overall simulation time. However, this tuning can be quite complex and requires some expertise in numerical analysis and linear algebra.

Both direct and iterative methods can expect improvements from future software development in MUMPS. Nested dissection orderings for direct methods may improve their performance. Future releases of MUMPS will include the possibility to use different mesh partitioners in order to compute a more efficient ordering for the factorization of the linear systems. Concerning iterative solvers, the study of two-level spectral preconditioners, that has to be completed, is promising and it might give more robust preconditioners but also give some clue to understand the deficiency of the other two-level preconditioners tested.

Bibliography

[1] V. I. Agoshkov. Poincaré-Steklov operators and domain decomposition methods in finite dimensional spaces. In Roland Glowinski, Gene H. Golub, Gérard A. Meurant, and Jacques Périaux, editors, *First International Symposium on Domain Decomposition Methods for Partial Differential Equations*, Philadelphia, PA, 1988. SIAM.

[2] P. Alart, M. Barboteu, P. Le Tallec, and M. Vidrascu. Additive Schwarz method for nonsymmetric problems : application to frictional multicontact problems. In *Thirteenth International Conference on Domain Decomposition Methods*, 2002. To appear.

[3] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17:886–905, 1996.

[4] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. `MUMPS`: a general purpose distributed memory sparse solver. In A. H. Gebremedhin, F. Manne, R. Moe, and T. Sørevik, editors, *Proceedings of PARA2000, the Fifth International Workshop on Applied Parallel Computing, Bergen, June 18-21*, pages 122–131. Springer-Verlag, 2000. Lecture Notes in Computer Science 1947.

[5] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[6] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, pages 501–520, 2000.

[7] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis, tuning and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software*, 27:388–421, 2001.

[8] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and P. Plecháč. PARASOL: An integrated programming environment for parallel sparse matrix solvers. In *PINEAPL Workshop, A Workshop on the Use of Parallel Numerical Libraries in Industrial End-user Applications, CERFACS, February*, 1998.

[9] M. Arioli, J. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM Journal on Matrix Analysis and Applications*, 10:165–190, 1989.

[10] M. Arioli, I. S. Duff, and D. Ruiz. Stopping criteria for iterative solvers. *SIAM Journal on Matrix Analysis and Applications*, 13:138–144, 1992.

[11] C. Ashcraft. The fan-both family of column-based distributed Cholesky factorisation algorithm. In A. George, J.R. Gilbert, and J.W.H Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 159–190. Springer-Verlag NY, 1993.

[12] J. Baglama, D. Calvetti, G. H. Golub, and L. Reichel. Adaptively preconditioned GMRES algorithms. *SIAM J. Scientific Computing*, 20(1):243–269, 1999.

[13] M. V. Bhat, W. G. Habashi, J. W. H. Liu, V. N. Nguyen, and M. F. Peeters. A note on nested dissection for rectangular grids. *SIAM Journal on Matrix Analysis and Applications*, 14(1):253–258, January 1993.

[14] Å. Björck. Numerics of Gram-Schmidt orthogonalization. *Linear Algebra and its Applications*, 197,198:297–316, 1994.

[15] P. E. Bjørstad, J. Koster, and P. Krzyżanowski. Domain decomposition solvers for large scale industrial finite element problems. In *PARA2000 Workshop on Applied Parallel Computing*. Lecture Notes in Computer Science 1947, Springer-Verlag, 2000.

[16] P. E. Bjørstad and O. B. Widlund. Solving elliptic problems on regions partitioned into substructures. In Garrett Birkhoff and Arthur Schoenstadt, editors, *Elliptic Problem Solvers II*, pages 245–256, New York, 1984. Academic Press.

[17] P. E. Bjørstad and O. B. Widlund. Iterative methods for the solution of elliptic problems on regions partitioned into substructures. *SIAM J. Numer. Anal.*, 23(6):1093–1120, 1986.

[18] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM Press, 1997.

[19] A. El Boukili. *Analyse mathématique et simulation numérique bidimensionnelle des équations des semi-conducteurs par l'approche éléments finis mixtes*. PhD thesis, Université Paris VI, Paris, France, 1995.

[20] A. Bouras and V. Frayssé. A relaxing strategy for inexact matrix-vector products for Krylov methods. Technical Report TR/PA/00/15, CERFACS, Toulouse, France, 2000.

[21] A. Bouras, V. Frayssé, and L. Giraud. A relaxation strategy for inner-outer linear solvers in domain decomposition methods. Technical Report TR/PA/00/17, CERFACS, Toulouse, France, 2000.

[22] J.-F. Bourgat, R. Glowinski, P. Le Tallec, and M. Vidrascu. Variational formulation and algorithm for trace operator in domain decomposition calculations. In Tony Chan, Roland Glowinski, Jacques Périaux, and Olof Widlund, editors, *Domain Decomposition Methods*, pages 3–16, Philadelphia, PA, 1989. SIAM.

[23] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The construction of preconditioners for elliptic problems by substructuring, I. *Math. Comp.*, 47(175):103–134, 1986.

[24] F. Brezzi and M. Fortin. *Mixed and Hybrid Finite Element Methods*. Number 15 in Springer Series in Computational Mathematics. Springer-Verlag, 1991.

[25] B. Carpentieri. *Sparse preconditioners for dense linear systems from electromagnetic applications*. PhD thesis, CERFACS, Toulouse, France, 2002.

[26] L. M. Carvalho. *Preconditioned Schur complement methods in distributed memory environments*. PhD thesis, INPT/CERFACS, France, october 1997. TH/PA/97/41, CERFACS.

[27] L. M. Carvalho, L. Giraud, and G. Meurant. Local preconditioners for two-level non-overlapping domain decomposition methods. *Numerical Linear Algebra with Applications*, 8(4):207–227, 2001.

[28] L. M. Carvalho, L. Giraud, and P. Le Tallec. Algebraic two-level preconditioners for the schur complement method. *SIAM J. Scientific Computing*, 22(6):1987–2005, 2001.

[29] F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations*. SIAM, Philadelphia, 1996.

[30] T. F. Chan. Rank revealing QR-factorizations. *Linear Algebra and its Applications*, 88/89:67–82, 1987.

[31] T. F. Chan and T. P. Mathew. The interface probing technique in domain decomposition. *SIAM J. on Matrix Analysis and Applications*, 13(1):212–238, 1992.

[32] T. F. Chan and T. P. Mathew. Domain decomposition algorithms. In *Acta Numerica 1994*, pages 61–143. Cambridge University Press, 1994.

[33] T. F. Chan and D. C. Resasco. A survey of preconditioners for domain decomposition. Technical Report /DCS/RR-414, Yale University, 1985.

[34] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000.

[35] S. Chandrasekaran and I. Ipsen. On rank-revealing factorizations. *SIAM Journal on Matrix Analysis and Applications*, 15:592–622, 1994.

[36] R. K. Coomer and I. G. Graham. Massively parallel methods for semiconductor device modelling. *Computing*, 56(1):1–27, 1996.

[37] R. W. Cottle. Manifestations of the Schur complement. *Linear Algebra and its Applications*, 8:189–211, 1974.

[38] Y.-H. De Roeck. *Résolution sur Ordinateurs Multi-Processeurs de Problème d'Elasticité par Décomposition de Domaines*. PhD thesis, Université Paris IX Daupine, 1991.

[39] Y.-H. De Roeck and P. Le Tallec. Analysis and test of a local domain decomposition preconditioner. In Roland Glowinski, Yuri Kuznetsov, Gérard Meurant, Jacques Périaux, and Olof Widlund, editors, *Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 112–128. SIAM, Philadelphia, PA, 1991.

[40] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, October 1999.

[41] J. Dongarra, S. Moore, and A. Trefethen. Numerical libraries and tools for scalable parallel cluster computing. *The International Journal of High Performance Computing Applications*, 15(2):175–180, Summer 2001.

[42] M. Dryja. A capacitance matrix method for Dirichlet problem on polygon region. *Numer. Math.*, 39:51–64, 1982.

[43] M. Dryja, B. F. Smith, and O. B. Widlund. Schwarz analysis of iterative substructuring algorithms for elliptic problems in three dimensions. *SIAM J. Numer. Anal.*, 31(6):1662–1694, 1993.

[44] M. Dryja and O. B. Widlund. An additive variant of the Schwarz alternating method for the case of many subregions. Technical Report 339, also Ultracomputer Note 131, Department of Computer Science, Courant Institute, 1987.

[45] M. Dryja and O. B. Widlund. Towards a unified theory of domain decomposition algorithms for elliptic problems. In T. Chan, R. Glowinski, J. Périaux, and O. Widlund, editors, *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 3–21. SIAM, Philadelphia, PA, 1990.

[46] I. S. Duff. Sparse numerical linear algebra: direct methods and preconditioning. In I. S. Duff and G. A. Watson, editors, *The State of the Art in Numerical Analysis*, pages 27–62, Oxford, 1997. Oxford University Press.

[47] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, London, 1986.

[48] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[49] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.

[50] J. Erhel, K. Burrage, and B. Pohl. Restarted GMRES preconditioned by deflation. *J. Comput. Appl. Math.*, 69:303–318, 1996.

[51] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1995.

[52] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM Journal on Numerical Analysis*, 21(2):352–362, April 1984.

[53] Qing Fan, P. A. Forsyth, J. R. F. McMacken, and Wei-Pai Tang. Performance issues for iterative solvers in device simulation. *SIAM Journal on Scientific Computing*, 17(1):100–117, January 1996.

[54] C. Farhat and F.-X. Roux. A Method of Finite Element Tearing and Interconnecting and its Parallel Solution Algorithm. *Int. J. Numer. Meth. Engng.*, 32:1205–1227, 1991.

[55] V. Frayssé and L. Giraud. A set of conjugate gradient routines for real and complex arithmetics. Technical Report TR/PA/00/47, CERFACS, Toulouse, France, 2000.

[56] V. Frayssé, L. Giraud, and S. Gratton. A set of GMRES routines for real and complex arithmetics. Technical Report TR/PA/97/49, CERFACS, Toulouse, France, 1997.

[57] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM : Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[58] J. A. George. Solution of linear systems of equations: direct methods for finite-element problems. In J.R. Bunch and D. J. Rose, editors, *Sparse Matrix Techniques.Lecture notes in mathematics, 572.*, pages 52–101. Springer-Verlag, 1977.

[59] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[60] L. Giraud. Combining shared and distributed memory programming models on clusters of symmetric multiprocessors: Some basic promising experiments. Working Note WN/PA/01/19, CERFACS, Toulouse, France, 2001.

[61] L. Giraud, J. Koster, A. Marrocco, and J.-C. Rioual. Domain decomposition methods in semiconductor device modeling. Technical Report TR/PA/01/51, CERFACS, Toulouse, France, 2001. To appear in the proceedings of the 13^{th} conference on Domain Decomposition Methods in Scientific Computing, 2002.

[62] L. Giraud and R. S. Tuminaro. Domain decomposition algorithms for the drift-diffusion equations. In R. Sincovec, D. Keyes, M. Leuze, L. Petzold, and D. Reed, editors, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 719–726. SIAM, 1993.

[63] L. Giraud and R. S. Tuminaro. Schur complement preconditioners for anisotropic problems. *IMA Journal of Numerical Analysis*, 19(1):1–17, 1999. Also published as Sandia Nat. Lab Technical Report 98-8488J.

[64] R. Glowinski and P. Le Tallec. Augmented lagrangian and operator splitting methods in nonlinear mechanics. *SIAM, Studies in Applied Mathematics*, 1989.

[65] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins Univ. Press, 1989. Second Edition.

[66] G. H. Golub and Q. Ye. Inexact preconditioned conjugate gradient method with inner-outer iteration. Sccm-97-04, Stanford University, 1997.

[67] Gene Golub and G. Meurant. Matrices, moments and quadrature II. How to compute the error in iterative methods. *BIT*, 37:687–705, 1997.

[68] A. Greenbaum. *Iterative methods for solving linear systems*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[69] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference. Volume 2, The MPI-2 Extensions*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, USA, second edition, 1998.

[70] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT press, 1994.

[71] F. Hecht and A. Marrocco. Mixed finite element simulation of heterojunction structures including a boundary layer model for the quasi-fermi levels. *COMPEL*, 13(4):757–770, 1994.

[72] B. Hendrickson and R. Leland. The CHACO User's Guide. Version 1.0. Technical Report SAND93-2339 • UC-405, Sandia National Laboratories, Albuquerque, 1993.

[73] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear system. *J. Res. Nat. Bur. Stds.*, B49:409–436, 1952.

[74] HSL. A collection of Fortran codes for large scale scientific computation, 2000. http://www.cse.clrc.ac.uk/Activity/HSL.

[75] G. Karypis and V. Kumar. METIS, unstructured graph partitioning and sparse matrix ordering system. version 2.0. Technical report, University of Minnesota, Department of Computer Science, Minneapolis, MN 55455, August 1995.

[76] C. T. Kelley. *Iterative methods for optimization*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1999.

[77] D. E. Keyes and W. D. Gropp. A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation. *SIAM Journal on Scientific and Statistical Computing*, 8(2):166–202, 1987.

[78] S. A. Kharchenko and A. Yu. Yeremin. Eigenvalue translation based preconditioners for the GMRES(k) method. *Numerical Linear Algebra with Applications*, 2(1):51–77, 1995.

[79] P. Laug. How to implement MODULEF. Technical Report RT-0069, Inria, Institut National de Recherche en Informatique et en Automatique, 1986.

[80] P.-L. Lions. On the Schwarz alternating method I. In Roland Glowinski, Gene H. Golub, Gérard A. Meurant, and Jacques Périaux, editors, *First International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 1–42, Philadelphia, PA, 1988. SIAM.

[81] P. L. Lions. On the Schwarz alternating method II. In Tony Chan, Roland Glowinski, Jacques Périaux, and Olof Widlund, editors, *Domain Decomposition Methods*, pages 47–70, Philadelphia, PA, 1989. SIAM.

[82] P. L. Lions. On the Schwarz alternating method III: a variant for nonoverlapping subdomains. In Tony F. Chan, Roland Glowinski, Jacques Périaux, and Olof Widlund, editors, *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations , held in Houston, Texas, March 20-22, 1989*, Philadelphia, PA, 1990. SIAM.

[83] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and Practice. *SIAM Review*, 34:82–109, 1992.

[84] J. Mandel. Balancing domain decomposition. *Comm. Numer. Meth. Engrg.*, 9:233–241, 1993.

[85] J. Mandel and R. Tezaur. Convergence of substructuring method with Lagrange multipliers. *Numer. Math.*, 73:473–487, 1996.

[86] A. Marrocco. Simulations numériques de dispositifs électroniques via éléments finis mixtes, adaptation de maillage et décomposition de domaine. Rapport de recherche, INRIA, To appear.

[87] R. B. Morgan. A restarted GMRES method augmented with eigenvectors. *SIAM Journal on Matrix Analysis and Applications*, 16:1154–1171, 1995.

[88] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface. Technical Report Version 2.0, 2000.

[89] PARASOL. Deliverable 2.1d (final report): MUMPS Version 4.0. A MUltifrontal Massively Parallel Solver. Technical report, June 30, 1999.

[90] F. Pellegrini and J. Roman. Sparse matrix ordering with SCOTCH. In *Proceedings of HPCN'97, Vienna, LNCS 1225*, pages 370–378, April 1997.

[91] F. Pellegrini, J. Roman, and P. R. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. *Concurrency: Practice and Experience*, 12:69–84, 2000.

[92] A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer–Verlag, Berlin, 1994.

[93] J. R. Rice. Experiments on Gram-Schmidt orthogonalization. *Mathematics of Computation*, 20:325–328, 1966.

[94] D. Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical Report RT/APO/01/4, Département Informatique ENSEEIHT - IRIT, Toulouse, 2001.

[95] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

[96] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.

[97] H. A. Schwarz. Ueber einen Grenzübergang durch alternierendes Verfahren. *Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich*, 15:272–286, May 1870.

[98] S. Selberherr. *Analysis and simulation of semiconductor devices*. Springer Verlag, Wien, New York, 1984.

[99] J. N. Shadid and R. S. Tuminaro. Sparse iterative algorithm software for large-scale MIMD machines: An initial discussion and implementation. *Concurrency: Practice and Experience*, 4(6):481–497, 1992.

[100] B. F. Smith. *Domain Decomposition Algorithms for the Partial Differential Equations of Linear Elasticity*. PhD thesis, Courant Institute of Mathematical Sciences, September 1990. Tech. Rep. 517, Department of Computer Science, Courant Institute.

[101] B. F. Smith, P. E. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[102] P. Le Tallec. Domain decomposition methods in computational mechanics. In J. Tinsley Oden, editor, *Computational Mechanics Advances*, volume 1 (2), pages 121–220. North-Holland, 1994.

[103] P. Le Tallec, Y.-H. De Roeck, and M. Vidrascu. Domain-decomposition methods for large linearly elliptic three dimensional problems. *J. of Computational and Applied Mathematics*, 34, 1991.

[104] P. Le Tallec, J. Mandel, and M. Vidrascu. Balancing domain decomposition for plates. In *Domain Decomposition Methods in Scientific and Engineering Computing: Proceedings of the Seventh International Conference on Domain Decomposition*, volume 180 of *Contemporary Mathematics*, pages 515–524, Providence, Rhode Island, 1994. American Mathematical Society.

[105] P. Le Tallec, J. Mandel, and M. Vidrascu. A Neumann-Neumann domain decomposition algorithm for solving plate and shell problems. *SIAM Journal on Numerical Analysis*, 35(2):836–867, April 1998.

[106] G. Torres. Etude et implantation de méthodes de décomposition de domaine sur machine multiprocesseur à mémoire distribuée. Internal report, CERFACS, 1998.

[107] H. A. van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13:631–644, 1992.

[108] F. Guevara Vasquez. Internship report on domain decomposition methods for the solution of partial differential equations. Technical Report TR/PA/00/98, CERFACS, Toulouse, France, 2000.

[109] J. Warsa, M. Benzi, T. Wareing, and J. Morel. Preconditioning a mixed discontinuous finite element method for radiation diffusion. *Numerical Linear Algebra with Applications*, 2002. submitted.

[110] O. B. Widlund. Iterative substructuring methods: Algorithms and theory for elliptic problems in the plane. In Roland Glowinski, Gene H. Golub, Gérard A. Meurant, and Jacques Périaux, editors, *First International Symposium on Domain Decomposition Methods for Partial Differential Equations*, Philadelphia, PA, 1988. SIAM.

[111] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

[112] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford Science Publications, 1965.

[113] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2:77–79, 1981.