Journal of Computational Physics 227 (2008) 9408–9427

Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

Computing entries of the inverse of a sparse matrix using the FIND algorithm

S. Li [a], S. Ahmed [b], G. Klimeck [c], E. Darve [a,*]

[a] Institute for Computational and Mathematical Engineering, Stanford University, 496 Lomita Mall, Durand Building, Room 209, Stanford, CA 94305-4040, USA
[b] Electrical and Computer Engineering Department, Southern Illinois University Carbondale, IL 62901, USA
[c] Network for Computational Nanotechnology, Birck Nanotechnology Center, Purdue University, West Lafayette, IN 47907, USA

ARTICLE INFO

Article history:
Received 17 April 2007
Received in revised form 27 May 2008
Accepted 23 June 2008
Available online 17 July 2008

PACS:
02.60.Dc
02.70.−c
73.23.−b
73.63.−b

Keywords:
Nested dissection
Green's function
NEGF
Nanotransistor
Gaussian elimination
Sparse matrix

ABSTRACT

An accurate and efficient algorithm, called fast inverse using nested dissection (FIND), for computing non-equilibrium Green's functions (NEGF) for nanoscale transistors has been developed and applied in the simulation of a novel dual-gate metal-oxide-semiconductor field-effect transistor (MOSFET) device structure. The method is based on the algorithm of nested dissection. A graph of the matrix is constructed and decomposed using a tree structure. An upward and downward traversal of the tree yields significant performance improvements for both the speed and memory requirements, compared to the current state-of-the-art recursive methods for NEGF. This algorithm is quite general and can be applied to any problem where certain entries of the inverse of a sparse matrix (e.g., its diagonal entries, the first row or column, etc.) need to be computed. As such it is applicable to the calculation of the Green's function of partial differential equations. FIND is applicable even when complex boundary conditions are used, for example non reflecting boundary conditions.

1. Introduction

The non-equilibrium Green's function (NEGF) approach is being considered as a state-of-the-art modeling tool in predicting performance and designing emerging nanoscale devices. Development of multi-dimensional simulators based on the NEGF approach is crucial to capture both the quantum mechanical effects and the effect of scattering with phonons and other electrons. Despite the fact that transport issues for nano-transistors, nanowires and molecular electronic devices are very different from one another, they can be treated with the common formalism provided by the NEGF [1]. The approach is based on the coupled solution of the Schrödinger and Poisson equations. So far, the difficulties in understanding the various terms in the resultant equations and the computational burden needed for its actual implementation are perceived as great challenges. A successful utilization of the Green's function approach commercially is the nano-electronics modeling (NEMO) simulator [2], which is effectively 1D and is primarily applicable to resonant tunneling diodes. Accurate and reliable multi-dimensional modeling of realistic future nanoscale devices requires enormous computational efforts, yet the currently

* Corresponding author. Tel.: +1 650 725 2560; fax: +1 650 723 3525.
E-mail address: darve@stanford.edu (E. Darve).

0021-9991/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2008.06.033

available algorithms are prohibitively expensive. This paper focuses on an accurate and efficient implementation of the NEGF approach for 2D MOSFET device structures.

Our algorithm, fast inverse using nested dissection (FIND), reduces the computational cost of the most expensive part of NEGF, which is the solution of the Green's function equation for the electron density, which is then used in the Poisson equation. In a typical simulation, the Poisson equation needs to be solved self-consistently with the Schrödinger equation. Consequently, the electron density gets typically computed multiple times until convergence is achieved. This leads to huge computational costs which FIND can reduce by orders of magnitude.

The most expensive calculation is computing some of (but not all) the entries of the matrix G^r [1]:

$$G^r(E) = [EI - H - \Sigma]^{-1} = A^{-1} \quad \text{(retarded Green's function)} \tag{1}$$

and $G^<(E) = G^r\Sigma^<(G^r)^\dagger$ (less-than Green's function). In these equations, I is the identity matrix, and E is the energy level. $\dagger$ denotes the transpose conjugate of a matrix. The Hamiltonian matrix H describes the system at hand (e.g., nano-transistor). It is usually a sparse matrix with connectivity only between neighboring mesh nodes, except for nodes at the boundary of the device which may have a non-local coupling (e.g., non-reflecting boundary condition). The matrices Σ and $\Sigma^<$ correspond to the self energy and can be assumed to be diagonal matrices. See Svizhenko [3] for this terminology and notations. In this work, all these matrices are considered to be given and we will focus on the problem of efficiently computing some entries in G^r and $G^<$. As an example of entries which must be computed, the diagonal entries of G^r are required to compute the density of states, while the diagonal entries of $G^<$ allow computing the electron density. The current can be computed from the upper diagonal entries of $G^<$.

Even though the matrix A in Eq. (1) is, by the usual standards, a mid-size sparse matrix of size typically $10{,}000 \times 10{,}000$, computing the entries of $G^<$ is a major challenge since this operation is repeated at all energy levels for every iteration of the Poisson–Schrödinger solver. Overall, the diagonal of $G^<(E)$ for the different values of the energy level E can be computed as many as thousands of times.

The problem of computing certain entries of the inverse of a sparse matrix is relatively common in computational engineering. Examples include:

- *Least square fitting: in the linear least-square fitting procedure, coefficients a_k* are computed so that the error

$$\sum_i \left[Y_i - \sum_k a_k \phi_k(x_i) \right]^2$$

  is minimal, where (x_i, Y_i) are the data points. It can be shown, under certain assumptions that, in the presence of measurement errors in the observations Y_i, the error in the coefficients a_k is proportional to C_{kk} where C is the inverse matrix of A:

$$A_{jk} = \sum_i \phi_j(x_i)\phi_k(x_i)$$

- *Eigenvalues of tri-diagonal matrices: the inverse iteration method attempts to compute the eigenvector* $\boldsymbol{v}$ associated with eigenvalue λ by solving iteratively the equation

$$(A - \hat{\lambda} I)\boldsymbol{x}_k = s_k \boldsymbol{x}_{k-1}$$

  where $\hat{\lambda}$ is an approximation of λ and s_k is used for normalization. Varah [4] and Wilkinson [5–7] have extensively discussed optimal choices of starting vectors for this method. An important result is that, in general, choosing the vector $\boldsymbol{e}_l$ (lth vector in the standard basis), where l is the index of the column with the largest norm among all columns of $(A - \hat{\lambda} I)^{-1}$, is a nearly optimal choice. A good approximation can be obtained by choosing l such that the lth entry on the diagonal of $(A - \hat{\lambda} I)^{-1}$ is the largest among all diagonal entries.
- *Accuracy estimation: when solving a linear equation* $A\boldsymbol{x} = \boldsymbol{b}$, one is often faced with errors in A and $\boldsymbol{b}$, either because of uncertainties in physical parameters or inaccuracies in their numerical calculation. In general the accuracy in the computed solution $\boldsymbol{x}$ will depend on the condition number of A: $\|A\|\|A^{-1}\|$, which can be estimated from the diagonal entries of A and its inverse in some cases.
- *Sensitivity computation: when solving* $A\boldsymbol{x} = \boldsymbol{b}$, the sensitivity of x_i to A_{jk} is given by $\partial x_i / \partial A_{jk} = x_k (A^{-1})_{ij}$.

Many other examples can be found in the literature.

Currently the state-of-the-art is a method developed by Klimeck and Svizhenko et al. [3], called the recursive Green's function method (RGF). This approach can be shown to be the most efficient for "nearly 1D" devices, i.e. devices which are very elongated in one direction and very thin in the two other directions.

Assume that the matrix A is the result of discretizing a partial differential equation in 2D using a local stencil, e.g., with a 5 point stencil. Assume the mesh is the one given on Fig. 1.

For a 5 point stencil, the matrix A can be written as a tri-diagonal block matrix where blocks on the diagonal are denoted by A_q $(1 \leqslant i \leqslant n)$, on the upper diagonal by B_q $(1 \leqslant i \leqslant n-1)$, and on the lower diagonal by C_q $(2 \leqslant i \leqslant n)$.

RGF computes the diagonal of A^{-1} by computing recursively two sequences. The first sequence, in increasing order, is defined recursively as [3]:

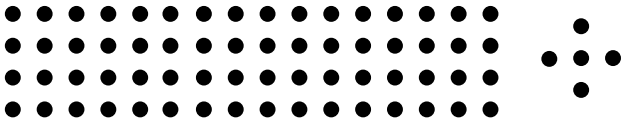

Fig. 1. Left: example of 2D mesh to which RGF can be applied. Right: 5-point stencil.

$$F_1 = A_1^{-1}$$
$$F_q = (A_q - C_q F_{q-1} B_{q-1})^{-1}$$

The second sequence, in decreasing order, is defined as:

$$G_n = F_n$$
$$G_q = F_q + F_q B_q G_{q+1} C_{q+1} F_q$$

The matrix G_q is in fact the q diagonal block of the inverse matrix G^r of A. If we denote by N_x the number of points in the cross section of the device and N_y along its length ($n = N_x N_y$) the cost of this method can be seen to be $\mathcal{O}(N_x^3 N_y)$. Therefore when N_x is small this is a computationally very attractive approach. The memory requirement is $\mathcal{O}(N_x^2 N_y)$.

FIND improves on RGF by reducing the computational cost to $\mathcal{O}(N_x^2 N_y)$ and the memory to $\mathcal{O}(N_x N_y \ln N_x)$. FIND follows some of the ideas of the nested dissection algorithm [8]. The mesh is decomposed into 2 subsets, which are further subdivided recursively into smaller subsets. A series of Gaussian eliminations are then performed, first going up the tree and then down, to finally yield entries in the inverse of A. Details are described in Sections 4 and 5.

In this article, we focus on the calculation of the diagonal of G^r and will reserve the extension to $G^<$ for a future publication. Nevertheless, we mention that we have already shown that such an extension is possible and gains similar to those described in this paper can be achieved.

As will be shown below, FIND can be applied to any 2D or 3D device, even though it is most efficient in 2D. The geometry of the device can be arbitrary as well as the boundary conditions. The only requirement is that the matrix A comes from a stencil discretization, i.e., points in the mesh should be connected only with their neighbors. The efficiency degrades with the extent of the stencil, i.e., nearest neighbor stencil vs. second nearest neighbor.

2. Review of existing methods

In addition to FIND which is based on nested dissection, other techniques have been developed to compute certain entries in the inverse of a sparse matrix. We will now review them. The conclusion, though, is that none of these techniques is applicable to our problem and hence this is our motivation for the development of FIND.

Takahashi et al. [9] observed that if the matrix A is decomposed using an LU factorization $A = LDU$ then:

$$G^r = A^{-1} = D^{-1}L^{-1} + (I - U)G^r, \quad \text{and} \tag{2}$$
$$G^r = U^{-1}D^{-1} + G^r(I - L). \tag{3}$$

Erisman et al. [10] applied this result to compute certain entries of G^r. Let's define a matrix C such that:

$$C_{ij} = \begin{cases} 1, & \text{if } L_{ij} \text{ or } U_{ij} \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

Erisman et al. showed the following theorem:

Any entry G^r_{ij} such that $C_{ji} = 1$ can be computed as a function of L, U, D and entries G^r_{pq} such that $p \geqslant i$, $q \geqslant j$, and $C_{qp} = 1$.

This implies that efficient recursive equations can be constructed. Specifically, from Eq. (2), for $i < j$:

$$G^r_{ij} = -\sum_{k=i+1}^{n} U_{ik} G^r_{kj}$$

The key observation is that if we want to compute G^r_{ij} with $L_{ji} \neq 0$ and $U_{ik} \neq 0$ ($k > i$) then C_{jk} must be equal to 1. This proves that the theorem holds in that case. A similar result holds for $j < i$ using Eq. (3). For $i = j$, we get [using Eq. (2)]:

$$G^r_{ii} = (D_{ii})^{-1} - \sum_{k=i+1}^{n} U_{ik} G^r_{ki}$$

Despite the appealing simplicity of this algorithm, it has the drawback that the method does not extend to the calculation of $G^<$ which is a key requirement in our problem.

K. Bowden developed an interesting set of matrix sequences which allows in principle to calculate the inverse of any block tri-diagonal matrices very efficiently [11]. Without going into the details, four sequences of matrices are defined using recurrence relations, K_p, L_p, M_p and N_p. Then an expression is found for any block (i,j) of matrix G^r, G^r_{ij}:

$$j \geqslant i, \quad G^{r}_{ij} = K_i N_0^{-1} N_j$$
$$j \leqslant i, \quad G^{r}_{ij} = L_i L_0^{-1} M_j$$

However the recurrence relation used to define the four sequences of matrices turns out to be unstable due to roundoff errors. Consequently this approach is not applicable to matrices of large size.

Schröder, Trottenberg et al. [12,13] created a method called total reduction which allows efficiently decimating the mesh by removing half of the nodes at each step. This is similar to the cyclic reduction approach [14]. This leads to a new matrix in the remaining unknowns which has a longer stencil than the original matrix. In the case of the Laplace equation, this stencil decays very fast and therefore can be approximated with an exponentially small error using a constant number of terms. This can be used to develop fast algorithms to compute the inverse matrix. However in the case of the Schrödinger equation, the stencil decays very slowly and cannot be truncated. After a couple of decimations, the matrix, even though much smaller, contain so many non-zero entries that overall no computational gain is achieved.

Consequently, we decided to start from the method of nested dissection [8]. This leads to an algorithm, FIND, which is exact (in the absence of roundoff errors) and stable. It allows computing entries in both G^r and $G^<$. This is asymptotically the fastest direct method. It is most efficient in 1D and 2D. It can be applied in 3D as well; even though it leads to a speed-up in 3D the scaling is not as advantageous in this case.

In this article, we will focus on the 2D case which is practically of most interest for this approach. We will describe the method in the context of a matrix A corresponding to the discretization of a partial differential equation with a stencil extending to the nearest neighbor only, and a structured rectangular mesh. This is a typical geometry and discretization used for modeling MOSFETs [3]. Extensions to longer stencils or non-rectangular meshes (even finite-element meshes) are possible but will be postponed to future publications.

3. Motivation, background, and description of the physical problem

For quite some time, semiconductor devices have been scaled aggressively in order to meet the demands of reduced cost per function on a chip used in modern integrated circuits. There are some problems associated with device scaling, however [15]. Critical dimensions, such as transistor gate length and oxide thickness, are reaching physical limitations. Considering the manufacturing issues, photolithography becomes difficult as the feature sizes approach the wavelength of ultraviolet light. In addition, it is difficult to control the oxide thickness when the oxide is made up of just a few monolayers. In addition to the processing issues, there are also some fundamental device issues. As the oxide thickness becomes very thin, the gate leakage current due to tunneling increases drastically. This significantly affects the power requirements of the chip and the oxide reliability. Short-channel effects, such as drain-induced barrier lowering, degrade the device performance. Hot carriers also degrade device reliability.

To fabricate devices beyond current scaling limits, integrated circuit companies are simultaneously pushing (1) the planar, bulk silicon complementary metal oxide semiconductor (CMOS) design while exploring alternative gate stack materials (high-k dielectric and metal gates), band engineering methods (using strained Si or SiGe [15–17]), and (2) alternative transistor structures that include primarily partially-depleted and fully-depleted silicon-on-insulator (SOI) devices. SOI devices are found to be advantageous over their bulk silicon counterparts in terms of reduced parasitic capacitances, reduced leakage currents, increased radiation hardness, as well as inexpensive fabrication process. IBM launched the first fully functional SOI mainstream microprocessor in 1999 predicting a 25–35% performance gain over bulk CMOS [18]. Today there is also an extensive research in double-gate structures, and FinFET transistors [15], which have better electrostatic integrity and theoretically have better transport properties than single-gated FETs. A number of non-classical and revolutionary technology such as carbon nanotubes and nanoribbons or molecular transistors have been pursued in recent years, but it is not quite obvious, in view of the predicted future capabilities of CMOS, that they will be competitive.

There is a virtual consensus that the most scalable MOSFET devices are double-gate SOI MOSFETs with a sub-10 nm gate length, ultra-thin, intrinsic channels and highly doped (degenerate) bulk electrodes – see, e.g., recent reviews [19,20] and Fig. 2. In such transistors, short channel effects typical of their bulk counterparts are minimized, while the absence of dopants in the channel maximizes the mobility. Such advanced MOSFETs may be practically implemented in several ways including planar, vertical, and FinFET geometries. However, several design challenges have been identified such as a process tolerance requirement of within 10% of the body thickness and an extremely sharp doping profile with a doping gradient of 1 nm/decade. The Semiconductor Industry Association forecasts that this new device architecture may extend MOSFETs to the 22 nm node (9 nm physical gate length) by 2016 [21]. Intrinsic device speed may exceed 1 THz and integration densities will be more than 1 billion transistors/cm^2. In this work, we have focused on this particular device structure and employed the NEGF method in the calculations of its transport properties.

The first step in the NEGF method to model nanoscale devices such as the double-gate SOI MOSFET in Fig. 2 is to identify a suitable basis set and Hamiltonian matrix for an isolated channel region. The self-consistent potential, which is a part of the Hamiltonian matrix, is included in this step. The second step is to compute the self-energy matrices, which describe how the channel couples to the source/drain contacts and to the scattering process. For simplicity, only ballistic transport is treated in this paper. After identifying the Hamiltonian matrix and the self-energies, the third step is to compute the retarded Green's

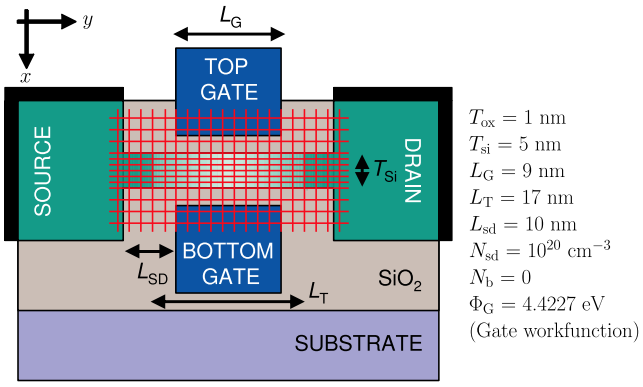

Fig. 2. The model of a widely-studied double-gate SOI MOSFET with ultra-thin intrinsic channel. Typical values of key device parameters are also shown.

function. Once the retarded Green's function is known, one can then calculate other Green's functions and determine the physical quantities of interest.

Recently, the NEGF approach has been applied in the simulations of two-dimensional MOSFET structures [3] as in Fig. 2. The intrinsic device is discretized using a 2D non-uniform spatial grid (with N_x and N_y nodes along the depth and length directions, respectively) with semi-infinite boundaries. Non-uniform spatial grids are essential to limit the total number of grid points while at the same time resolving physical features. The Hamiltonian of a valley b for electrons associated with the device under consideration is as follows:

$$H_b(\mathbf{r}) = -\frac{\hbar^2}{2}\left[\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{1}{m_x^b}\frac{\mathrm{d}}{\mathrm{d}x}\right) + \frac{\mathrm{d}}{\mathrm{d}y}\left(\frac{1}{m_y^b}\frac{\mathrm{d}}{\mathrm{d}y}\right) + \frac{\mathrm{d}}{\mathrm{d}z}\left(\frac{1}{m_z^b}\frac{\mathrm{d}}{\mathrm{d}z}\right)\right] + V(\mathbf{r})$$

where m_x^b, m_y^b and m_z^b are the components of effective mass in valley b. The equation of motion for the retarded Green's function (G^{r}) and less-than Green's function ($G^<$) are found to be (see [3] for details and notations):

$$\left[E - \frac{\hbar^2 k_z^2}{2m_z} - H_b(\mathbf{r}_1)\right] G_b^r(\mathbf{r}_1, \mathbf{r}_2, k_z, E) - \int \Sigma_b^r(\mathbf{r}_1, \mathbf{r}, k_z, E) G_b^r(\mathbf{r}, \mathbf{r}_2, k_z, E)\,\mathrm{d}\mathbf{r} = \delta(\mathbf{r}_1 - \mathbf{r}_2)$$

$$G_b^<(\mathbf{r}_1, \mathbf{r}_2, k_z, E) = \int G_b^r(\mathbf{r}_1, \mathbf{r}, k_z, E)\Sigma_b^<(\mathbf{r}, \mathbf{r}', k_z, E) G_b^r(\mathbf{r}_2, \mathbf{r}', k_z, E)^* \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{r}'$$

where * denotes the complex conjugate. Given G_b^{r} and $G_b^<$, the density of states (DOS) and the charge density, ρ, can be written as a sum of contributions from the individual valleys:

$$\mathrm{DOS}(\mathbf{r}, k_z, E) = \sum_b N_b(\mathbf{r}, k_z, E) = -\frac{1}{\pi}\sum_b \mathrm{Im}[G_b^r(\mathbf{r}, \mathbf{r}, k_z, E)]$$

$$\rho(\mathbf{r}, k_z, E) = \sum_b \rho_b(\mathbf{r}, k_z, E) = -i\sum_b G_b^<(\mathbf{r}, \mathbf{r}, k_z, E)$$

The self-consistent solution of the Green's function is often the most time intensive step in the simulation of electron density. RGF [3] is an efficient approach to computing the diagonal blocks of the discretized Green's function. The operation count required to solve for all elements of G_b^{r} scales as $N_x^3 N_y$ making it very expensive in this particular case. Note that RGF provides all the diagonal blocks of the matrix even though only the diagonal is really needed. Faster algorithms to solve for the diagonal elements with operation count smaller than $N_x^3 Ny$ have been, for the past few years, very desirable. Our newly developed FIND algorithm addresses this particular issue of computing the diagonal of G^{r} and $G^<$, thereby reducing the simulation time of NEGF significantly compared to the conventional RGF scheme.

4. Brief description of the algorithm for computing the diagonal of A^{-1}

The main idea of the algorithm is to perform many LU factorizations on the given matrix to compute the diagonal elements of its inverse. Each LU factorization allows computing one diagonal entry in A^{-1}; since the LU factorizations for all the diagonal entries overlap significantly, we reuse them and thus reduce the computational cost.

The next paragraphs give a more precise view of FIND. Details are kept for the next section.

For a given $n \times n$ matrix A, the last entry on the diagonal of its inverse is given by $A_{nn}^{-1} = 1/U_{nn}$, where $A = LU$ is the LU factorization of A. Therefore, the first issue is performing LU factorizations efficiently. By a proper reordering of the matrix which minimizes fill-ins (see the method of nested dissection of George et al. [8]), we can preserve most of the sparsity of the original matrix A and thus make the LU factorization very efficient.

Although we can only compute A^{-1}_{nn} in this way, we can choose any node and reorder the original matrix to make that node correspond to the (n,n) entry of the reordered matrix. In this way, all the diagonal elements of A^{-1} can be computed.

The second issue is that if we have to perform a full LU factorization for each of the n reordered matrices, the algorithm will not be computationally efficient even though each LU factorization is very fast. However, many intermediate results of the LU factorizations are identical. If we reorder those matrices properly, perform the LU factorizations in the right order, and save these intermediate results, we can reduce the computational cost considerably. In an optimal implementation, computing all the diagonal entries of A^{-1} has the same O complexity as computing a single diagonal entry (e.g., $O(N^{3/2})$ for a square mesh with N nodes). This is shown in Sections 5.4 and 7.

We now give the details of the algorithm in Section 5 and rigorous proofs of its correctness in Section 6 (i.e., the algorithm indeed produces the desired result).

5. Detailed description of the algorithm

The non-zero entries of a matrix A can be represented using a graph where each node corresponds to a row or column of the matrix. If an entry A_{ij} is non-zero, we create an edge (possibly directed) between node i and j. In our case, each row or column in the matrix can be assumed to be associated with a node of a computational mesh. FIND is based on a tree decomposition of this graph. Even though different trees can be used, we will assume that a binary tree is used in which the mesh is first subdivided into 2 sub-meshes (also called clusters of nodes), each sub-mesh is subdivided into 2 sub-meshes and so on (see Fig. 7). For each cluster, we can define three important sets:

- *Boundary set: this is the set of all mesh nodes in the cluster which have a connection with a node outside the set.*
- *Inner set: this is the set of all mesh nodes in the cluster which do not have a connection with a node outside the set.*
- *Adjacent set: this is the set of all mesh nodes outside the cluster which have a connection with a node outside the set.*

This is illustrated in Fig. 3. This is the case where each node is connected to its nearest neighbor as in a 5 point stencil. This can be generalized to more complex connectivities.

5.1. Upward pass

The first stage of the algorithm, or upward pass, consists in eliminating all the inner mesh nodes contained in the tree clusters. We first eliminate all the inner mesh nodes contained in the leaf clusters, then proceed to the next level in the tree. Re-using the previous elimination, the remaining inner mesh nodes are again eliminated. This recursive process is shown in Fig. 4.

Notation: $\overline{C}$ will denote the complement of C (that is all the mesh nodes not in C). The adjacent set of C is then always the boundary set of $\overline{C}$. The following notation for matrices will be used in this text:

$$M = [a_{11} \quad a_{12} \quad \dots \quad a_{1n}; \quad a_{21} \quad a_{22} \quad \dots \quad a_{2n}; \quad \dots]$$

denotes a matrix with the vector $[a_{11} \quad a_{12} \quad \dots \quad a_{1n}]$ on the first row and the vector $[a_{21} \quad a_{22} \quad \dots \quad a_{2n}]$ on the second (and so on for other rows). The same notation is used when a_{ij} is a matrix. $A(U,V)$ denotes the submatrix of A obtained by extracting the rows (resp. columns) corresponding to mesh nodes in clusters U (resp. V).

We now define the notation U_C. Assume we eliminate all the inner mesh nodes of cluster C from matrix A. Denote the resulting matrix A_{C+} (the notation $C+$ has a special meaning described in more details in the proof of correctness of the algorithm) and B_C the boundary set of C. Then,

$$U_C = A_{C+}(B_C, B_C)$$

To be completely clear about the algorithm we describe in more details how an elimination is performed. Assume we have a matrix formed by 4 blocks A, B, C and D where A has p columns:

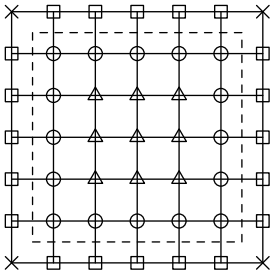

Fig. 3. Cluster and definition of the sets. The cluster is composed of all the mesh nodes inside the dash line. The triangles form the inner set, the circles the boundary set and the squares the adjacent set. The crosses are mesh nodes outside the cluster which are not connected to the mesh nodes in the cluster.

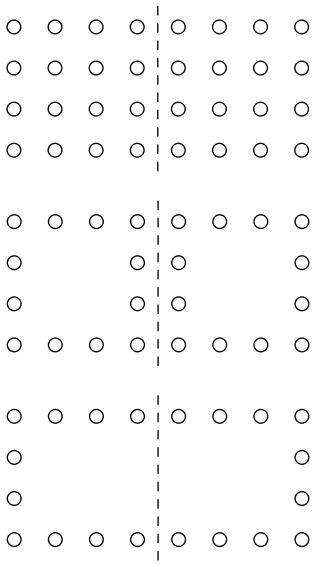

Fig. 4. The top figure is a 4×8 cluster with two 4×4 child clusters separated by the dash line. The middle figure shows the result of eliminating the inner mesh nodes in the child clusters. The bottom figure shows the result of eliminating the inner mesh nodes in the 4×8 cluster. The elimination from the middle row can be re-used to obtain the elimination at the bottom.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

The process of elimination of the first p columns in this matrix consists in computing an "updated block D" (denoted D^*) given by the formula:

$$D^* = D - CA^{-1}B$$

The matrix D^* can also be obtained by performing a Gaussian elimination on $[A\ B; C\ D]$ and stopping after p steps.

The pseudo-code for procedure eliminateInnerNodes implements this elimination procedure.

Procedure `eliminateInnerNodes(`*cluster C*`)`. This procedure should be called with the root of the tree: eliminateInnerNodes(root).

Data: tree decomposition of the mesh; the matrix A.
Input: cluster C with n boundary mesh nodes.
Output: all the inner mesh nodes of cluster C are eliminated by the procedure. The $n \times n$ matrix U_C with the result of the elimination is saved.

```
1  if C is not a leaf then
2      C1 = left child of C;
3      C2 = right child of C;
4      eliminateInnerNodes(C1)    /* The boundary set is denoted B_C1 */;
5      eliminateInnerNodes(C2)    /* The boundary set is denoted B_C2 */;
6      A_C = [U_C1 A(B_C1,B_C2); A(B_C2,B_C1) U_C2];
       /* A(B_C1,B_C2) and A(B_C2,B_C1) are values from the original
          matrix A.                                                  */
7  else
8      A_C = A(C,C);
9  if C is not the root then
10     Eliminate from A_C the mesh nodes which are inner nodes of C; U_C is the
       resulting matrix;
11     Save U_C;
```

5.2. Downward pass

The second stage of the algorithm, or downward pass, consists in removing all the mesh nodes which are outside of a leaf cluster. This stage re-uses the elimination computed during the first stage. Denote C_1 and C_2 the two children of the root cluster (which is the entire mesh). Denote C_{11} and C_{12} the two children of C_1. If we re-use the elimination of the inner mesh nodes of C_2 and C_{12}, we can efficiently eliminate all the mesh nodes which are outside of C_1 and do not belong to its adjacent set, i.e., the inner mesh nodes of $\overline{C}_1$. This is illustrated in Fig. 5.

The process then continues in a similar fashion down to the leaf clusters. A typical situation is depicted in Fig. 6. Once we have eliminated all the inner mesh nodes of $\overline{C}$, we proceed to its children C_1 and C_2. Take C_1 for example. To remove all the inner mesh nodes of $\overline{C}_1$, similar to the elimination in the upward pass, we simply need to remove some nodes in the boundary sets of $\overline{C}$ and C_2 because $\overline{C}_1 = \overline{C} \cup C_2$. The complete algorithm is given in procedure eliminateOuterNodes. For completeness we give the list of subroutines to call to perform the entire calculation:

```
1. treeBuild(A)      /* This routine is not described in this paper */;
2. eliminateInnerNodes(root);
3. eliminateOuterNodes(root);
```

Procedure `eliminateOuterNodes(`*cluster C*`)`. This procedure should be called with the root of the tree: eliminateOuterNodes(root).

Data: tree decomposition of the mesh; the matrix A; the upward pass [eliminateInnerNodes()] should have been completed.
Input: cluster C with n adjacent mesh nodes.
Output: all the inner mesh nodes of cluster $\bar{\mathrm{C}}$ are eliminated by the procedure. The $n \times n$ matrix $\mathrm{U}_{\bar{\mathrm{C}}}$ with the result of the elimination is saved.

```
1  if C is not the root then
2      D = parent of C        /* The boundary set of D̄ is denoted B_D̄ */;
3      D1 = sibling of C      /* The boundary set of D1 is denoted B_D1 */;
4      A_C̄ = [U_D̄ A(B_D̄,B_D1); A(B_D1,B_D̄) U_D1];
       /* A(B_D̄,B_D1) and A(B_D1,B_D̄) are values from the original matrix
          A.                                                          */
       /* If D is the root, then D̄ = ∅ and A_C̄ = U_D1.                */
5      Eliminate from A_C̄ the mesh nodes which are inner nodes of C̄; U_C̄ is the
       resulting matrix;
6      Save U_C̄;
7  if C is not a leaf then
8      C1 = left child of C;
9      C2 = right child of C;
10     eliminateOuterNodes(C1);
11     eliminateOuterNodes(C2);
12 else
13     Calculate [A^-1](C,C) using Equation 4;
```

Once we have reached the leaf clusters, the calculation is almost complete. Take a leaf cluster C. At this stage in the algorithm, we have computed $U_{\overline{C}}$. Denote by $A_{\overline{C}_+}$ the matrix obtained by eliminating all the inner mesh nodes of $\overline{C}$ (all the nodes except the squares and circles in Fig. 6); $A_{\overline{C}_+}$ contains mesh nodes in the adjacent set of C (i.e., the boundary set of $\overline{C}$) and the mesh nodes of C:

$$A_{\overline{C}_+} = \begin{bmatrix} U_{\overline{C}} & A(B_{\overline{C}}, C) \\ A(C, B_{\overline{C}}) & A(C, C) \end{bmatrix}$$

The entries of $[A^{-1}](C,C)$ are given by:

$$[A^{-1}](C,C) = [A(C,C) - A(C,B_{\overline{C}})(U_{\overline{C}})^{-1}A(B_{\overline{C}},C)]^{-1} \tag{4}$$

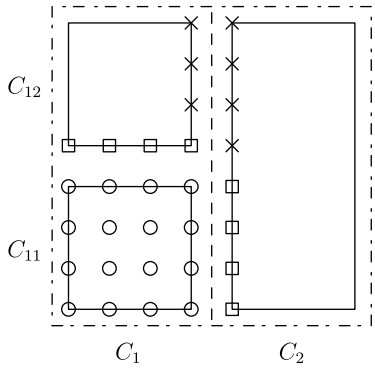

Fig. 5. The first step in the downward pass of the algorithm. Cluster C_1 is on the left and cluster C_2 on the right. The circles are mesh nodes in cluster C_{11}. The mesh nodes in the adjacent set of C_{11} are denoted by squares; they are not eliminated at this step. The crosses are mesh nodes which are either in the boundary set of C_{12} or C_2. These nodes need to be eliminated at this step. The dash dotted line around the figure goes around the entire computational mesh (including mesh nodes in C_2 which have already been eliminated). There are no crosses in the top left part of the figure because these nodes are inner nodes of C_{12}.

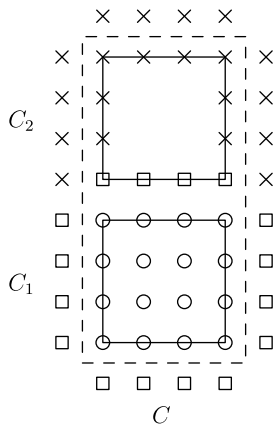

Fig. 6. A further step in the downward pass of the algorithm. Cluster C has two children C_1 and C_2. As previously, the circles are mesh nodes in cluster C_1. The mesh nodes in the adjacent set of C_1 are denoted by squares; they are not eliminated at this step. The crosses are nodes which need to be eliminated. They belong either to the adjacent set of C or the boundary set of C_2.

5.3. Nested dissection algorithm of George et al.

The algorithm in this paper uses a nested dissection-type approach similar to the nested dissection algorithm of George et al. [8]. We highlight the similarities and differences to help the reader relate our new approach, FIND, to these well-established methods. Both approaches are based on a similar nested dissection of the mesh. George's algorithm is used to solve a linear system whereas our algorithm calculates entries in the inverse of the matrix. The new objective requires multiple LU factorizations. This can be done in a computationally efficient manner if partial factorizations are re-used. For example in the sub-routine eliminateInnerNodes, the Gaussian elimination for cluster C reuses the result of the elimination for C1 and C2 by using as input U_{C1} and U_{C2}. See Procedure eliminateInnerNodes. The matrix U_C produced by the elimination is then saved. Note that Procedure eliminateInnerNodes could be used, with small modifications, to implement George's algorithm.

The second sub-routine eliminateOuterNodes reuses results from both eliminateInnerNodes and eliminateOuterNodes. See Procedure eliminateOuterNodes. The matrices $U_{\overline{D}}$ (produced by eliminateOuterNodes) and U_{D1} (produced by eliminateInnerNodes) are being reused. The matrix obtained after elimination, $U_{\overline{C}}$, is then saved.

To be able to reuse these partial LU factorizations, FIND requires more independence among the partial eliminations compared to George's algorithm. As a result, FIND uses "separators" [8] with double the width. This is required for reusing the partial elimination results in particular during the downward pass. Interestingly this makes FIND easy to parallelize even though we do not discuss this point in this paper.

5.4. Sketch of the computational complexity

We sketch a derivation of the computational complexity. A more detailed derivation is given in Section 7. Assume for simplicity that the mesh is square. We denote N the total number of mesh nodes. We assume moreover that N is of the form

$N = (2^l)^2$ and that the leaf clusters in the tree contain only a single node. The cost of operating on a cluster of size $2^p \times 2^p$ both in the upward and downward passes is $O((2^p)^3)$, because the size of both adjacent set and boundary set is of order 2^p. There are $N/(2^p)^2$ such clusters at each level, and consequently the cost per level is $O(N2^p)$. The total cost is therefore simply $O(N2^l) = O(N^{3/2})$. This is the same computational cost (in the O sense) as the nested dissection algorithm of George et al. [8]. It is now apparent that FIND has the same order of computational complexity as a single LU factorization.

6. Proof of correctness of the algorithm

In this section, we derive a formal proof of the correctness of the algorithm. The proof relies primarily on the properties of the Gaussian elimination and the definition of the boundary set, inner set, and adjacent set. These sets can be defined in very general cases (unstructured grids, etc). In fact, at least symbolically, the operations to be performed depend only on the graph defined by the matrix. Consequently it is possible to derive a proof of correctness in a very general setting. This is reflected by the relatively general and formal presentation of the proof.

The algorithm is based on a tree decomposition of the mesh (similar to a domain decomposition). However in the proof we define an augmented tree which essentially contains the original tree and in addition a tree associated with the complement of each cluster ($\overline{C}$ in our notation). The reason for this is that it allows us to present the algorithm in a unified form where the upward and downward passes can be viewed as traversing a single tree: the augmented tree. Even though from an algorithmic standpoint, the augmented tree is not needed (and perhaps make things more complicated), from a theoretical and formal standpoint, this is actually a natural graph to consider.

6.1. The definition and properties of mesh node sets and trees

We start this section by defining M as the set of all the nodes in the mesh. If we partition M into subsets C_i, each C_i being a cluster of mesh nodes, we can build a binary tree with its leaf nodes corresponding to these clusters. We denote such tree as $T_0 = \{C_i\}$. The subsets C_i are defined recursively in the following way: Let $C_1 = M$, then partition C_1 into C_2 and C_3, then partition C_2 into C_4 and C_5, C_3 into C_6 and C_7, and partition each C_i into C_{2i} and C_{2i+1}, until C_i reaches the predefined minimum size of the clusters in the tree. In T_0, C_i and C_j are the two children of C_k iff $C_i \cup C_j = C_k$ and $C_i \cap C_j = \emptyset$, i.e., $\{C_i, C_j\}$ is a partition of C_k. Fig. 7 shows the partitioning of the mesh and the binary tree T_0, where for notation simplicity, we use the subscripts of the clusters to stand for the clusters.

Let $C_{-i} = \overline{C}_i = M \setminus C_i$. Now we can define an *augmented tree* T_r^+ with respect to a leaf node $C_r \in T_0$ as $T_r^+ = (\{C_j | C_r \subseteq C_{-j}, j < -3\}) \cup (T_0 \setminus \{C_j | C_r \subseteq C_j, j > 0\})$. Such augmented tree is constructed to partition C_{-r} in a way similar to T_0, i.e., in T_r^+, C_i and C_j are the two children of C_k iff $C_i \cup C_j = C_k$ and $C_i \cap C_j = \emptyset$. In addition, since $C_2 = C_{-3}$, $C_3 = C_{-2}$ and $C_{-1} = \emptyset$, the tree nodes $C_{\pm 1}$, C_{-2}, and C_{-3} are removed from T_r^+ to avoid redundancy. Two examples of such augmented tree are shown in Fig. 8.

We denote by I_i the inner nodes of cluster C_i and B_i the boundary nodes as defined in Section 5. Then we recursively define the set of *private inner nodes* of C_i as $S_i = I_i \setminus \cup_{C_j \subset C_i} S_j$ with $S_i = I_i$ if C_i is a leaf node in T_r^+, where C_i and $C_j \in T_r^+$. Fig. 9 shows these subsets for a 4×8 cluster.

Now we study the properties of these subsets. To make the main text short and easier to follow, we only list below two important properties without proof. For other properties and their proofs, please see Appendix A.

The following property shows two different ways of looking at the same subset. This change of view happens repeatedly in our algorithm.

Property 3. *If C_i and C_j are the two children of C_k, then $S_k \cup B_k = B_i \cup B_j$ and $S_k = (B_i \cup B_j) \setminus B_k$.*

The next property is important in that it shows that the whole mesh can be partitioned into subsets S_i, B_{-r}, and C_r. Such property makes it possible to define a consistent ordering.

Property 4. *For any given augmented tree T_r^+ and all $C_i \in T_r^+$, S_i, B_{-r}, and C_r are all disjoint and $M = (\cup_{C_i \in T_r^+} S_i) \cup B_{-r} \cup C_r$.*

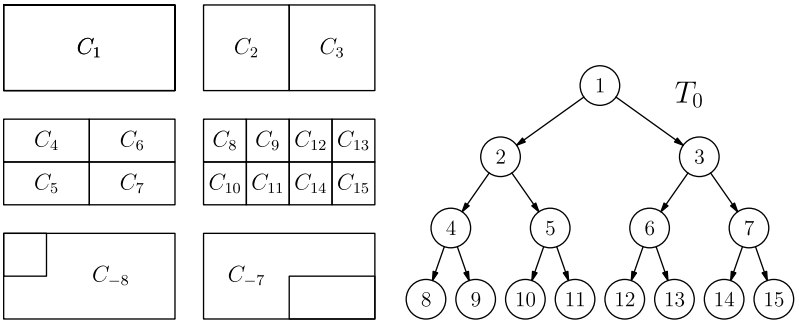

Fig. 7. The mesh and its partitions. $C_1 = M$.

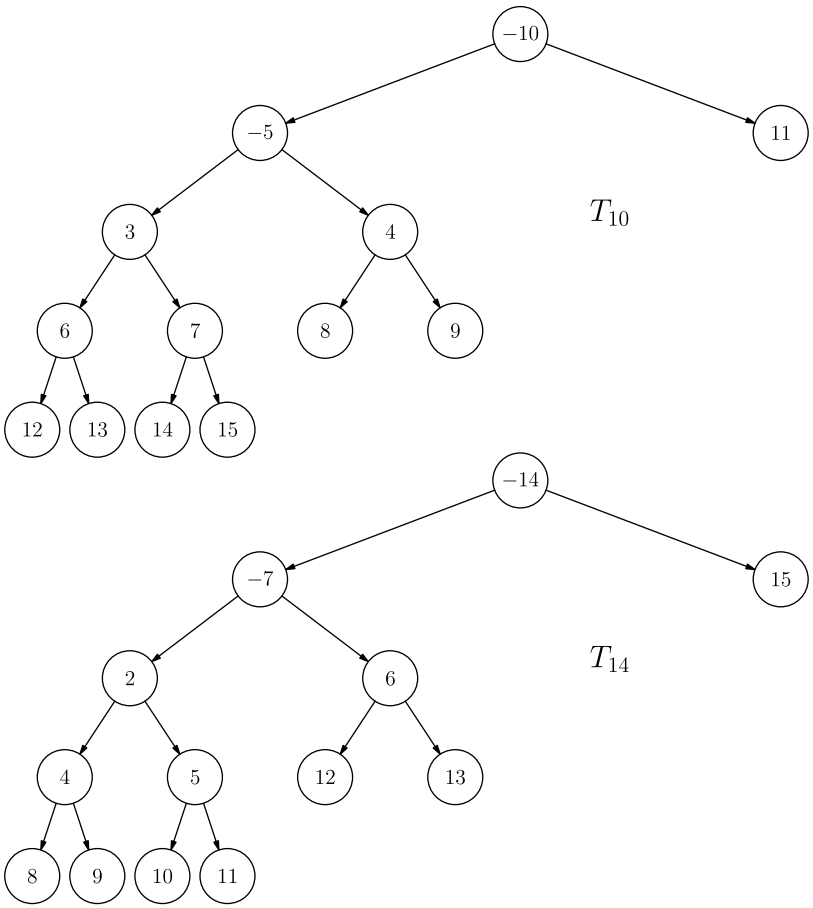

Fig. 8. Examples of augmented trees.

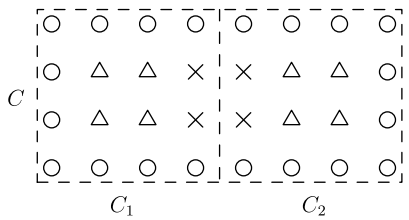

Fig. 9. Cluster C has two children C_1 and C_2. The inner nodes of cluster C_1 and C_2 are shown using triangles. The private inner nodes of C are shown with crosses. The boundary set of C is shown using circles.

Now consider the ordering of A. For a given submatrix $A(U,V)$, if all the indices corresponding to U appear before the indices corresponding to V, we say $U < V$. We define a *consistent ordering of A with respect to* T_r^+ as any ordering such that

(1) The indices of nodes in S_i are contiguous;
(2) $C_i \subset C_j$ implies $S_i < S_j$; and
(3) The indices corresponding to B_{-r} and C_r appear at the end.

Since it is possible that $C_i \not\subset C_j$ and $C_j \not\subset C_i$, we can see that the consistent ordering of A with respect to T_r^+ is not unique. For example, if C_j and C_k are the two children of C_i in T_r^+, then both orderings $S_j < S_k < S_i$ and $S_k < S_j < S_i$ are consistent. When we discuss the properties of the Gaussian elimination of A, all the properties apply to any consistent ordering so we do not make distinction below among different consistent orderings. From now on, we assume all the matrices we deal with have a consistent ordering.

6.2. Correctness of the algorithm

The major task of showing the correctness of the algorithm is to prove that the partial eliminations introduced in Section 5 are independent of one another, and that they produce the same result for matrices with consistent ordering; therefore, from an algorithmic point of view, these eliminations can be reused.

We now study the properties of the Gaussian elimination for a matrix A with consistent ordering.

Notations: For a given A, the order of S_i is determined so we can write the indices of S_i as $i_1, i_2, \ldots$, etc. For notation convenience, we write $\cup_{S_j<S_i} S_j$ as $S_{<i}$ and $(\cup_{S_i<S_j} S_j) \cup B_{-r} \cup C_r$ as $S_{>i}$. If $g = i_j$ then we denote $S_{i_{j+1}}$ by S_{g+}. If i is the index of the last S_i in the sequence, which is always $-r$ for T_r^+, then S_{i+} stands for B_{-r}. When we perform a Gaussian elimination, we eliminate the columns of A corresponding to the mesh nodes in each S_i from left to right as usual. We do not eliminate the last group of columns that correspond to C_r, which remains unchanged until we compute the diagonal of A^{-1}. Starting from $A_{i_1} = A$, we define the following intermediate matrices for each $g = i_1, i_2, \ldots, -r$ as the results of each step of Gaussian elimination:

$$A_{g+} = \text{the result of eliminating the } S_g \text{ columns in } A_g$$

Since the intermediate matrices depend on the ordering of the matrix A, which depends on T_r^+, we also sometimes denote them explicitly as $A_{r,i}$ to indicate the dependency.

Example. Let us consider Fig. 8 for cluster 10. In that case, a consistent ordering is: $S_{12}, S_{13}, S_{14}, S_{15}, S_6, S_7, S_8, S_9, S_3, S_4, S_{-5}, S_{11}, S_{-10}, B_{-10}, C_{10}$. The sequence i_j is: $i_1 = 12$, $i_2 = 13$, $i_3 = 14$, ..., etc. Pick $i = 15$, then $S_{<i} = S_{12} \cup S_{13} \cup S_{14}$. Pick $i = -5$, then $S_{>i} = S_{11} \cup S_{-10} \cup B_{-10} \cup C_{10}$. For $g = i_j = 15$, $S_{g+} = S_6$.

The first theorem in this section shows that the matrix preserves a certain sparsity pattern during the elimination process such that eliminating the S_i columns only affects the (B_i, B_i) entries. The precise statement of Theorem 1 is in the appendix with a proof. The following two matrices show one step of the elimination, with the right pattern of 0s:

$$A_i = \begin{pmatrix} A_i(S_{<i}, S_{<i}) & A_i(S_{<i}, S_i) & A_i(S_{<i}, B_i) & A_i(S_{<i}, S_{>i} \setminus B_i) \\ 0 & A_i(S_i, S_i) & A_i(S_i, B_i) & 0 \\ 0 & A_i(B_i, S_i) & A_i(B_i, B_i) & A_i(B_i, S_{>i} \setminus B_i) \\ 0 & 0 & A_i(S_{>i} \setminus B_i, B_i) & A_i(S_{>i} \setminus B_i, S_{>i} \setminus B_i) \end{pmatrix}$$

$$\Rightarrow$$

$$A_{i+} = \begin{pmatrix} A_i(S_{<i}, S_{<i}) & A_i(S_{<i}, S_i) & A_i(S_{<i}, B_i) & A_i(S_{<i}, S_{>i} \setminus B_i) \\ 0 & A_i(S_i, S_i) & A_i(S_i, B_i) & 0 \\ 0 & 0 & A_{i+}(B_i, B_i) & A_i(B_i, S_{>i} \setminus B_i) \\ 0 & 0 & A_i(S_{>i} \setminus B_i, B_i) & A_i(S_{>i} \setminus B_i, S_{>i} \setminus B_i) \end{pmatrix}$$

where $A_{i+}(B_i, B_i) = A_i(B_i, B_i) - A_i(B_i, S_i) A_i(S_i, S_i)^{-1} A_i(S_i, B_i)$.

Note: in the above matrices, for notation convenience, $A_i(\bullet, B_i)$ is written as a block. In reality, however, it is usually not a block for any A with consistent ordering.

Because the matrix preserves the sparsity pattern during the elimination process, we know that certain entries remain unchanged. More specifically, Corollaries 1 and 2 can be used to determine when the entries $(B_i \cup B_j, B_i \cup B_j)$ remain unchanged. Corollary 3 shows that the entries corresponding to leaf nodes remain unchanged until their elimination. Such properties are important when we compare the elimination process of matrices with different orderings. For proofs of these corollaries, please see Appendix A.

Corollary 1. *If C_i and C_j are the two children of C_k, then $A_k(B_i, B_j) = A(B_i, B_j)$ and $A_k(B_j, B_i) = A(B_j, B_i)$.*

Corollary 2. *If C_i is a child of C_k, then $A_k(B_i, B_i) = A_{i+}(B_i, B_i)$.*

These two corollaries tell us that when we are about to eliminate the mesh nodes in S_k based on B_i and B_j, we can use the entries (B_i, B_j) and (B_j, B_i) from the original matrix A, and the entries (B_i, B_i) and (B_j, B_j) obtained after elimination of S_i and S_j.

Corollary 3. *If C_i is a leaf node in T_r^+, then $A_i(C_i, C_i) = A(C_i, C_i)$.*

This corollary tells us that we can use the entries from the original matrix A for leaf clusters (even though other mesh nodes may have already been eliminated at that point).

Based on Theorem 1 and the above three corollaries, we can compare the partial elimination results of matrices with different orderings:

Theorem 2. *For any r and s such that $C_i \in T_r^+$ and $C_i \in T_s^+$, we have:*

$$A_{r,i}(S_i \cup B_i, S_i \cup B_i) = A_{s,i}(S_i \cup B_i, S_i \cup B_i)$$

Proof. If C_i is a leaf node, then by Corollary 3, we have $A_{r,i}(S_i \cup B_i, S_i \cup B_i) = A_{r,i}(C_i, C_i) = A_r(C_i, C_i) = A_s(C_i, C_i) = A_{s,i}(C_i, C_i) = A_{s,i}(S_i \cup B_i, S_i \cup B_i)$

If the equality holds for i and j such that C_i and C_j are the two children of C_k, then

- By Theorem 1, we have $A_{r,i+}(B_i,B_i) = A_{s,i+}(B_i,B_i)$ and $A_{r,j+}(B_j,B_j) = A_{s,j+}(B_j,B_j)$.
- By Corollary 2, we have $A_{r,k}(B_i,B_i) = A_{r,i+}(B_i,B_i) = A_{s,i+}(B_i,B_i) = A_{s,k}(B_i,B_i)$ and $A_{r,k}(B_j,B_j) = A_{r,j+}(B_j,B_j) = A_{s,j+}(B_j,B_j) = A_{s,k}(B_j,B_j)$.
- By Corollary 1, we have $A_{r,k}(B_i,B_j) = A_r(B_i,B_j) = A_s(B_i,B_j) = A_{s,k}(B_i,B_j)$ and $A_{r,k}(B_j,B_i) = A_r(B_j,B_i) = A_s(B_j,B_i) = A_{s,k}(B_j,B_i)$.

Now we have $A_{r,k}(B_i \cup B_j, B_i \cup B_j) = A_{s,k}(B_i \cup B_j, B_i \cup B_j)$. By Property 4, we have $A_{r,k}(S_k \cup B_k, S_k \cup B_k) = A_{s,k}(S_k \cup B_k, S_k \cup B_k)$. By mathematical induction, the theorem is proved. □

If we go one step further, based on Theorems 1 and 2, we have the following corollary:

Corollary 4. *For any r and s such that $C_i \in T_r^+$ and $C_i \in T_s^+$, we have:*

$$A_{r,i+}(B_i,B_i) = A_{s,i+}(B_i,B_i)$$

Theorem 2 and Corollary 4 show that the partial elimination results are common for matrices with different orderings during the elimination process, which is the key foundation of our algorithm.

6.3. The algorithm

Corollary 4 shows that $A_{\bullet,i+}(B_i,B_i)$ is the same for all augmented trees, so we can have the following definition for any r:

$$U_i = A_{r,i+}(B_i,B_i)$$

By Theorem 1, Corollaries 1 and 2, for all i, j, and k such that C_i and C_j are the two children of C_k, we have

$$U_k = A_{r,k}(B_k,B_k) - A_{r,k}(B_k,S_k)A_{r,k}(S_k,S_k)^{-1}A_{r,k}(S_k,B_k) \tag{5}$$

where

$$A_{r,k}(B_k,B_k) = \begin{pmatrix} U_i(B_k\cap B_i, B_k\cap B_i) & A(B_k\cap B_i, B_k\cap B_j) \\ A(B_k\cap B_j, B_k\cap B_i) & U_j(B_k\cap B_j, B_k\cap B_j)\end{pmatrix}$$

$$A_{r,k}(S_k,S_k) = \begin{pmatrix} U_i(S_k\cap B_i, S_k\cap B_i) & A(S_k\cap B_i, S_k\cap B_j) \\ A(S_k\cap B_j, S_k\cap B_i) & U_j(S_k\cap B_j, S_k\cap B_j)\end{pmatrix}$$

$$A_{r,k}(B_k,S_k) = \begin{pmatrix} U_i(B_k\cap B_i, S_k\cap B_i) & 0 \\ 0 & U_j(B_k\cap B_j, S_k\cap B_j)\end{pmatrix}$$

and

$$A_{r,k}(S_k,B_k) = \begin{pmatrix} U_i(S_k\cap B_i, B_k\cap B_i) & 0 \\ 0 & U_j(S_k\cap B_j, B_k\cap B_j)\end{pmatrix}$$

If C_k is a leaf node, then by Corollary 3, we have $A_{r,k}(B_k,B_k) = A_r(B_k,B_k)$, $A_{r,k}(S_k,S_k) = A_r(S_k,S_k)$, $A_{r,k}(B_k,S_k) = A_r(B_k,S_k)$, and $A_{r,k}(S_k,B_k) = A_r(S_k,B_k)$.

By Eq. (5), we can compute U for upper level clusters based on (i) the original matrix A and (ii) the values of U for lower level clusters. The values of U for leaf nodes can be computed directly through Gaussian elimination. The last step of the elimination is shown below:

	$S_{<-r}$	S_{-r}	B_{-r}	C_r			$S_{<-r}$	S_{-r}	B_{-r}	C_r
$S_{<-r}$	×	×	×	0		$S_{<-r}$	×	×	×	0
S_{-r}	0	×	×	0	⇒	S_{-r}	0	×	×	0
B_{-r}	0	×	×	×		B_{-r}	0	0	×	×
C_r	0	0	×	×		C_r	0	0	×	×

In Eq. (5), we do not make distinction between positive tree nodes and negative tree nodes. We simply look at the augmented tree, eliminate all the private inner nodes, and get the corresponding boundary nodes updated. This makes the theoretical description of the algorithm more concise and consistent. When we turn the update rule into an algorithm, however, we do not actually construct the augmented tree. Instead, we use the original tree and treat the positive tree nodes and negative tree nodes separately.

Since no negative node is the descendant of any positive node in the augmented trees, we can first compute U_i for all $i > 0$. The relations among the positive tree nodes are the same in the original tree T_0 and in the augmented trees T_r^+, so we go through T_0 to compute U_i level by level from the bottom up and call it the *upward pass*. This is done in procedure *eliminateInnerNodes* of the algorithm in Section 5.1.

Once all the positive tree nodes have been computed, we compute U_i for all the negative tree nodes. For these nodes, if C_i and C_j are the two children of C_k in T_0, then C_{-k} and C_j are the two children of C_{-i} in T_r^+. Since C_{-k} is a descendant of C_{-i} in T_r^+ if

and only if C_i is a descendant of C_k in T_0, we compute all the U_i for $i < 0$ by going through T_0 level by level from the top down and call it the *downward pass*. This is done in the procedure *eliminateOuterNodes* in Section 5.2.

7. Complexity analysis

7.1. Running time analysis

In this section, we will analyze the most computationally intensive operations in the algorithm and give the asymptotic behavior of the computational cost. By Eq. 5, we see that the computational cost for U_i is $T \approx B_i^2 S_i + S_i^3/3 + B_i S_i^2$ flops, where both B_i and S_i are of order a for clusters of size $a \times a$. For a squared mesh, since the number of clusters in each level is proportional to a^{-2}, the computational cost for each level is proportional to a and forms a geometric series. As a result, the top level dominates the computational cost and the total computational cost is of the same order as the computational cost of the topmost level. We now study the complexity in more details in the two passes below.

Before we start, we want to emphasize the distinction between a squared mesh and an elongated mesh. In both cases, we want to keep all the clusters in the cluster tree to be as close as possible to square. For a squared mesh $N_x \times N_x$, we can keep all the clusters in the cluster tree to be either of size $a \times a$ or $a \times 2a$. For an elongated mesh of size $N_x \times N_y$, where $N_y \gg N_x$, we cannot do the same thing. Let the level of clusters each of size $N_x \times (2N_x)$ be *level L*, then all the clusters above level L cannot be of size $a \times 2a$. We will see the mergings and partitionings for clusters below and above the level L follow have different behaviors.

In the upward pass, as a typical case, $a \times a$ clusters merge to $a \times 2a$ clusters and then to $2a \times 2a$ clusters. In the first merge, the size of B_k is at most $6a$ and the size of S_k is at most $2a$. The time to compute U_k is $T \leqslant (6^2 \times 2 + 2^3/3 + 2^2 \times 6)a^3 \leqslant \frac{296}{3}a^3$ flops and the per node cost is at most $\frac{148/3}{a}$ flops, depending on the size of each node. In the second merge, the size of B_k is at most $8a$ and the size of S_k is at most $4a$. The time to compute U_k is $T \leqslant (8^2 \times 4 + 4^3/3 + 4^2 \times 8)a^3 \leqslant \frac{1216}{3}a^3$ and the per node cost is at most $\frac{304}{3}a$. So we have the following level-by-level running time for a mesh of size $N_x \times N_y$ with leaf nodes of size $a \times a$ for merges of clusters below level L:

$$\frac{148}{3}N_xN_ya \xrightarrow{\times 2} \frac{304}{3}N_xN_ya \xrightarrow{\times 1} \frac{296}{3}N_xN_ya \xrightarrow{\times 2} \frac{608}{3}N_xN_ya \ldots$$

We can see that the cost doubles from merging $a \times a$ clusters to merging $a \times 2a$ clusters while remains the same from merging $a \times 2a$ clusters to merging $2a \times 2a$ clusters. This is mostly because the size of S doubles in the first change while remains the same in the second change, as seen in Fig. 10. In Fig. 10, the upper figure shows the merging of two $a \times a$ clusters into an $a \times 2a$ cluster and the lower figure corresponds to two $a \times 2a$ clusters merging into a $2a \times 2a$ cluster. The arrows point to all the mesh nodes belonging to the set, e.g., B_k is the set of all the boundary nodes of C_k in the top figure. Also note that the

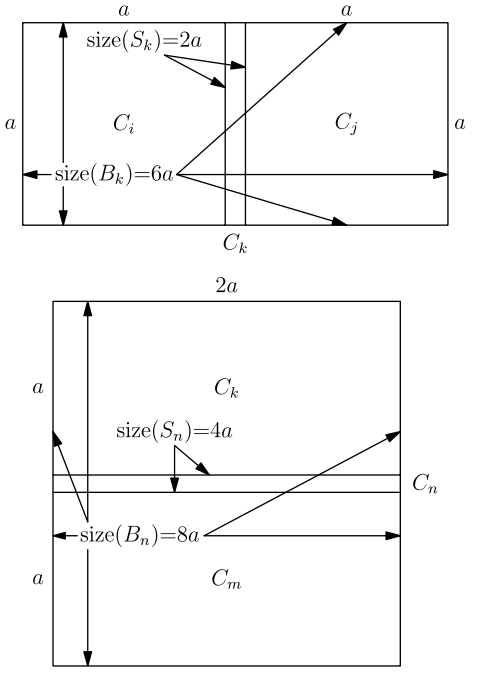

Fig. 10. Merging clusters below level L.

actual size of the sets could be a little smaller than the number shown in the figure. We will talk about this at the end of this section.

For clusters above level L, the computational cost for each merging remains the same since both the size of B and the size of S are $2N_x$. The computational cost for each merging is $T \approx \frac{56}{3}N_x^3 \approx 19N_x^3$ flops. This is shown in Fig. 11. Since for clusters above level L, we have only half mergings in the parent level compared to the child level, the cost decreases geometrically for levels above level L.

Adding all the computational costs together, the total computational cost in the upward pass is

$$T \leqslant 151N_xN_y(a + 2a + \cdots + N_x/2) + 19N_x^3(N_y/2N_x + N_y/4N_x + \cdots + 1) \approx 151_x^2N_y + 19N_x^2N_y = 170N_x^2N_y$$

In the downward pass, similar to the upward pass, for clusters above level L, the computational cost for each partitioning remains the same since both the size of B and the size of S are $2N_x$. Since the size of B and S is the same, the computational cost for each merge is also the same: $T \approx \frac{56}{3}N_x^3 \approx 19N_x^3$ flops. This is shown in Fig. 12.

For clusters below level L, the cost for each level begins decreasing as we go downward in the cluster tree. When $2a \times 2a$ clusters are partitioned to $a \times 2a$ clusters, the size of B_k is at most $6a$ and the size of S_k is at most $8a$. Similar to the analysis for upward pass, the time to compute U_k is $T \leqslant 422a \times 4a^2$. When $a \times 2a$ clusters are partitioned to $a \times a$ clusters, the size of B_k is at most $4a$ and the size of S_k is at most $6a$. The time to compute U_k is $T \leqslant 312a \times 2a^2$.

So we have the following level-by-level running time per node, starting from $N_x \times N_x$ down to the leaf clusters:

$$\ldots 422a \overset{\times 1.35}{\rightarrow} 312a \overset{\times 1.48}{\rightarrow} 211a \overset{\times 1.35}{\rightarrow} 156a$$

We can see that the computational cost changes more smoothly compared to that in the upward pass. This is because both the size of B and the size of S increase relatively smoothly, as shown in Fig. 13.

The computational cost in the downward pass is

$$\begin{aligned} T &\leqslant 734N_xN_ya + 734 \times 2N_xN_ya + \cdots + 734N_xN_yN_x/2 + 19N_x^3(N_y/2N_x) + 19N_x^3(N_y/4N_x) + \cdots + 19N_x^3 \\ &\approx 734N_x^2N_y + 19N_x^2N_y = 753N_x^2N_y \text{flops} \end{aligned}$$

So the total computational cost is

$$T \approx 170N_x^2N_y + 753N_x^2N_y = 923N_x^2N_y \text{flops}$$

The cost for the downward pass is significantly larger than that for the upward pass because the size of sets B's and S's are significantly larger.

In the above analysis, some of our estimates were not very accurate because we did not consider minor costs of the computation. For example, during the upward pass (similar during the downward pass):

- when the leaf clusters are not 2×2, we need to consider the cost of eliminating the inner mesh nodes of leaf clusters,
- the sizes of B and S are also different for clusters on the boundary of the mesh, where the connectivity of the mesh nodes is different from that of the inner mesh nodes.

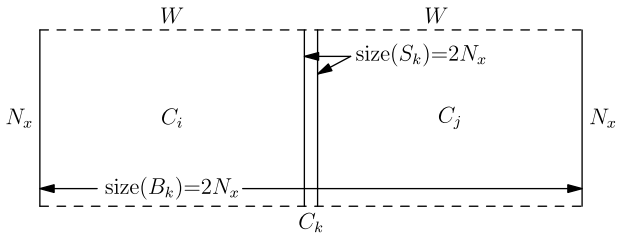

Fig. 11. Merging rectangular clusters. Two $N_x \times W$ clusters merge into an $N_x \times 2W$ cluster.

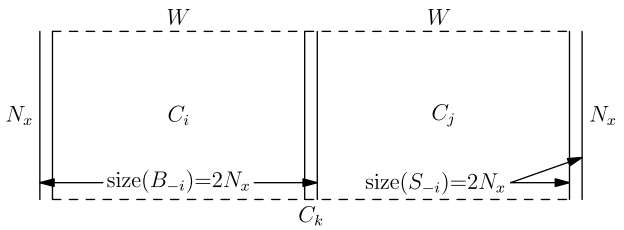

Fig. 12. Partitioning of clusters above level L.

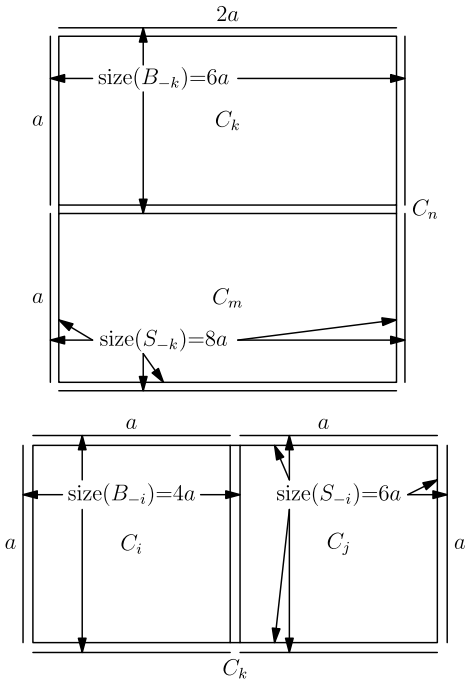

Fig. 13. Partitioning of clusters below level L.

Such inaccuracy, however, becomes less and less significant as the size of the mesh becomes bigger and bigger. It does not affect the asymptotic behavior of running time and memory cost either.

7.2. Memory cost analysis

Since the negative tree nodes are the ascendants of the positive tree nodes in the augmented trees, the downward pass needs U_i for $i > 0$, which are computed during the upward pass. Since these matrices are not used immediately, we need to store them for each positive node. This is where the major memory cost comes from and we will only analyze this part of the memory cost.

Let the memory storage for one matrix entry be one unit. Starting from the root cluster until level L, the memory for each cluster is about the same while the number of clusters doubles in each level. Each U_i is of size $2N_x \times 2N_x$ so the memory cost for each node is $4N_x^2$ units and the total cost is

$$\sum_{i=0}^{\log_2(N_y/N_x)} (2^i \times 4N_x^2) \approx 8N_xN_y\text{units}$$

Below level L, we maintain the clusters to be of size either $a \times a$ or $a \times 2a$ by cutting across the longer edge. For a cluster of size $a \times a$, the size B is $4a$ and the memory cost for each cluster is $16a^2$ units. We have $N_y \times N_x/(a \times a)$ clusters in each level so we need $16N_xN_y$ units of memory for each level, which is independent of the size of the cluster in each level. For a cluster of size $a \times 2a$, the size B is $6a$ and the memory cost for each cluster is $36a^2$ units. We have $N_y \times N_x/(a \times 2a)$ clusters in each level so we need $18N_xN_y$ units of memory for each level, which is independent of the size of the cluster in each level. For simplicity of the estimation of the memory cost, we let $16 \approx 18$.

There are 2 $\log_2(N_x)$ levels in this part, so the memory cost needed in this part is about $32N_xN_y \log_2(N_x)$ units.

The total memory cost in the upward pass is thus:

$$8N_xN_y + 32N_xN_y\log_2(N_x) = 8N_xN_y(1 + 4\log_2(N_x))\text{units}$$

If all the numbers are double decision complex numbers, the cost will be $128N_xN_y(1 + 4 \log_2(N_x))$ bytes.

8. Simulation of device and comparison with RGF

To assess the performance and applicability and benchmark FIND against RGF, we applied these methods to the non self-consistent calculation of density-of-states and electron density in a realistic non-classical double-gate SOI MOSFETs as depicted in Fig. 2 with a sub-10 nm gate length, ultra-thin, intrinsic channels and highly doped (degenerate) bulk electrodes. In

such transistors, short channel effects typical for their bulk counterparts are minimized, while the absence of dopants in the channel maximizes the mobility and hence drive current density. The "active" device consists of two gate stacks (gate contact and SiO_2 gate dielectric) above and below a thin silicon film. The thickness of the silicon film is 5 nm. Using a thicker body reduces the series resistance and the effect of process variation but it also degrades the short channel effects. The top and bottom gate insulator thickness is 1 nm, which is expected to be near the scaling limit for SiO_2. For the gate contact, a metal gate with tunable work function, ϕ_G, is assumed, where ϕ_G is adjusted to 4.4227 to provide a specified off-current value of 4 μA/μm. The background doping of the silicon film is taken to be intrinsic, however, to take into account the diffusion of the dopant ions; the doping profile from the heavily doped S/D extensions to the intrinsic channel is graded with a coefficient of g which equals to 1 dec/nm. The doping of the S/D regions equals 1×10^{20} cm^{-3}. According to the ITRS road

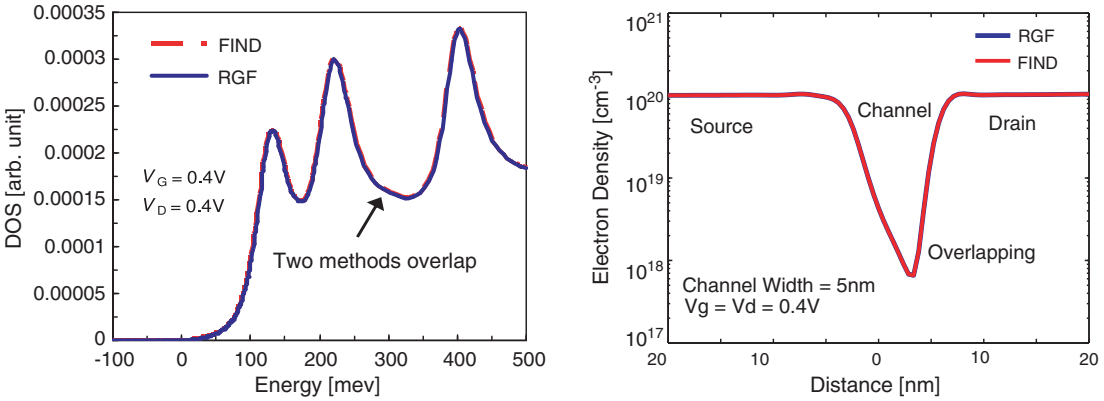

Fig. 14. Density-of-states (DOS) and electron density plots from RGF and FIND.

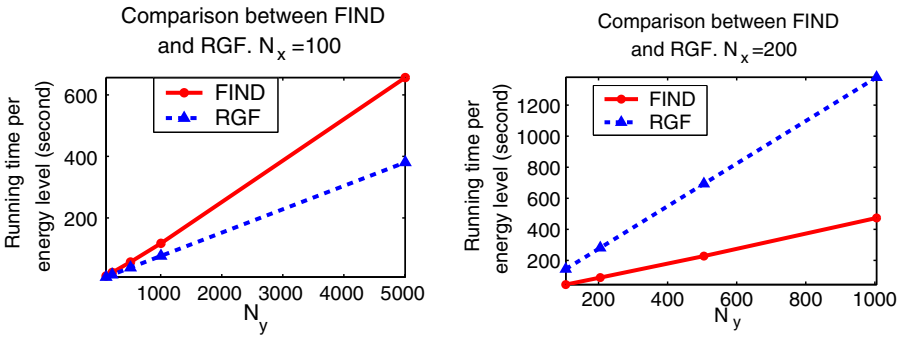

Fig. 15. Comparison of the running time of FIND and RGF when N_x is fixed.

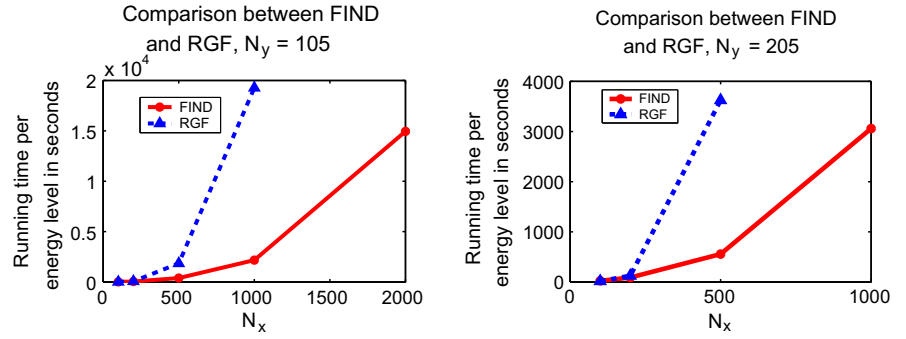

Fig. 16. Comparison of the running time between FIND and RGF when N_y is fixed.

map [21], the high performance logic device would have a physical gate length of L_G = 9 nm at the year 2016. The length, L_T, is an important design parameter in determining the on-current, while gate metal work function, ϕ_G, directly controls the off-current. The doping gradient, g, affects both on-current and off-current.

Fig. 14 shows that the RGF and FIND algorithms produce identical density of states and electron density. The code used in this simulation is nanoFET and is available on the nanoHUB (www.nanohub.org). The nanoHUB is a web-based resource for research, education, and collaboration in nanotechnology; it is an initiative of the NSF-funded Network for Computational Nanotechnology (NCN).

Fig. 15 shows the comparisons of running time between FIND and RGF. In the left figure, N_x = 100 and N_y ranges from 105 to 5005. In the right figure, N_x = 200 and N_y ranges from *105* to *1005*. We can see that FIND shows a considerable speed-up in the right figure when N_x = 200. The running time is linear with respect to N_y in both cases, as predicted in the computational cost analysis. The scaling with respect to N_x is different. It is equal to N_x^3 for RGF and N_x^2 for FIND.

Fig. 16 show the comparisons of running times between FIND and RGF when N_y's are fixed. We can see clearly the speed-up of FIND in the figure when N_x increases.

9. Conclusion

We have developed an efficient method of computing the diagonal entries of the retarded Green's function (density of states) and the diagonal of the less-than Green's function (density of charges). The algorithm is exact and uses Gaussian eliminations. A simple extension allows computing off diagonal entries for current density calculations. This algorithm can be applied to the calculation of any set of entries of A^{-1} where A is a sparse matrix.

In this paper, we described the algorithm and proved its correctness. We analyzed its computational and memory costs in details. Numerical results and comparisons with RGF confirmed the accuracy, stability and efficiency of FIND.

We considered an application to quantum transport in a nano-transistor. A 2D rectangular nano-transistor discretized with a mesh of size $N_x \times N_y$, $N_x < N_y$ was chosen. In that case, the cost is $O(N_x^2 N_y)$, which improves over the result of RGF [22], which scales as $O(N_x^3 N_y)$. This demonstrates that FIND allows simulating larger and more complex devices using a finer mesh and with a small computational cost. Our algorithm can be generalized to other structures such as nanowires, nanotubes, molecules and 3D transistors, with arbitrary shapes. FIND was incorporated in an open source code, nanoFET [23], which can be found on the nanohub web portal (www.nanohub.org).

Acknowledgments

We thank Dr. M.P. Anantram for his advice on the comparison with the RGF algorithm and realistic device simulation. We also thank the NSF for its generous funding support through the Network for Computational Nanotechnology (see www.ncn.purdue.edu). The NCN is a network of universities with a vision to pioneer the development of nanotechnology from science to manufacturing through innovative theory, exploratory simulation, and novel cyberinfrastructure. The use of nanoHUB.org computational resources operated by the Network for Computational Nanotechnology funded by the National Science Foundation (NSF) is acknowledged.

Appendix A. Properties, theorem, corollaries, and their proofs

In this appendix, we first list all the properties of the mesh node subsets with proofs if necessary, then the precise statement of Theorem 1 with its proof, and lastly the three corollaries with proofs, which were used in Section 6.2.

The first property is fairly simple but will be used again and again in proving theorems, corollaries, and other properties.

Property 1. *By definition of T_r^+, one of the following relations must hold: $C_i \subset C_j$, $C_i \supset C_j$, or $C_i \cap C_j = \emptyset$.*

The next property shows another way of looking at the inner mesh nodes. It will be used in the proof of Property 3.

Property 2. *$I_i = \cup_{C_j \subseteq C_i} S_j$, where C_i and $C_j \in T_r^+$.*

Proof. By the definition of S_i, we have $I_i \subseteq (I_i \setminus \cup_{C_j \subset C_i} S_j) \cup (\cup_{C_j \subset C_i} S_j) = S_i \cup (\cup_{C_j \subset C_i} S_j) = \cup_{C_j \subseteq C_i} S_j$, so it remains to show $\cup_{C_j \subseteq C_i} S_j \subseteq I_i$. Since $S_j \subseteq I_j$ and then $\cup_{C_j \subseteq C_i} S_j \subseteq \cup_{C_j \subseteq C_i} I_j$, if suffices to show $\cup_{C_j \subseteq C_i} I_j \subseteq I_i$.

To show this, we show that $C_j \subseteq C_i$ implies $I_j \subseteq I_i$. For any C_i, $C_j \in T_r^+$ and $C_j \subseteq C_i$, if $I_j \not\subseteq I_i$, then $I_j \backslash I_i \neq \emptyset$ and $I_j \subset C_j \subseteq C_i = I_i \cup B_i \Rightarrow I_j \backslash I_i \subseteq (I_i \cup B_i) \backslash I_i = B_i \Rightarrow I_j \backslash I_i = (I_j \backslash I_i) \cap B_i \subset I_j \cap B_i \Rightarrow I_j \cap B_i \neq \emptyset$, which contradicts the definition of I_i and B_i. So we have $I_j \subseteq I_i$ and the proof is complete. □

The following property shows that the whole mesh can be partitioned into subsets S_i, B_{-r}, and C_r, as has been stated in Section 6.1 and illustrated in Fig. 9.

Property 3. *If C_i and C_j are the two children of C_k, then $S_k \cup B_k = B_i \cup B_j$ and $S_k = (B_i \cup B_j) \backslash B_k$.*

Property 4. *For any given augmented tree T_r^+ and all $C_i \in T_r^+$, S_i, B_{-r}, and C_r are all disjoint and $M = (\cup_{C_i \in T_r^+} S_i) \cup B_{-r} \cup C_r$.*

Proof. To show that all the S_i in T_r^+ are disjoint, consider any S_i and S_k, $i \neq k$. If $C_i \cap C_k = \emptyset$, since $S_i \subseteq C_i$ and $S_k \subseteq C_k$, we have $S_i \cap S_k = \emptyset$. If $C_k \subset C_i$, then we have $S_k \subset \cup_{C_j \subset C_i} S_j$, and then by the definition of S_i, we have $S_i \cap S_k = \emptyset$. Similarly, if $C_i \subset C_k$, we have $S_i \cap S_k = \emptyset$ as well. By Property 1, the relation between any C_i, $C_k \in T_r^+$ must be one of the above three cases, so we have $S_i \cap S_k = \emptyset$.

Since $\forall C_i \in T_r^+$ we have $C_i \subseteq C_{-r}$, by Property 2, we have $S_i \subseteq I_{-r}$. Since $I_{-r} \cap B_{-r} = \emptyset$, we have $S_i \cap B_{-r} = \emptyset$. Since $C_{-r} \cap C_r = \emptyset$, $S_i \subset C_{-r}$, and $B_{-r} \subset C_{-r}$, we have $S_i \cap C_r = \emptyset$ and $B_{-r} \cap C_r = \emptyset$. By Property 2 again, we have $\cup_{C_i \in T_r^+} S_i = \cup_{C_i \subseteq C_{-r}} S_i = I_{-r}$. So we have $(\cup_{C_i \in T_r^+} S_i) \cup B_{-r} \cup C_r = (I_{-r} \cup B_{-r}) \cup C_r = C_{-r} \cup C_r = M$. □

Below we list properties of S_i for specific orderings.

Property 5. *If $S_i < S_j$, then $C_j \not\subset C_i$, which implies either $C_i \subset C_j$ or $C_i \cap C_j = \emptyset$.*

This property is straightforward from the definition of $S_i < S_j$ and Property 1.

The following two properties are related to the elimination process and will be used in the proofs of Theorems 1 and 2.

Property 6. *For any k,u such that $C_k, C_u \in T_r^+$, if $S_k < S_u$, then $S_u \cap I_k = \emptyset$.*

Proof. By Property 5, we have $C_u \not\subset C_k$. So for all j such that $C_j \subseteq C_k$, we have $j \neq u$ and thus $S_j \cap S_u = \emptyset$ by Property 3. By Property 2, $I_k = \cup_{C_j \subseteq C_k} S_j$, so we have $I_k \cap S_u = \emptyset$. □

Property 7. *If C_j is a child of C_k, then for any C_u such that $S_j < S_u < S_k$, we have $C_j \cap C_u = \emptyset$ and thus $B_j \cap B_u = \emptyset$.*

This is because the C_u can be neither a descendant of C_j nor an ancestor of C_k.

Proof. By Property 5, either $C_j \subset C_u$ or $C_j \cap C_u = \emptyset$. Since C_j is a child of C_k and $u \neq k$, we have $C_j \subset C_u \Rightarrow C_k \subset C_u \Rightarrow S_k < S_u$, which contradicts the given condition $S_u < S_k$. So $C_j \not\subset C_u$ and then $C_j \cap C_u = \emptyset$. □

Now we re-state Theorem 1 more precisely with its proof.

Theorem 1. *If we perform Gaussian elimination as described in Section 6.2 on the original matrix A with ordering consistent with any given T_r^+, then*

(1) $A_g(S_{\geqslant g}, S_{<g}) = 0$;
(2) $\forall h \geqslant g, A_g(S_h, S_{>h} \backslash B_h) = A_g(S_{>h} \backslash B_h, S_h) = 0$;
(3) (a) $A_{g+}(B_g, S_{>g} \backslash B_g) = A_g(B_g, S_{>g} \backslash B_g)$;
    (b) $A_{g+}(S_{>g} \backslash B_g, B_g) = A_g(S_{>g} \backslash B_g, B_g)$;
    (c) $A_{g+}(S_{>g} \backslash B_g, S_{>g} \backslash B_g) = A_g(S_{>g} \backslash B_g, S_{>g} \backslash B_g)$;
(4) $A_{g+}(B_g, B_g) = A_g(B_g, B_g) - A_g(B_g, S_g) A_g(S_g, S_g)^{-1} A_g(S_g, B_g)$.

The matrices A_i and A_{i+} show one step of elimination and may help understand this theorem.

Proof. Since (1) and (2) imply (3) and (4) for each i and performing Gaussian elimination implies (1), it is sufficient to prove (2). We will prove (2) by strong mathematical induction.

(1) For $g = i_1$, (2) holds because of the property of the original matrix, i.e., an entry in the original matrix is nonzero iff the corresponding two mesh nodes connect to each other and no mesh nodes in S_h and $S_{>h} \backslash B_h$ are connected to each other. The property is shown in the matrix below:

	S_h	B_h	$S_{>h} \setminus B_h$
S_h	×	×	0
B_h	×	×	×
$S_{>h} \setminus B_h$	0	×	×

(2) If (2) holds for all $g = j$ such that $S_j < S_k$, then by Property 1 we have either $C_j \subset C_k$ or $C_j \cap C_k = \emptyset$.
- if $C_j \subset C_k$, consider u such that $S_k < S_u$. By Property 6, we have $I_k \cap S_u = \emptyset$. Since $C_k = I_k \cup B_k$, we have $(S_u \backslash B_k) \cap C_k = \emptyset$. So we have $B_j \cap (S_u \backslash B_k) \subseteq (S_u \backslash B_k) \cup C_j \subseteq (S_u \backslash B_k) \cap C_k = \emptyset \Rightarrow B_j \cap (S_{>k} \backslash B_k) = \emptyset$.
- if $C_j \cap C_k = \emptyset$, then $B_j \subset C_j$ and $S_k \subset C_k \Rightarrow B_j \cap S_k = \emptyset$.

So in both cases, we have $(B_j, B_j) \cap (S_k, S_{>k} \backslash B_k) = \emptyset$. Since for every $S_j < S_k$, (1), (2), (3), and (4) hold, i.e., eliminating the S_j columns only affects the (B_j, B_j) entries, we have $A_k(S_k, S_{>k} \backslash B_k) = A_k(S_{>k} \backslash B_k, S_k) = 0$. Since the argument is valid for all $h \geqslant k$, we have $\forall h \geqslant k$, $A_k(S_h, S_{>h} \backslash B_h) = A_k(S_{>h} \backslash B_h, S_h) = 0$. So (2) holds for $g = k$ as well.

By strong mathematical induction, we have that (2) holds for all g such that $C_g \in T_r^+$. □

Below we restate Corollaries Corollaries 1–3 and give their proofs.

Corollary 1. *If C_i and C_j are the two children of C_k, then $A_k(B_i, B_j) = A(B_i, B_j)$ and $A_k(B_j, B_i) = A(B_j, B_i)$.*

Proof. Without loss of generality, let $S_i < S_j$. For any $S_u < S_k$, consider the following three cases: $S_u < S_j$, $S_u = S_j$, and $S_j < S_u < S_k$.
If $S_u < S_j$, then by Property 1, either $C_u \subset C_j$ or $C_u \cap C_j = \emptyset$.

- if $C_u \subset C_j$, then since $C_i \cap C_j = \emptyset$, we have $C_u \cap C_i = \emptyset \Rightarrow B_u \cap B_i = \emptyset$;
- if $C_u \cap C_j = \emptyset$, then since $B_u \subset C_u$ and $B_j \subset C_j$, we have $B_u \cap B_j = \emptyset$.

So we have $(B_u, B_u) \cap (B_i, B_j) = (B_u, B_u) \cap (B_j, B_i) = \emptyset$.
If $S_u = S_j$, then $B_u = B_j$ and $B_i \cap B_j = \emptyset$, so we also have $(B_u, B_u) \cap (B_i, B_j) = (B_u, B_u) \cap (B_j, B_i) = \emptyset$.
If $S_j < S_u < S_k$, then by Property 7, we have $B_u \cap B_j = \emptyset$. So we have $(B_u, B_u) \cap (B_i, B_j) = (B_u, B_u) \cap (B_j, B_i) = \emptyset$ as well.
So for every $S_u < S_k$, we have $(B_u, B_u) \cap (B_i, B_j) = (B_u, B_u) \cap (B_j, B_i) = \emptyset$. By Theorem 1, eliminating S_u only changes $A(B_u, B_u)$, so we have $A_k(B_i, B_j) = A(B_i, B_j)$ and $A_k(B_j, B_i) = A(B_j, B_i)$. □

Corollary 2. *If C_i is a child of C_k, then $A_k(B_i, B_i) = A_{i+}(B_i, B_i)$.*

Proof. Consider u such that $S_i < S_u < S_k$. By Property 7, we have $B_u \cap B_i = \emptyset$. By Theorem 1, eliminating S_u columns will only affect (B_u, B_u) entries, we have $A_k(B_i, B_i) = A_{i+}(B_i, B_i)$. □

Corollary 3. *If C_i is a leaf node in T_r^+, then $A_i(C_i, C_i) = A(C_i, C_i)$.*

Proof. Consider $S_u < S_i$, by Property 1, we have either $C_u \subset C_i$ or $C_u \cap C_i = \emptyset$. Since C_i is a leaf node, there is no u such that $C_u \subset C_i$. So we have $C_u \cap C_i = \emptyset \Rightarrow B_u \cap C_i = \emptyset$. By Theorem 1, we have $A_i(C_i, C_i) = A(C_i, C_i)$. □

References

[1] S. Datta, Nanoscale device modeling: the Green's function method, Superlattices and Microstructures 28 (4) (2000) 253–278.
[2] R. Lake, G. Klimeck, R.C. Bowen, D. Jovanovic, Single and multiband modeling of quantum electron transport through layered semiconductor devices, Journal of Applied Physics 81 (1997) 7845.
[3] A. Svizhenko, M.P. Anantram, T.R. Govindan, B. Biegel, Two-dimensional quantum mechanical modeling of nanotransistors, Journal of Applied Physics 91 (4) (2002) 2343–2354.
[4] J. Varah, The calculation of the eigenvectors of a general complex matrix by inverse iteration, Mathmatics of Compution 22 (1968) 785–791.
[5] G. Peters, J. Wilkinson, Linear Algebra, Handbook for Automatic Computation, vol. II, Springer-Verlag, 1971. Chapter: The calculation of specified eigenvectors by inverse iteration, pp. 418–791..
[6] J. Wilkinson, Inverse iteration in theory and practice, Symposia Matematica, vol. X, Institutio Nationale di Alta Matematica Monograf, Bologna, Italy, 1972, pp. 361–379.
[7] G. Peters, J. Wilkinson, Inverse iteration, ill-conditioned equations and Newton's method, SIAM Review 21 (1979) 339–360.
[8] A. George, Nested dissection of a regular finite-element mesh, SIAM Journal on Numerical Analysis 10 (2) (1973) 345–363.
[9] K. Takahashi, J. Fagan, M.-S. Chin, Formation of a sparse bus impedance matrix and its application to short circuit study, in: 8th PICA Conference Proceedings, Minneapolis, Minn., 1973, pp. 63–69.
[10] A.M. Erisman, W.F. Tinney, On computing certain elements of the inverse of a sparse matrix, Numerical Mathematics 18 (3) (1975) 177–179.
[11] K. Bowden, A direct solution to the block tridiagonal matrix inversion problem, International Journal of General Systems 15 (3) (1989) 185–198.
[12] J. Schröder, U. Trottenberg, Reduction method for difference equations for boundary value problems. I, Numerische Mathematik 22 (1) (1973) 37–68.
[13] J. Schröder, U. Trottenberg, K. Witsch, On fast Poisson solvers and applications, in: R. Bulirsch, R. Grigorieff, J. Schröder (Eds.), a Conference on Numerical Treatment of Differential Equations, 4–10 July 1976, Oberwolfach, West Germany, 1978, pp. 153–87.
[14] B.L. Buzbee, G.H. Golub, C.W. Nielson, On direct methods for solving Poisson's equations, SIAM Journal on Numerical Analysis 7 (4) (1970) 627–656.
[15] H.S. Wong, Beyond the conventional transistor, IBM Journal of Research and Development 46 (2002) 133–168.
[16] J. Welser, J.L. Hoyt, J.F. Gibbons, NMOS and PMOS transistors fabricated in strained silicon/relaxed silicon-germanium structures, IEDM Technical Digest (1992) 1000.
[17] D. Vasileska, S.S. Ahmed, Narrow-width SOI devices: the role of quantum mechanical size quantization effect and the unintentional doping on the device operation, IEEE Transactions on Electron Devices 52 (2005) 227.
[18] G. Shahidi, SOI technology for the GHz era, IBM Journal of Research and Development 46 (2002) 121–131.
[19] T.J. Walls, V.A. Sverdlov, K.K. Likharev, Nanoscale SOI MOSFETs: a comparison of two options, Solid-State Electronics 48 (2004) 857–865.
[20] S. Hasan, J. Wang, M. Lundstrom, Device design and manufacturing issues for 10 nm-scale MOSFETs: a computational study, Solid-State Electronics 48 (2004) 867–875.
[21] S.I.A. (SIA), International Technology Roadmap for Semiconductors 2001 Edition, Semiconductor Industry Association (SIA), 2706 Montopolis Drive, Austin, Texas 78741 %3chttp://public.itrs.net/Files/2001ITRS/Home.htm%3e, 2001, Chapter: Table C, p. 32.
[22] A. Svizhenko, M.P. Anantram, T. Govindan, B. Biegel, R. Venugopal, Two-dimensional quantum mechanical modeling of nanotransistors, Journal of Applied Physics 91 (4) (2002) 2343–2354.
[23] M.P. Anantram, S.S. Ahmed, A. Svizhenko, D. Kearney, G. Klimeck, NanoFET, doi:10.254/nanohub-r1090.5, 2007.