Contents lists available at ScienceDirect

Journal of Computational Physics

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

A hybrid method for the parallel computation of Green's functions

Dan Erik Petersen [a], Song Li [c], Kurt Stokbro [a], Hans Henrik B. Sørensen [d], Per Christian Hansen [e], Stig Skelboe [a], Eric Darve [b,c,*]

[a] *Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK–2100 Copenhagen, Denmark*
[b] *Department of Mechanical Engineering, Stanford University, 496 Lomita Mall, Durand Building, Room 209, Stanford, CA 94305-4040, USA*
[c] *Institute for Computational and Mathematical Engineering, Stanford University, 496 Lomita Mall, Durand Building, Stanford, CA 94305-4042, USA*
[d] *Department of Computer Science, University of Aarhus, IT-Parken, Aabogade 34, DK-8200 Aarhus N, Denmark*
[e] *Informatics and Mathematical Modelling, Technical University of Denmark, Richard Petersens Plads, Bldg. 321, DK-2800 Lyngby, Denmark*

A R T I C L E I N F O

Article history:
Received 30 September 2008
Received in revised form 19 March 2009
Accepted 23 March 2009
Available online 5 April 2009

PACS:
71.15.−m
02.70.−c

Keywords:
Quantum transport
Density functional theory
Green's function
Cyclic reduction
Inverse matrix
Sparse matrix
Nested dissection
Parallel computing

A B S T R A C T

Quantum transport models for nanodevices using the non-equilibrium Green's function method require the repeated calculation of the block tridiagonal part of the Green's and lesser Green's function matrices. This problem is related to the calculation of the inverse of a sparse matrix. Because of the large number of times this calculation needs to be performed, this is computationally very expensive even on supercomputers. The classical approach is based on recurrence formulas which cannot be efficiently parallelized. This practically prevents the solution of large problems with hundreds of thousands of atoms. We propose new recurrences for a general class of sparse matrices to calculate Green's and lesser Green's function matrices which extend formulas derived by Takahashi and others. We show that these recurrences may lead to a dramatically reduced computational cost because they only require computing a small number of entries of the inverse matrix. Then, we propose a parallelization strategy for block tridiagonal matrices which involves a combination of Schur complement calculations and cyclic reduction. It achieves good scalability even on problems of modest size.

1. Introduction

Commercial electronic devices are rapidly approaching the scale where quantum mechanical effects are affecting the device characteristics [1]. For example, nano-transistors in the 10–30 nm range are strongly affected by quantum mechanical effects such as tunneling and leakage. To allow for continued scaling of silicon technology, new materials are introduced [2] and new nanomaterials such as nanowires [3] or carbon nanotubes [4] are currently being considered as wires or as active parts in future generations of transistors. Also, more exotic devices using DNA [5] or other small molecules [6] are currently under investigation for future use in electronic devices.

To model such devices, it is necessary to apply a quantum mechanical framework [2]. In a system where current flows from a large reservoir (source) to a large drain, the non-equilibrium Green's function approach is applicable [7–9]. This is

* Corresponding author. Fax: +1 650 725 1587.
E-mail address: darve@stanford.edu (E. Darve).

a very general framework which can be applied to all the systems mentioned above. This method involves the coupled solution of the Schrödinger equation for the quantum wave function and the Poisson equation for the electrostatic potential. The Schrödinger equation can be solved in multiple ways, including using a Kohn–Sham Hamiltonian. The main computational difficulty of this problem is the self-consistent solution of the Kohn–Sham Hamiltonian and the Poisson equation, which usually requires an iterative scheme with many iterations. The solution of this system requires the calculation of a Green's function which, once discretized, is often written in the matrix form:

$$\mathbf{G} \overset{\text{def}}{=} \mathbf{A}^{-1}, \quad \text{where } \mathbf{A} \overset{\text{def}}{=} E\mathbf{S}^{\text{o}} - \mathbf{H} - \mathbf{\Sigma}^{\text{L}} - \mathbf{\Sigma}^{\text{R}} \tag{1}$$

where E is an energy point (a scalar), $\mathbf{S}^{\text{o}}$ is an overlap matrix, $\mathbf{H}$ is the Hamiltonian of the system, and $\mathbf{\Sigma}^{\text{L}}$ and $\mathbf{\Sigma}^{\text{R}}$ are the self-energies. The Green's function matrix is computed at many energy points and this is computationally quite expensive even for small matrices. New nanodevices usually require a description at the atomic level; however, their size is large enough that we may have thousands or even millions of atoms in the active part of the device and thus the dimension of $\mathbf{A}$ becomes very large. In such cases, direct inversion of $\mathbf{A}$ becomes impractical, and the application of the non-equilibrium Green's function framework requires new efficient parallel algorithms that exploit the sparsity of $\mathbf{A}$. The focus of this paper is the derivation and benchmarking of a new parallel algorithm that efficiently calculates select entries in $\mathbf{G}$ and the lesser Green's function matrix

$$\mathbf{G}^{<} \overset{\text{def}}{=} \mathbf{G}\mathbf{\Gamma}\mathbf{G}^{\dagger} \tag{2}$$

where $\mathbf{\Gamma}$ is at this point an arbitrary matrix. Assumptions regarding the non-zero entries of $\mathbf{\Gamma}$ will be formulated later on. The lesser Green's function matrix is needed to account for non-equilibrium and scattering effects. The notations in this paper are indicated in Table 1.

In most cases not all entries in $\mathbf{G}$ and $\mathbf{G}^{<}$ are needed. Usually only the diagonal entries of the matrix are required in the iterative process. This leads to a significant reduction in computational cost, which can be realized using a number of different methods [10]. Most of them are related to a technique developed by Takahashi et al. [11–13]. In this approach, a block LDU factorization of $\mathbf{A}$ is computed. Simple algebra shows that:

$$\mathbf{G} = \mathbf{D}^{-1}\mathbf{L}^{-1} + (\mathbf{I} - \mathbf{U})\mathbf{G} \tag{3}$$

$$= \mathbf{U}^{-1}\mathbf{D}^{-1} + \mathbf{G}(\mathbf{I} - \mathbf{L}) \tag{4}$$

where $\mathbf{L}$, $\mathbf{U}$, and $\mathbf{D}$ correspond, respectively, to the lower block unit triangular, upper block unit triangular, and diagonal block LDU factorization of $\mathbf{A}$. The dense inverse $\mathbf{G}$ is treated conceptually as having a block structure based on that of $\mathbf{A}$.

For example, consider the first equation and $j > i$. The block entry $\mathbf{g}_{ij}$ resides above the block diagonal of the matrix and therefore $[\mathbf{D}^{-1}\mathbf{L}^{-1}]_{ij} = 0$. The first equation can then be written as:

$$\mathbf{g}_{ij} = -\sum_{k>i} \mathbf{u}_{ik}\mathbf{g}_{kj} \tag{5}$$

This allows computing the entry $\mathbf{g}_{ij}$ if the entries $\mathbf{g}_{kj}(k > i)$ below it are known. Two similar equations can be derived for the case $j < i$ and $j = i$:

$$\mathbf{g}_{ij} = -\sum_{k>i} \mathbf{g}_{ik}\boldsymbol{\ell}_{kj} \tag{6}$$

$$\mathbf{g}_{ii} = \mathbf{d}_{ii}^{-1} - \sum_{k>i} \mathbf{u}_{ik}\mathbf{g}_{ki} \tag{7}$$

This approach leads to fast backward recurrences, as shown in [12].

Table 1
Notations used in this paper.

$\mathbf{A}$	matrix
$\mathbf{a}_{ij}$	block entry (i,j) of matrix $\mathbf{A}$; in general, a matrix and its block entries are denoted by the same letter. A block is denoted with a bold lowercase letter while the matrix is denoted by a bold uppercase letter
N	number of rows and columns in $\mathbf{A}$
n	number of blocks in matrix
d	block size
$\mathbf{A}^T$	transpose matrix
$\mathbf{A}^{\dagger}$	transpose conjugate matrix
$\mathbf{A}^{-\dagger}$	transpose conjugate of the inverse matrix
$\mathbf{L}, \mathbf{D}, \mathbf{U}$	block LDU factorization of $\mathbf{A}$
$\mathbf{A}(i_1 : i_2, j)$	block column vector containing rows i_1 through i_2 in column j of $\mathbf{A}$
$\mathbf{A}(i, j_1 : j_2)$	block row vector containing columns j_1 through j_2 in row i of $\mathbf{A}$
$\mathbf{A}(i_1 : i_2, j_1 : j_2)$	sub-matrix containing rows i_1 through i_2 and columns j_1 through j_2 of $\mathbf{A}$
$\mathbf{I}$	identity matrix
$\mathbf{0}$	all-zero matrix

The generalization of Takahashi's method to computing $\mathbf{G}^<$ is not straightforward. In particular Eqs. (3) and (4) do not extend to this case. In this paper, we will provide a new way of deriving Eqs. (5)–(7). This derivation will then be extended to $\mathbf{G}^<$ for which similar relations will be derived. These recurrences are most efficient when the sparsity graph of $\mathbf{\Gamma}$ (Eq. (2)) is a subset of the graph of $\mathbf{A}$, i.e., $\gamma_{ij} \neq 0 \Rightarrow \mathbf{a}_{ij} \neq 0$.

For the purpose of computer implementation, an important distinction must be made to distinguish what we term 1D, 2D and 3D problems. In principle, all problems are 3 dimensional. However, if the mesh or device is elongated in one direction, say z, then the mesh can be split into "slices" along the z direction. This gives rise to a matrix $\mathbf{A}$ with block tridiagonal structure. The problem is then termed 1 dimensional. Similarly, if the mesh is elongated in two directions, the matrix $\mathbf{A}$ assumes a block penta-diagonal form and the problem is called 2 dimensional.

This paper is organized as follows. We first describe general relations to compute Green's functions (Sections 2 and 3). These are applicable to meshes with arbitrary connectivity. In Section 4, we calculate the computational cost of these approaches for 1D and 2D Cartesian meshes and for discretizations involving nearest neighbor nodes only, e.g., a 3 point stencil in 1D or a 5 point stencil in 2D. We also compare the computational cost of this approach with a recently published method by Li and Darve (the FIND algorithm [14]).

In Section 5, a parallel implementation of the recurrences for 1 dimensional problems is proposed. The original algorithms by Takahashi [11] and Svizhenko [10] are not parallel since they are based on intrinsically sequential recurrences. However, we show that an appropriate reordering of the nodes and definition of the blocks in $\mathbf{A}$ lead to a large amount of parallelism. In practical cases for nano-transistor simulations, we found that the communication time was small and that the scalability was very good, even for small benchmark cases. This scheme is based on a combination of domain decomposition and cyclic reduction techniques[1] (see, for example, Varga and Hageman [16]). Section 6 has some numerical results.

2. Recurrence formulas for the inverse matrix

Consider a general matrix $\mathbf{A}$ written in block form:

$$\mathbf{A} \overset{\text{def}}{=} \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \tag{8}$$

From the LDU factorization of $\mathbf{A}$, we can form the factors $\mathbf{L}^S \overset{\text{def}}{=} \mathbf{A}_{21}\mathbf{A}_{11}^{-1}, \mathbf{U}^S \overset{\text{def}}{=} \mathbf{A}_{11}^{-1}\mathbf{A}_{12}$, and the Schur complement $\mathbf{S} \overset{\text{def}}{=} \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$. The following equation holds for the inverse matrix $\mathbf{G} = \mathbf{A}^{-1}$:

$$\mathbf{G} = \begin{bmatrix} \mathbf{A}_{11}^{-1} + \mathbf{U}^S\mathbf{S}^{-1}\mathbf{L}^S & -\mathbf{U}^S\mathbf{S}^{-1} \\ -\mathbf{S}^{-1}\mathbf{L}^S & \mathbf{S}^{-1} \end{bmatrix} \tag{9}$$

This equation can be verified by direct multiplication with $\mathbf{A}$. It allows computing the inverse matrix using backward recurrence formulas. These formulas can be obtained by considering step i of the LDU factorization of $\mathbf{A}$, which has the following form:

$$\mathbf{A} = \begin{bmatrix} \mathbf{L}(1:i-1,1:i-1) & \mathbf{0} \\ \mathbf{L}(i:n,1:i-1) & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{D}(1:i-1,1:i-1) & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^i \end{bmatrix} \begin{bmatrix} \mathbf{U}(1:i-1,1:i-1) & \mathbf{U}(1:i-1,i:n) \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \tag{10}$$

The Schur complement matrices $\mathbf{S}^i$ are defined by this equation for all i. From Eq. (10), the first step of the LDU factorization of $\mathbf{S}^i$ is the same as the $i+1$th step in the factorization of $\mathbf{A}$:

$$\mathbf{S}^i = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{L}(i+1:n,i) & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{d}_{ii} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^{i+1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{U}(i,i+1:n) \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \tag{11}$$

Combining Eqs. (9) and (11) with the substitutions:

$$\begin{aligned} &\mathbf{A}_{11} \to \mathbf{d}_{ii} \quad \mathbf{U}^S \to \mathbf{U}(i,i+1:n) \\ &\mathbf{S} \to \mathbf{S}^{i+1} \quad \mathbf{L}^S \to \mathbf{L}(i+1:n,i) \end{aligned}$$

we arrive at:

$$[\mathbf{S}^i]^{-1} = \begin{bmatrix} \mathbf{d}_{ii}^{-1} + \mathbf{U}(i,i+1:n)[\mathbf{S}^{i+1}]^{-1}\mathbf{L}(i+1:n,i) & -\mathbf{U}(i,i+1:n)[\mathbf{S}^{i+1}]^{-1} \\ -[\mathbf{S}^{i+1}]^{-1}\mathbf{L}(i+1:n,i) & [\mathbf{S}^{i+1}]^{-1} \end{bmatrix}$$

From Eq. (9), we have:

$$[\mathbf{S}^i]^{-1} = \mathbf{G}(i:n,i:n), \text{ and } [\mathbf{S}^{i+1}]^{-1} = \mathbf{G}(i+1:n,i+1:n)$$

[1] The method of cyclic reduction was first proposed by Schröder in 1954 and published in German [15].

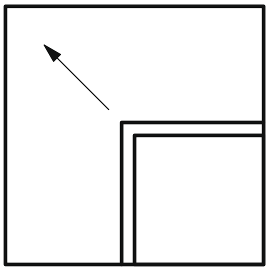


Fig. 1. Schematics of how the recurrence formulas are used to calculate **G**.

We have therefore proved the following backward recurrence relations:

$$\boxed{\begin{aligned}&\mathbf{G}(i+1:n,i) = -\mathbf{G}(i+1:n,i+1:n)\mathbf{L}(i+1:n,i)\\&\mathbf{G}(i,i+1:n) = -\mathbf{U}(i,i+1:n)\mathbf{G}(i+1:n,i+1:n)\\&\mathbf{g}_{ii} = \mathbf{d}_{ii}^{-1} + \mathbf{U}(i,i+1:n)\mathbf{G}(i+1:n,i+1:n)\mathbf{L}(i+1:n,i)\end{aligned}} \tag{12}$$

starting from $\mathbf{g}_{nn} = \mathbf{d}_{nn}^{-1}$. These equations are identical to Eqs. (5)–(7), respectively. A recurrence for $i = n-1$ down to $i = 1$ can therefore be used to calculate **G**. See Fig. 1.

We have assumed that the matrices produced in the algorithms are all invertible (when required), and this may not be true in general. However, this has been the case in all practical applications encountered by the authors so far, for problems originating in electronic structure theory.

By themselves, these recurrence formulas do not lead to any reduction in the computational cost. However, we now show that even though the matrix **G** is full, we do not need to calculate all the entries. We denote $\mathbf{L}^{\mathrm{sym}}$ and $\mathbf{U}^{\mathrm{sym}}$ the lower and upper triangular factors in the symbolic factorization of **A**. The non-zero structure of $(\mathbf{L}^{\mathrm{sym}})^T$ and $(\mathbf{U}^{\mathrm{sym}})^T$ is denoted by Q, that is Q is the set of all pairs (i,j) such that $\ell_{ji}^{\mathrm{sym}} \neq \mathbf{0}$ or $\mathbf{u}_{ji}^{\mathrm{sym}} \neq \mathbf{0}$. Then:

$$\mathbf{g}_{ij} = \begin{cases} -\sum\limits_{l>j,\,(i,l)\in Q} \mathbf{g}_{il}\boldsymbol{\ell}_{lj} & \text{if } i>j \\ -\sum\limits_{k>i,\,(k,j)\in Q} \mathbf{u}_{ik}\mathbf{g}_{kj} & \text{if } i<j \\ \mathbf{d}_{ii}^{-1} + \sum\limits_{k>i,\,l>i,\,(k,l)\in Q} \mathbf{u}_{ik}\mathbf{g}_{kl}\boldsymbol{\ell}_{li} & \text{if } i=j \end{cases} \tag{13}$$

where $(i,j) \in Q$.

Proof. Assume that $(i,j) \in Q, i>j$, then $\mathbf{u}_{ji}^{\mathrm{sym}} \neq 0$. Assume also that $\ell_{lj}^{\mathrm{sym}} \neq 0, l>j$, then by properties of the Gaussian elimination $(i,l) \in Q$. This proves the first case. The case $i<j$ can be proved similarly. For the case $i=j$, assume that $\mathbf{u}_{ik}^{\mathrm{sym}} \neq 0$ and $\ell_{li}^{\mathrm{sym}} \neq 0$, then by properties of the Gaussian elimination $(k,l) \in Q$. □

This implies that we do not need to calculate all the entries in **G** but rather only the entries in **G** corresponding to indices in Q. This represents a significant reduction in computational cost. The cost of computing the entries in **G** in the set Q is the same (up to a constant factor) as the LDU factorization.

Similar results can be derived for the $\mathbf{G}^<$ matrix:

$$\mathbf{G}^< = \mathbf{G}\boldsymbol{\Gamma}\mathbf{G}^\dagger \tag{14}$$

if we assume that $\boldsymbol{\Gamma}$ has the same sparsity pattern as **A**, that is: $\gamma_{ij} \neq 0 \Rightarrow \mathbf{a}_{ij} \neq 0$. The block γ_{ij} denotes the block (i,j) of matrix $\boldsymbol{\Gamma}$.

To calculate the recurrences, we use the LDU factorization of **A** introduced previously. The matrix $\mathbf{G}^<$ satisfies:

$$\mathbf{A}\mathbf{G}^<\mathbf{A}^\dagger = \boldsymbol{\Gamma}$$

Using the same block notations as on page 5, we multiply this equation to the left by $(\mathbf{LD})^{-1}$ and to the right by $(\mathbf{LD})^{-\dagger}$:

$$\begin{bmatrix}\mathbf{I} & \mathbf{U}^S\\ \mathbf{0} & \mathbf{S}\end{bmatrix}\mathbf{G}^<\begin{bmatrix}\mathbf{I} & \mathbf{0}\\ (\mathbf{U}^S)^\dagger & \mathbf{S}^\dagger\end{bmatrix} = \begin{bmatrix}\mathbf{A}_{11} & \mathbf{0}\\ \mathbf{L}^S\mathbf{A}_{11} & \mathbf{I}\end{bmatrix}^{-1}\boldsymbol{\Gamma}\begin{bmatrix}\mathbf{A}_{11} & \mathbf{0}\\ \mathbf{L}^S\mathbf{A}_{11} & \mathbf{I}\end{bmatrix}^{-\dagger}$$

The equation above can be expanded as:

$$\begin{bmatrix}\mathbf{G}_{11}^< + \mathbf{U}^S\mathbf{G}_{21}^< + \mathbf{G}_{12}^<(\mathbf{U}^S)^\dagger + \mathbf{U}^S\mathbf{G}_{22}^<(\mathbf{U}^S)^\dagger & (\mathbf{G}_{12}^< + \mathbf{U}^S\mathbf{G}_{22}^<)\mathbf{S}^\dagger\\ \mathbf{S}(\mathbf{G}_{21}^< + \mathbf{G}_{22}^<(\mathbf{U}^S)^\dagger) & \mathbf{S}\mathbf{G}_{22}^<\mathbf{S}^\dagger\end{bmatrix} = \begin{bmatrix}\mathbf{A}_{11}^{-1}\boldsymbol{\Gamma}_{11}\mathbf{A}_{11}^{-\dagger} & \mathbf{A}_{11}^{-1}(\boldsymbol{\Gamma}_{12} - \boldsymbol{\Gamma}_{11}(\mathbf{L}^S)^\dagger)\\ (\boldsymbol{\Gamma}_{21} - \mathbf{L}^S\boldsymbol{\Gamma}_{11})\mathbf{A}_{11}^{-\dagger} & \boldsymbol{\Gamma}_{22} - \mathbf{L}^S\boldsymbol{\Gamma}_{12} - \boldsymbol{\Gamma}_{21}(\mathbf{L}^S)^\dagger + \mathbf{L}^S\boldsymbol{\Gamma}_{11}(\mathbf{L}^S)^\dagger\end{bmatrix} \tag{15}$$

Using steps similar to the proof of Eqs. (12) and (15) leads to a forward recurrence that is performed in combination with the LDU factorization.

Let us define $\mathbf{\Gamma}^1 \overset{\text{def}}{=} \mathbf{\Gamma}$, then, for $1 \leqslant i \leqslant n-1$:

$$\mathbf{\Gamma}_{\mathrm{L}}^{i+1} \overset{\text{def}}{=} \left(\mathbf{\Gamma}_{21}^i - \mathbf{L}(i+1:n,i)\,\gamma_{11}^i\right)\mathbf{d}_{ii}^{-\dagger} \tag{16}$$

$$\mathbf{\Gamma}_{\mathrm{U}}^{i+1} \overset{\text{def}}{=} \mathbf{d}_{ii}^{-1}\left(\mathbf{\Gamma}_{12}^i - \gamma_{11}^i\,\mathbf{L}(i+1:n,i)^\dagger\right) \tag{17}$$

$$\mathbf{\Gamma}^{i+1} \overset{\text{def}}{=} \mathbf{\Gamma}_{22}^i - \mathbf{L}(i+1:n,i)\,\mathbf{\Gamma}_{12}^i - \mathbf{\Gamma}_{21}^i\,\mathbf{L}(i+1:n,i)^\dagger + \mathbf{L}(i+1:n,i)\,\gamma_{11}^i\,\mathbf{L}(i+1:n,i)^\dagger \tag{18}$$

with

$$\begin{bmatrix} \gamma_{11}^i & \mathbf{\Gamma}_{12}^i \\ \mathbf{\Gamma}_{21}^i & \mathbf{\Gamma}_{22}^i \end{bmatrix} \overset{\text{def}}{=} \mathbf{\Gamma}^i$$

and the following block sizes (in terms of number of blocks of the sub-matrix):

Sub-matrix	Size (blocks)
γ_{11}^i	1×1
$\mathbf{\Gamma}_{12}^i$	$1 \times (n-i+1)$
$\mathbf{\Gamma}_{21}^i$	$(n-i+1) \times 1$
$\mathbf{\Gamma}_{22}^i$	$(n-i+1) \times (n-i+1)$

Note that the **U** factors are not needed in this process. Once this recurrence is complete, we can perform a backward recurrence to find $\mathbf{G}^<$, as in Eq. (12). This recurrence can be proved using Eq. (15):

$$\boxed{\begin{aligned}
&\mathbf{G}^<(i+1:n,i) = \mathbf{G}(i+1:n,i+1:n)\Gamma_{\mathrm{L}}^{i+1} - \mathbf{G}^<(i+1:n,i+1:n)\mathbf{U}(i,i+1:n)^\dagger \\
&\mathbf{G}^<(i,i+1:n) = \Gamma_{\mathrm{U}}^{i+1}\mathbf{G}(i+1:n,i+1:n)^\dagger - \mathbf{U}(i,i+1:n)\mathbf{G}^<(i+1:n,i+1:n) \\
&\mathbf{g}_{ii}^< = \mathrm{d}_{ii}^{-1}\gamma_{11}^i\mathrm{d}_{ii}^{-\dagger} - \mathbf{U}(i,i+1:n)\mathbf{G}^<(i+1:n,i) - \mathbf{G}^<(i,i+1:n)\mathbf{U}(i,i+1:n)^\dagger \\
&\qquad -\mathbf{U}(i,i+1:n)\mathbf{G}^<(i+1:n,i+1:n)\mathbf{U}(i,i+1:n)^\dagger
\end{aligned}} \tag{19}$$

starting from $\mathbf{g}_{nn}^< = \mathbf{g}_{nn}\,\mathbf{\Gamma}^n\,\mathbf{g}_{nn}^\dagger$. The proof is omitted but is similar to the one for Eq. (12).

As before, the computational cost can be reduced significantly by taking advantage of the fact that the calculation can be restricted to indices (i,j) in Q. For this, we need to further assume that $(i,j) \in Q \iff (j,i) \in Q$.

First, we observe that the cost of computing $\mathbf{\Gamma}^i, \mathbf{\Gamma}_{\mathrm{L}}^i, \mathbf{\Gamma}_{\mathrm{U}}^i$ for all i is of the same order as the LDU factorization (i.e., equal up to a constant multiplicative factor). This can be proved by observing that calculating $\mathbf{\Gamma}^{i+1}$ using Eq. (18) leads to the same fill-in as the Schur complement calculations for $\mathbf{S}^{i+1}$.

Second, the set of Eq. (19) for $\mathbf{G}^<$ can be simplified to:

$$\mathbf{g}_{ij}^< = \begin{cases}
\displaystyle\sum_{l>j,\,(i,l)\in Q} \mathbf{g}_{il}\,[\gamma_{\mathrm{L}}^{j+1}]_{l-j,1} - \sum_{l>j,\,(i,l)\in Q} \mathbf{g}_{il}^<\,\mathbf{u}_{jl}^\dagger & \text{if } i>j \\
\displaystyle\sum_{k>i,\,(j,k)\in Q} [\gamma_{\mathrm{U}}^{i+1}]_{1,k-i}\,\mathbf{g}_{jk}^\dagger - \sum_{k>i,\,(k,j)\in Q} \mathbf{u}_{ik}\,\mathbf{g}_{kj}^< & \text{if } i<j \\
\displaystyle\mathbf{d}_{ii}^{-1}\gamma_{11}^i\mathbf{d}_{ii}^{-\dagger} - \sum_{k>i,\,(k,i)\in Q} \mathbf{u}_{ik}\,\mathbf{g}_{ki}^< - \sum_{k>i,\,(i,k)\in Q} \mathbf{g}_{ik}^<\,\mathbf{u}_{ik}^\dagger & \\
\displaystyle\quad - \sum_{k>i,\,l>i,\,(k,l)\in Q} \mathbf{u}_{ik}\,\mathbf{g}_{kl}^<\,\mathbf{u}_{il}^\dagger & \text{if } i=j
\end{cases} \tag{20}$$

where $(i,j) \in Q$. (Notation clarification: γ_{jl}^i, $[\gamma_{\mathrm{L}}^i]_{jl}$, and $[\gamma_{\mathrm{U}}^i]_{jl}$ are respectively blocks of $\mathbf{\Gamma}^i, \mathbf{\Gamma}_{\mathrm{L}}^i$, and $\mathbf{\Gamma}_{\mathrm{U}}^i$.) The proof of this result is similar to the one for Eq. (13) for **G**. The reduction in computational cost is realized because all the sums are restricted to indices in Q.

Observe that the calculation of **G** and $\mathbf{\Gamma}^i$ depends only on the LDU factorization, while the calculation of $\mathbf{G}^<$ depends also on $\mathbf{\Gamma}^i$ and **G**.

3. Sequential algorithm for 1D problems

In this section, we present a sequential implementation of the relations presented above, for 1D problems. Section 5 will discuss the parallel implementation.

In 1 dimensional problems, matrix **A** assumes a block tridiagonal form. The LDU factorization is obtained using the following recurrences:

$$\boldsymbol{\ell}_{i+1,i} = \mathbf{a}_{i+1,i}\,\mathbf{d}_{ii}^{-1} \tag{21}$$
$$\mathbf{u}_{i,i+1} = \mathbf{d}_{ii}^{-1}\,\mathbf{a}_{i,i+1} \tag{22}$$
$$\mathbf{d}_{i+1,i+1} = \mathbf{a}_{i+1,i+1} - \mathbf{a}_{i+1,i}\,\mathbf{d}_{ii}^{-1}\,\mathbf{a}_{i,i+1} = \mathbf{a}_{i+1,i+1} - \boldsymbol{\ell}_{i+1,i}\,\mathbf{a}_{i,i+1} = \mathbf{a}_{i+1,i+1} - \mathbf{a}_{i+1,i}\,\mathbf{u}_{i,i+1} \tag{23}$$

From Eq. (13), the backward recurrences for **G** are given by:

$$\mathbf{g}_{i+1,i} = -\mathbf{g}_{i+1,i+1}\,\boldsymbol{\ell}_{i+1,i} \tag{24}$$
$$\mathbf{g}_{i,i+1} = -\mathbf{u}_{i,i+1}\,\mathbf{g}_{i+1,i+1} \tag{25}$$
$$\mathbf{g}_{ii} = \mathbf{d}_{ii}^{-1} + \mathbf{u}_{i,i+1}\,\mathbf{g}_{i+1,i+1}\,\boldsymbol{\ell}_{i+1,i} = \mathbf{d}_{ii}^{-1} - \mathbf{g}_{i,i+1}\,\boldsymbol{\ell}_{i+1,i} = \mathbf{d}_{ii}^{-1} - \mathbf{u}_{i,i+1}\,\mathbf{g}_{i+1,i} \tag{26}$$

starting with $\mathbf{g}_{nn} = \mathbf{d}_{nn}^{-1}$. To calculate $\mathbf{G}^{<}$, we first need to compute $\boldsymbol{\Gamma}^{i}$. Considering only the non-zero entries in the forward recurrence Eqs. (16)–(18), we need to calculate:

$$\gamma_{\mathrm{L}}^{i+1} \overset{\mathrm{def}}{=} (\gamma_{i+1,i} - \boldsymbol{\ell}_{i+1,i}\,\gamma^{i})\,\mathbf{d}_{ii}^{-\dagger}$$
$$\gamma_{\mathrm{U}}^{i+1} \overset{\mathrm{def}}{=} \mathbf{d}_{ii}^{-1}\,(\gamma_{i,i+1} - \gamma^{i}\,\boldsymbol{\ell}_{i+1,i}^{\dagger})$$
$$\gamma^{i+1} \overset{\mathrm{def}}{=} \gamma_{i+1,i+1} - \boldsymbol{\ell}_{i+1,i}\,\gamma_{i,i+1} - \gamma_{i+1,i}\,\boldsymbol{\ell}_{i+1,i}^{\dagger} + \boldsymbol{\ell}_{i+1,i}\,\gamma^{i}\,\boldsymbol{\ell}_{i+1,i}^{\dagger}$$

starting from $\gamma^{1} \overset{\mathrm{def}}{=} \gamma_{11}$ (the (1,1) block in matrix $\boldsymbol{\Gamma}$); $\gamma^{i}, \gamma_{\mathrm{L}}^{i}$, and γ_{U}^{i} are 1×1 blocks. For simplicity, we have shortened the notations. The blocks $\gamma^{i}, \gamma_{\mathrm{L}}^{i}$, and γ_{U}^{i} are in fact the (1,1) block of $\boldsymbol{\Gamma}^{i}, \boldsymbol{\Gamma}_{\mathrm{L}}^{i}$, and $\boldsymbol{\Gamma}_{\mathrm{U}}^{i}$.

Once the factors $\mathbf{L}, \mathbf{U}, \mathbf{G}, \gamma^{i}, \gamma_{\mathrm{L}}^{i}$, and γ_{U}^{i} have been computed, the backward recurrence Eq. (19) for $\mathbf{G}^{<}$ can be computed:

$$\mathbf{g}_{i+1,i}^{<} = \mathbf{g}_{i+1,i+1}\,\gamma_{\mathrm{L}}^{i+1} - \mathbf{g}_{i+1,i+1}^{<}\,\mathbf{u}_{i,i+1}^{\dagger}$$
$$\mathbf{g}_{i,i+1}^{<} = \gamma_{\mathrm{U}}^{i+1}\,\mathbf{g}_{i+1,i+1}^{\dagger} - \mathbf{u}_{i,i+1}\,\mathbf{g}_{i+1,i+1}^{<}$$
$$\mathbf{g}_{ii}^{<} = \mathbf{d}_{ii}^{-1}\,\gamma^{i}\,\mathbf{d}_{ii}^{-\dagger} - \mathbf{u}_{i,i+1}\,\mathbf{g}_{i+1,i}^{<} - \mathbf{g}_{i,i+1}^{<}\,\mathbf{u}_{i,i+1}^{\dagger} - \mathbf{u}_{i,i+1}\,\mathbf{g}_{i+1,i+1}^{<}\,\mathbf{u}_{i,i+1}^{\dagger}$$

starting from $\mathbf{g}_{nn}^{<} = \mathbf{g}_{nn}\,\gamma^{n}\,\mathbf{g}_{nn}^{\dagger}$.

These recurrences are the most efficient way to calculate **G** and $\mathbf{G}^{<}$ on a sequential computer. On a parallel computer, this approach is of limited use since there is little room for parallelization. In Section 5 we describe a scalable algorithm to perform the same calculations in a truly parallel fashion.

4. Computational cost for 1D and 2D cartesian meshes with nearest neighbor stencils

In this section, we determine the computational cost of the sequential algorithms. From the recurrence relations (13) and (20), one can prove that the computational cost of calculating **G** and $\mathbf{G}^{<}$ has the same scaling with problem size as the LDU factorization. If one uses a nested dissection ordering associated with the mesh [17], we obtain the following costs for Cartesian grids assuming a local discretization stencil:

Dimension	Cost
1D	$O(n\,d^{3})$
2D (square grid)	$O(n^{3/2}\,d^{3})$
3D (cubic grid)	$O(n^{2}\,d^{3})$

We now do a more detailed analysis of the computational cost and a comparison with the FIND algorithm of Li et al. [14]. This algorithm comprises an LDU factorization and a backward recurrence.

For the 1D case, the LDU factorization needs $4d^{3}$ flops for each column: d^{3} for computing $\mathbf{d}_{ii}^{-1}$, d^{3} for $\boldsymbol{\ell}_{i+1,i}$, d^{3} for $\mathbf{u}_{i,i+1}$, and d^{3} for $\mathbf{d}_{i+1,i+1}$. So the total computational cost of LDU factorization is $4nd^{3}$ flops. Following Eq. (13), the cost of the backward recurrence is $3nd^{3}$. Note that the explicit form of $\mathbf{d}_{ii}^{-1}$ is not needed in the LDU factorization. However, the explicit form of $\mathbf{d}_{ii}^{-1}$ is needed for the backward recurrence and computing it in the LDU factorization reduces the total cost. In the above analysis, we have therefore included the cost d^{3} of obtaining $\mathbf{d}_{ii}^{-1}$ during the LDU factorization. The total cost in 1D is therefore:

$$7nd^{3}.$$

For the 2D case, similar to the nested dissection method [17] and FIND algorithm [14], we decompose the mesh in a hierarchical way. See Fig. 2. First we split the mesh in two parts (see the upper and lower parts of the mesh on the right panel of Fig. 2). This is done by identifying a set of nodes called the separator. They are numbered 31 in the right panel. The separator is such that it splits the mesh into 3 sets: set s_1, s_2 and the separator s itself. It satisfies the following properties: (i)

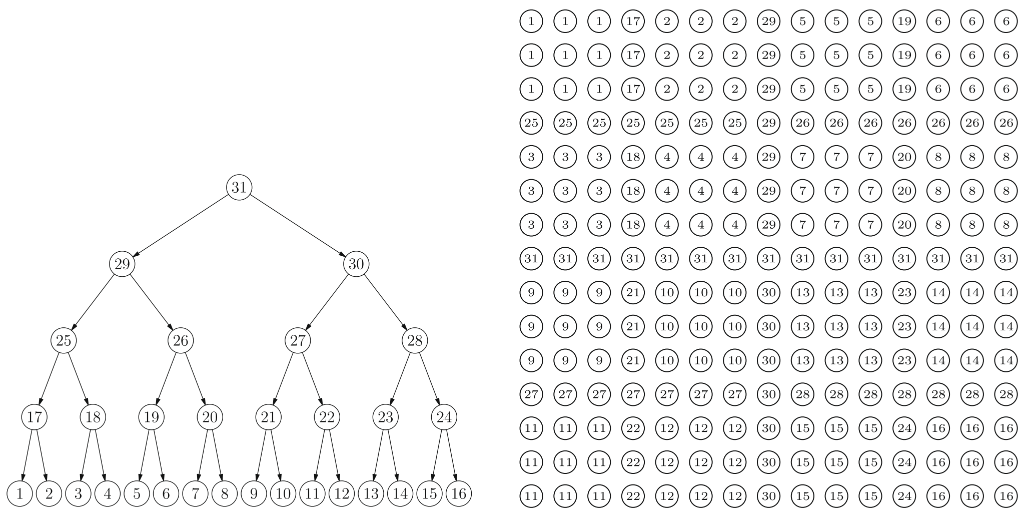


Fig. 2. The cluster tree (left) and the mesh nodes (right) for mesh of size 15×15. The number in each tree node indicates the cluster number. The number in each mesh node indicates the cluster it belongs to.

$\mathbf{a}_{ij} = 0$ if i belongs to s_1 and j to s_2, and vice versa (this is a separator set); (ii) for every i in s, there is at least one index j_1 in s_1 and one index j_2 in s_2 such that $\mathbf{a}_{ij_1} \neq 0$ and $\mathbf{a}_{ij_2} \neq 0$ (the set is minimal). Once the mesh is split into s_1 and s_2 the process is repeated recursively, thereby building a tree decomposition of the mesh. In Fig. 2, clusters 17–31 are all separators [17]. When we perform the LDU factorization, we eliminate the nodes corresponding to the lower level clusters first and then those in the higher level clusters.

Compared to nested dissection [17], we use vertical and horizontal separators instead of cross shaped separators to make the estimates of computational cost easier to derive. Compared to FIND [14], we use single separators | here instead of double separators ‖ since we do not need additional "independence" among clusters as required in [14] for multiple LU factorizations. See [14] for additional details regarding this distinction.

To estimate the computational cost, we need to consider the boundary nodes of a given cluster. Boundary nodes are nodes of a cluster that are connected to nodes outside the cluster. Since the non-zero pattern of the matrix changes as the elimination is proceeding, we need to consider the evolving connectivity of the mesh. For example, the nodes labeled 22 become connected to nodes 27 and 30 after nodes 11 and 12 have been eliminated; similarly, the nodes 20 become connected to nodes 26, 31 and 29. This is shown in Fig. 2. For more details, see [17,14].

Once the separators and boundaries are determined, we see that the computational cost of eliminating a separator is equal to $sb^2 + 2s^2b + \frac{1}{3}s^3 = s(b+s)^2 - \frac{2}{3}s^3$ flops, where s is the number of nodes in the separator set and b is the number of nodes in the boundary set. As in the 1D case, we include the cost for $\mathbf{d}_{ii}^{-1}$.

To make it simple, we focus on a 2D square mesh with N nodes and the typical nearest neighbor connectivity. If we ignore the effect of the boundary of the mesh, the size of the separators within a given level is fixed. If the ratio b/s is constant, then the cost for each separator is proportional to s^3 and the number of separators is proportional to N/s^2, so the cost for each level doubles every two levels. The computational costs thus form a geometric series and the top level cost dominates the total cost.

When we take the effect of the mesh boundary in consideration, the value of b for a cluster near the boundary of the mesh needs to be adjusted. For lower levels, such clusters form only a small fraction of all the clusters and thus the effect is not significant. For top levels, however, such effect cannot be ignored. Since the cost for the top levels dominates the total cost, we need to calculate the computational cost for top levels more precisely.

Table 2 shows the cost for the top level clusters. The last two rows are the upper bound of the total cost for the rest of the small clusters.

If we compute the cost for each level and sum them together, we obtain a cost of $24.9N^{3/2}$ flops for the LDU factorization.

For the backward recurrence, we have the same sets of separators. Each node in the separator is connected to all the other nodes in the separator and all the nodes in the boundary set. Since we have an upper triangular matrix now, when we deal with a separator of size s with b nodes on the boundary, the number of non-zero entries in each row increases from b to $s + b$. As a result, the cost for computing Eq. (13) is $3\left(b^2s + bs^2 + \frac{1}{3}s^3\right)$ flops for each step. The total computational cost for the backward recurrence is then $61.3N^{3/2}$ flops. The costs for each type of clusters are also listed in Table 2.

Adding together the cost of the LDU factorization and the backward recurrence, the total computational cost for the algorithm is $86.2N^{3/2}$ flops. For FIND, the cost is $147N^{3/2}$ flops [18]. Note that these costs are for the serial version of the two algorithms. Although the cost is reduced roughly by half compared to FIND, the parallelization of FIND is different and therefore the running time of both algorithms on parallel platforms may scale differently.

Table 2
Estimate of the computational cost for a 2D square mesh for different cluster sizes. The size is in unit of $N^{1/2}$. The cost is in unit of $N^{3/2}$ flops.

Size of cluster		Cost per cluster		Level	Number of clusters
Separator	Boundary	LDU	Back. Recurr.		
1/2	1	1.042	2.375	1	2
1/2	1	1.042	2.375	2	4
1/4	5/4	0.552	1.422	3	4
1/4	3/4	0.240	0.578	3	4
1/4	1	0.380	0.953	4	4
1/4	3/4	0.240	0.578	4	8
1/4	1/2	0.130	0.297	4	4
1/8	3/4	0.094	0.248	5	8
1/8	5/8	0.069	0.178	5	24
1/8	1/2	0.048	0.119	6	64
1/16	3/8	0.012	0.031	7	128
1/8	1/8	0.048	0.119	8 ...	$\leqslant$64
1/16	3/8	0.012	0.031	9 ...	$\leqslant$128

We now focus on the implementation of these recurrence relations for the calculation of **G** in the 1D case on a parallel computer. Similar ideas can be applied to the calculation of $\mathbf{G}^{<}$. The parallel implementation of the 2D case is significantly more complicated and will be the subject of another paper.

5. Parallel algorithm for 1D problems

We present a parallel algorithm for the calculation of the Green's function matrix **G** typically encountered in electronic structure calculations where the matrix **A** (cf. Eq. (1)) is assumed to be an $n \times n$ block matrix, and in block tridiagonal form as shown by

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} & & & \\ \mathbf{a}_{21} & \mathbf{a}_{22} & \mathbf{a}_{23} & & \\ & \mathbf{a}_{32} & \mathbf{a}_{33} & \mathbf{a}_{34} & \\ & & \ddots & \ddots & \ddots \end{bmatrix}. \tag{27}$$

where each block element $\mathbf{a}_{ij}$ is a dense complex matrix. In order to develop a parallel algorithm, we assume that we have at our disposal a total of $\mathcal{P}$ processing elements (e.g., single core on a modern processor). We also consider that we have **processes** with the convention that each process is associated with a unique processing element, and we assume that they can communicate among themselves. The processes are labeled $p_0, p_1, \ldots, p_{\mathcal{P}-1}$. The block tridiagonal structure **A** is then distributed among these processes in a *row-striped* manner, as illustrated in Fig. 3.

Thus each process is assigned ownership of certain contiguous rows of **A**. This ownership arrangement, however, also extends to the calculated blocks of the inverse **G**, as well as any LU factors determined during calculation in **L** and **U**. The manner of distribution is an issue of load balancing, and will be addressed later in the paper.

Furthermore, for illustrative purposes, we present the block matrix structures as having identical block sizes throughout. The algorithm presented has no condition for the uniformity of block sizes in **A**, and can be applied to a block tridiagonal matrix as presented on the right in Fig. 3. The size of the blocks throughout **A**, however, does have consequences for adequate load balancing.

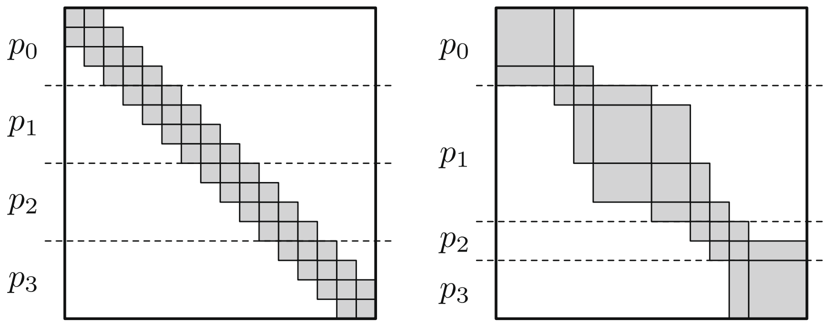
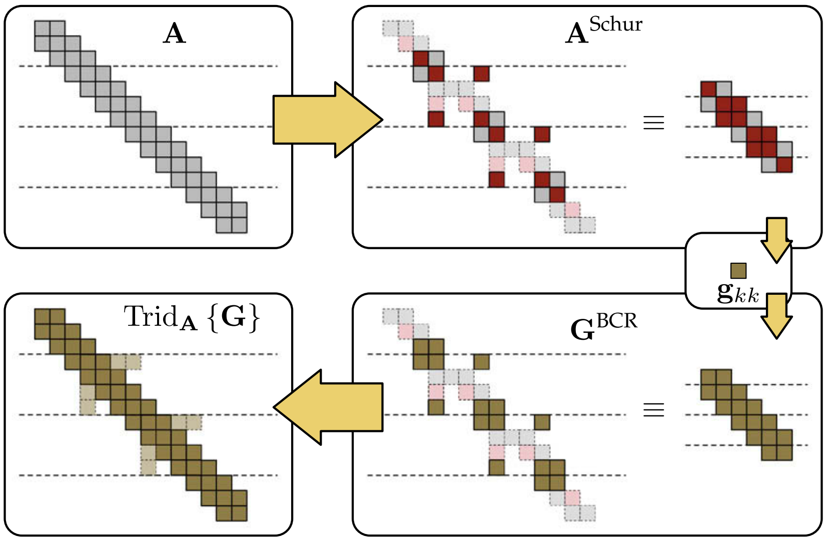


Fig. 3. Two different block tridiagonal matrices distributed among 4 different processes, labeled p_0, p_1, p_2 and p_3.



$\mathbf{A}$
$\mathbf{A}^{\text{Schur}}$
$\equiv$
$\mathbf{g}_{kk}$
$\mathbf{G}^{\text{BCR}}$
$\equiv$
$\text{Trid}_{\mathbf{A}}\{\mathbf{G}\}$

Fig. 4. The distinct phases of operations performed by the hybrid method in determining $\text{Trid}_{\mathbf{A}}\{\mathbf{G}\}$, the block tridiagonal portion of **G** with respect to the structure of **A**. The block matrices in this example are partitioned across 4 processes, as indicated by the horizontal dashed lines.

For the purposes of electronic structure applications, we only require the portion of the inverse **G** with the same block tridiagonal structure of **A**. We express this portion as $\text{Trid}_{\mathbf{A}}\{\mathbf{G}\}$.

The parallel algorithm is a *hybrid* technique in the sense that it combines the techniques of cyclic reduction and Schur block decomposition, but where we now consider individual elements to be dense matrix blocks. The steps taken by the hybrid algorithm to produce $\text{Trid}_{\mathbf{A}}\{\mathbf{G}\}$ is outlined in Fig. 4.

The algorithm begins with our block tridiagonal matrix **A** partitioned across a number of processes, as indicated by the dashed horizontal lines. Each process then performs what is equivalent to calculating a Schur complement on the rows/blocks that it owns, leaving us with a *reduced* system that is equivalent to a smaller block tridiagonal matrix $\mathbf{A}^{\text{Schur}}$. This phase, which we name the Schur reduction phase, is entirely devoid of interprocess communication.

It is on this smaller block tridiagonal structure that we perform block cyclic reduction (BCR), leaving us with a single block of the inverse, $\mathbf{g}_{kk}$. This block cyclic reduction phase involves interprocess communication.

From $\mathbf{g}_{kk}$ we then produce the portion of the inverse corresponding to $\mathbf{G}^{\text{BCR}} = \text{Trid}_{\mathbf{A}^{\text{Schur}}}\{\mathbf{G}\}$ in what we call the block cyclic production phase. This is done using Eq. (12). Finally, using $\mathbf{G}^{\text{BCR}}$, we can then determine the full tridiagonal structure of **G** that we desire without any further need for interprocess communication through a so-called Schur production phase. The block cyclic production phase and Schur production phase are a parallel implementation of the backward recurrences in Eq. (12).

5.1. Schur phases

In order to illustrate where the equations for the Schur reduction and production phases come from, we perform them for small examples in this section. Furthermore, this will also serve to illustrate how our hybrid method is equivalent to unpivoted block Gaussian elimination.

5.1.1. Corner block operations

Looking at the case of Schur corner reduction, we take the following block form as our starting point:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_{ii} & \mathbf{a}_{ij} & \mathbf{0} \\ \mathbf{a}_{ji} & \mathbf{a}_{jj} & \star \\ \mathbf{0} & \star & \star \end{bmatrix},$$

where $\star$ denotes some arbitrary entries (zero or not). In eliminating the block $\mathbf{a}_{ii}$, we calculate the following LU factors

$$\ell_{ji} = \mathbf{a}_{ji}\mathbf{a}_{ii}^{-1}$$
$$\mathbf{u}_{ij} = \mathbf{a}_{ii}^{-1}\mathbf{a}_{ij}$$

and we determine the Schur block

$$\mathbf{s} \overset{\text{def}}{=} \mathbf{a}_{jj} - \ell_{ji}\mathbf{a}_{ij}.$$

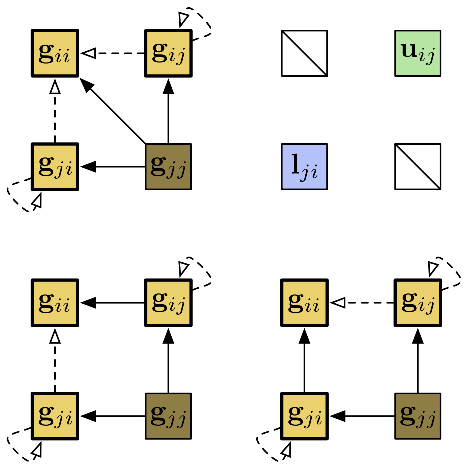


Fig. 5. Three different schemes that represent a corner production step undertaken in the BCR production phase, where we produce inverse blocks on row/column i using inverse blocks and LU factors from row/column j.

Let us now assume that the inverse block $\mathbf{g}_{jj}$ has been calculated. We then start the Schur corner production phase in which, using the LU factors saved from the reduction phase, we can obtain

$$\mathbf{g}_{ji} = -\mathbf{g}_{jj}\boldsymbol{\ell}_{ji}$$
$$\mathbf{g}_{ij} = -\mathbf{u}_{ij}\mathbf{g}_{jj}$$

(see Eq. (12)) and finally

$$\begin{aligned} \mathbf{g}_{ii} &= \mathbf{a}_{ii}^{-1} + \mathbf{u}_{ij}\mathbf{g}_{jj}\boldsymbol{\ell}_{ji} && (28)\\ &= \mathbf{a}_{ii}^{-1} - \mathbf{u}_{ij}\mathbf{g}_{ji} && (29)\\ &= \mathbf{a}_{ii}^{-1} - \mathbf{g}_{ij}\boldsymbol{\ell}_{ji}. && (30) \end{aligned}$$

This production step is visualized in Fig. 5. On the top right of the figure we have the stored LU factors preserved from the BCR elimination phase. On the top left, bottom right and bottom left we see three different schemes for producing the new inverse blocks $\mathbf{g}_{ii}, \mathbf{g}_{ij}$ and $\mathbf{g}_{ji}$ from $\mathbf{g}_{jj}$ and the LU factors $\mathbf{u}_{ij}$ and ℓ_{ji}, corresponding to Eqs. (28)–(30). A solid arrow indicates the need for the corresponding block of the inverse matrix and a dashed arrow indicates the need for the corresponding LU block.

Using this figure, we can determine what matrix blocks are necessary to calculate one of the desired inverse blocks. Looking at a given inverse block, the required information is indicated by the inbound arrows. The only exception is $\mathbf{g}_{ii}$ which also requires the block $\mathbf{a}_{ii}$.

Three different schemes to calculate $\mathbf{g}_{ii}$ are possible, since the block can be determined via any of the three equations Eqs. (28) and (29) or Eq. (30), corresponding respectively to the upper left, lower right and lower left schemes in Fig. 5.

Assuming we have process p_i owning row i and process p_j owning row j, we disregard choosing the lower left scheme (cf. Eq. (30)) since the computation of $\mathbf{g}_{ii}$ on process p_i will have to wait for process p_j to calculate and send $\mathbf{g}_{ji}$. This leaves us with the choice of either the upper left scheme (cf. Eq. (28)) or bottom right scheme (cf. Eq. (29)) where $\mathbf{g}_{jj}$ and ℓ_{ji} can be sent immediately, and both processes p_i and p_j can then proceed to calculate in parallel.

However, the bottom right scheme (cf. Eq. (29)) is preferable to the top left scheme (cf. Eq. (28)) since it saves an extra matrix–matrix multiplication. This motivates our choice of using Eq. (29) for our hybrid method.

5.1.2. Center block operations

The reduction/production operations undertaken by a center process begins with the following block tridiagonal form:

$$\mathbf{A} = \begin{bmatrix} \star & \star & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \star & \mathbf{a}_{ii} & \mathbf{a}_{ij} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{a}_{ji} & \mathbf{a}_{jj} & \mathbf{a}_{jk} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{a}_{kj} & \mathbf{a}_{kk} & \star \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \star & \star \end{bmatrix}.$$

Through a block permutation matrix **P**, we can transform it to the form

$$\mathrm{PAP} = \left(\begin{array}{c|cccc} \mathbf{a}_{jj} & \mathbf{a}_{ji} & \mathbf{a}_{jk} & 0 & 0 \\ \hline \mathbf{a}_{ij} & \mathbf{a}_{ii} & 0 & \star & 0 \\ \mathbf{a}_{kj} & 0 & \mathbf{a}_{kk} & 0 & \star \\ 0 & \star & 0 & \star & 0 \\ 0 & 0 & \star & 0 & \star \end{array}\right),$$

which we interpret as a 2 × 2 block matrix as indicated by the partitioning lines in the expression. We then perform a Schur complement calculation as done earlier for a corner process, obtaining parts of the LU factors

$$\text{Part of the L factor(column } j\text{)}: \quad \begin{pmatrix} \mathbf{a}_{ij} \\ \mathbf{a}_{kj} \end{pmatrix} \mathbf{a}_{jj}^{-1} = \begin{pmatrix} \ell_{ij} \\ \ell_{kj} \end{pmatrix} \tag{31}$$

$$\text{Part of the U factor(row } j\text{)}: \quad \mathbf{a}_{jj}^{-1}(\mathbf{a}_{ji}\ \mathbf{a}_{jk}) = (\mathbf{u}_{ji}\ \mathbf{u}_{jk}). \tag{32}$$

This then leads us to the 2 × 2 block Schur matrix

$$\begin{pmatrix} \mathbf{a}_{ii} & \mathbf{0} \\ \mathbf{0} & \mathbf{a}_{kk} \end{pmatrix} - \begin{pmatrix} \ell_{ij} \\ \ell_{kj} \end{pmatrix} (\mathbf{a}_{ji}\ \mathbf{a}_{jk}) = \begin{pmatrix} \mathbf{a}_{ii} - \ell_{ij}\mathbf{a}_{ji} & -\ell_{ij}\mathbf{a}_{jk} \\ -\ell_{kj}\mathbf{a}_{ji} & \mathbf{a}_{kk} - \ell_{kj}\mathbf{a}_{jk} \end{pmatrix} \tag{33}$$

Let us assume that we have now computed the following blocks of the inverse **G**:

$$\begin{pmatrix} \mathbf{g}_{ii} & \mathbf{g}_{ik} \\ \mathbf{g}_{ki} & \mathbf{g}_{kk} \end{pmatrix}$$

With this information, we can use the stored LU factors to determine the other blocks of the inverse [see Eq. (12)], getting

$$(\mathbf{g}_{ji}\ \mathbf{g}_{jk}) = -(\mathbf{u}_{ji}\ \mathbf{u}_{jk}) \begin{pmatrix} \mathbf{g}_{ii} & \mathbf{g}_{ik} \\ \mathbf{g}_{ki} & \mathbf{g}_{kk} \end{pmatrix} = (-\mathbf{u}_{ji}\mathbf{g}_{ii} - \mathbf{u}_{jk}\mathbf{g}_{ki} \qquad - \mathbf{u}_{ji}\mathbf{g}_{ik} - \mathbf{u}_{jk}\mathbf{g}_{kk})$$

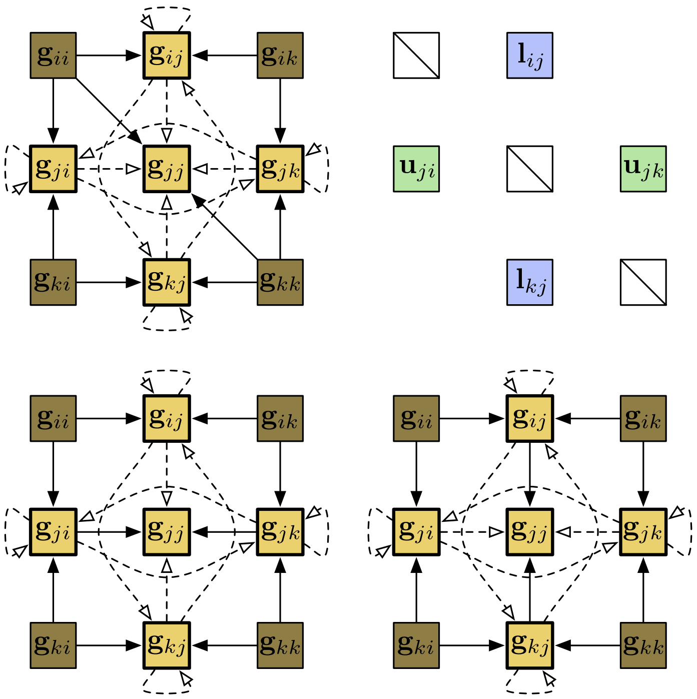


Fig. 6. The three different schemes that represent a center production step undertaken in the BCR production phase, where we produce inverse block elements on row/column j using inverse blocks and LU factors from rows i and k.

and

$$\begin{pmatrix} \mathbf{g}_{ij} \\ \mathbf{g}_{kj} \end{pmatrix} = -\begin{pmatrix} \mathbf{g}_{ii} & \mathbf{g}_{ik} \\ \mathbf{g}_{ki} & \mathbf{g}_{kk} \end{pmatrix} \begin{pmatrix} \boldsymbol{\ell}_{ij} \\ \boldsymbol{\ell}_{kj} \end{pmatrix} = \begin{pmatrix} -\mathbf{g}_{ii}\boldsymbol{\ell}_{ij} - \mathbf{g}_{ik}\boldsymbol{\ell}_{kj} \\ -\mathbf{g}_{ki}\boldsymbol{\ell}_{ij} - \mathbf{g}_{kk}\boldsymbol{\ell}_{kj} \end{pmatrix}.$$

The final block of the inverse is obtained as

$$\mathbf{g}_{jj} = \mathbf{a}_{jj}^{-1} + (\mathbf{u}_{ji}\ \mathbf{u}_{jk}) \begin{pmatrix} \mathbf{g}_{ii} & \mathbf{g}_{ik} \\ \mathbf{g}_{ki} & \mathbf{g}_{kk} \end{pmatrix} \begin{pmatrix} \boldsymbol{\ell}_{ij} \\ \boldsymbol{\ell}_{kj} \end{pmatrix} = \mathbf{a}_{jj}^{-1} - (\mathbf{g}_{ji}\ \mathbf{g}_{jk}) \begin{pmatrix} \boldsymbol{\ell}_{ij} \\ \boldsymbol{\ell}_{kj} \end{pmatrix} = \mathbf{a}_{jj}^{-1} - \mathbf{g}_{ji}\boldsymbol{\ell}_{ij} - \mathbf{g}_{jk}\boldsymbol{\ell}_{kj}. \tag{34}$$

The Schur production step for a center block is visualized in Fig. 6, where the arrows are given the same significance as for a corner production step shown in Fig. 5.

Again, three different schemes arise since $\mathbf{g}_{jj}$ can be determined via one of the following three equations, depending on how $\mathbf{g}_{jj}$ from Eq. (34) is calculated:

$$\mathbf{g}_{jj} = \mathbf{a}_{jj}^{-1} + \mathbf{u}_{ji}\mathbf{g}_{ii}\boldsymbol{\ell}_{ij} + \mathbf{u}_{jk}\mathbf{g}_{ki}\boldsymbol{\ell}_{ij} + \mathbf{u}_{ji}\mathbf{g}_{ik}\boldsymbol{\ell}_{kj} + \mathbf{u}_{jk}\mathbf{g}_{kk}\boldsymbol{\ell}_{kj} \tag{35}$$
$$\mathbf{g}_{jj} = \mathbf{a}_{jj}^{-1} - \mathbf{u}_{ji}\mathbf{g}_{ij} - \mathbf{u}_{jk}\mathbf{g}_{kj} \tag{36}$$
$$\mathbf{g}_{jj} = \mathbf{a}_{jj}^{-1} - \mathbf{g}_{ji}\boldsymbol{\ell}_{ij} - \mathbf{g}_{jk}\boldsymbol{\ell}_{kj} \tag{37}$$

where Eqs. (35)–(37) corresponds to the upper left, lower right and lower left schemes in Fig. 6. Similarly motivated as for the corner Schur production step, we choose the scheme related to Eq. (37) corresponding to the lower left corner of Fig. 6.

5.2. Block cyclic reduction phases

Cyclic reduction operates on a regular tridiagonal linear system by eliminating the odd-numbered unknowns recursively, until only a single unknown remains, uncoupled from the rest of the system. One can then solve for this unknown, and from there descend down the recursion tree and obtain the full solution. In the case of block cyclic reduction, the individual scalar unknowns correspond to block matrices, but the procedure operates in the same manner. For our application, the equations are somewhat different. We have to follow Eq. (12) but otherwise the pattern of computation is similar. The basic operation in the reduction phase of BCR is the row-reduce operation, where two odd-indexed rows are eliminated by row operations towards its neighbor. Starting with the original block tridiagonal form,

$$\begin{pmatrix} \star & \star & \star & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{a}_{ih} & \mathbf{a}_{ii} & \mathbf{a}_{ij} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{a}_{ji} & \mathbf{a}_{jj} & \mathbf{a}_{jk} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{a}_{kj} & \mathbf{a}_{kk} & \mathbf{a}_{kl} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \star & \star & \star \end{pmatrix},$$

we reduce from the odd rows i and k towards the even row j, eliminating the coupling element $\mathbf{a}_{ji}$ by a row operation involving row i and the factor $\boldsymbol{\ell}_{ji} = \mathbf{a}_{ji}\mathbf{a}_{ii}^{-1}$. Likewise, we eliminate $\mathbf{a}_{jk}$ by a row operation involving row k and the factor $\boldsymbol{\ell}_{jk} = \mathbf{a}_{jk}\mathbf{a}_{kk}^{-1}$, and obtain

$$\begin{pmatrix} \star & \star & \star & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{a}_{ih} & \mathbf{a}_{ii} & \mathbf{a}_{ij} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{a}_{jh}^{\mathrm{BCR}} & \mathbf{0} & \mathbf{a}_{jj}^{\mathrm{BCR}} & \mathbf{0} & \mathbf{a}_{jl}^{\mathrm{BCR}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{a}_{kj} & \mathbf{a}_{kk} & \mathbf{a}_{kl} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \star & \star & \star \end{pmatrix}, \tag{38}$$

where the new and updated elements are given as

$$\mathbf{a}_{jj}^{\mathrm{BCR}} \overset{\mathrm{def}}{=} \mathbf{a}_{jj} - \boldsymbol{\ell}_{ji}\mathbf{a}_{ij} - \boldsymbol{\ell}_{jk}\mathbf{a}_{kj}, \tag{39}$$
$$\mathbf{a}_{jh}^{\mathrm{BCR}} \overset{\mathrm{def}}{=} -\boldsymbol{\ell}_{ji}\mathbf{a}_{ih}, \tag{40}$$
$$\mathbf{a}_{jl}^{\mathrm{BCR}} \overset{\mathrm{def}}{=} -\boldsymbol{\ell}_{jk}\mathbf{a}_{kl}. \tag{41}$$

This process of reduction continues until we are left with only one row, namely $\mathbf{a}_{kk}^{\mathrm{BCR}}$, where k denotes the row/column we finally reduce to. From $\mathbf{a}_{kk}^{\mathrm{BCR}}$, we can determine one block of the inverse via $\mathbf{g}_{kk} = (\mathbf{a}_{kk}^{\mathrm{BCR}})^{-1}$. From this single block, the backward recurrence (12) produces all the other blocks of the inverse. The steps are similar to those in Sections 5.1.1 and 5.1.2. The key difference between the Schur phase and BCR is in the pattern of communication in the parallel implementation. The Schur phase is embarrassingly parallel while BCR requires communication at the end of each step.

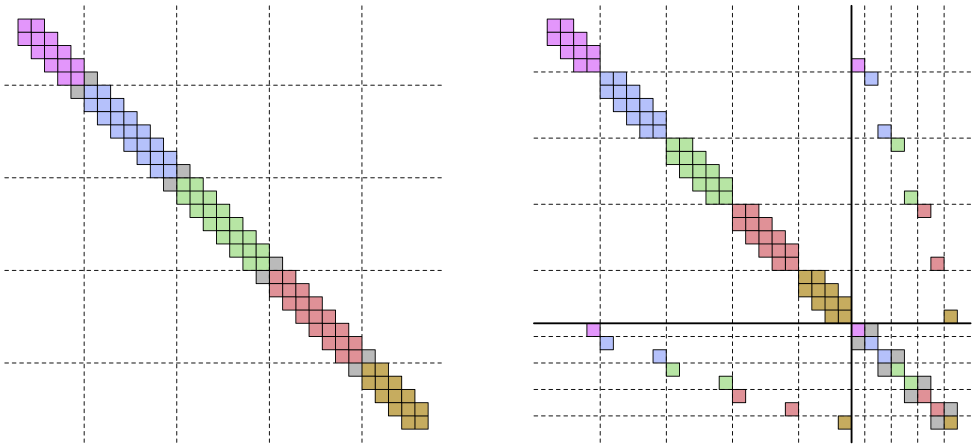


Fig. 7. Permutation for a 31 × 31 block matrix **A** (left), which shows that our Schur reduction phase is identical to an unpivoted Gaussian elimination on a suitably permuted matrix **PAP** (right). Colors are associated with processes. The diagonal block structure of the top left part of the matrix (right panel) shows that this calculation is embarrassingly parallel.

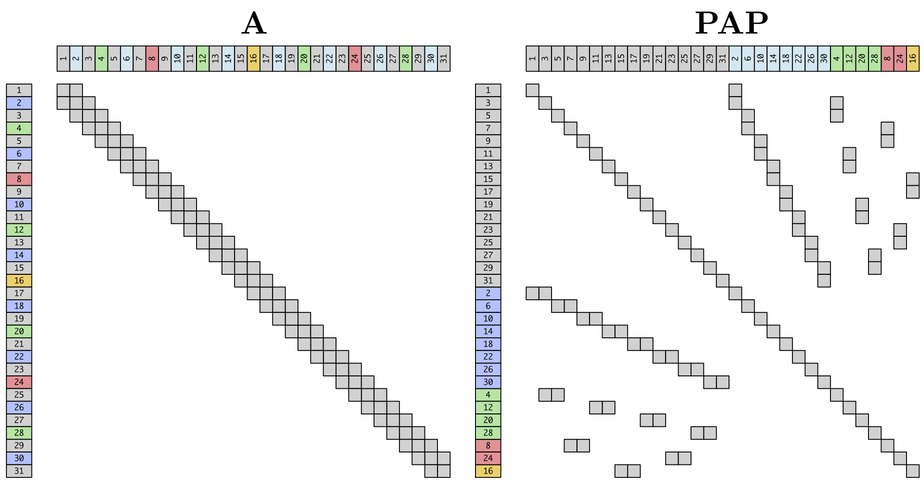


Fig. 8. BCR corresponds to an unpivoted Gaussian elimination of **A** with permutation of rows and columns.

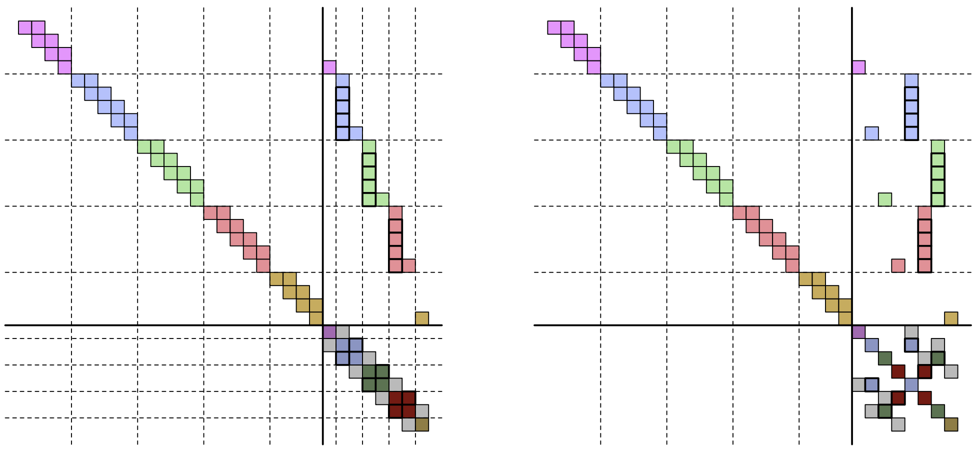


Fig. 9. Following a Schur reduction phase on the permuted block matrix **PAP**, we obtain the reduced block tridiagonal system in the lower right corner (left panel). This reduced system is further permuted to a form shown on the right panel, as was done for BCR in Fig. 8.

6. Numerical results

6.1. Stability

Our elimination algorithm is equivalent to an unpivoted Gaussian elimination on a suitably permuted matrix **PAP** for some permutation matrix **P** [19,20].

Figs. 7 and 8 show the permutation corresponding to the Schur phase and the cyclic reduction phase for a 31×31 block matrix **A**.

In the case of our algorithm, both approaches are combined. The process is shown on Fig. 9. Once the Schur reduction phase has completed on **A**, we are left with a reduced block tridiagonal system. The rows and columns of this tridiagonal system are then be permuted following the BCR pattern.

Our hybrid method is therefore equivalent to an unpivoted Gaussian elimination. Consequently, the stability of the method is dependent on the stability of using the diagonal blocks of **A** as pivots in the elimination process, as is the case for block cyclic reduction.

6.2. Load balancing

For the purposes of load balancing, we will assume that blocks in **A** are of *equal* size. Although this is not the case in general, it is still of use for the investigation of nanodevices that tend to be relatively homogeneous and elongated, and thus giving rise to block tridiagonal matrices **A** with many diagonal blocks and of relatively identical size. Furthermore, this assumption serves as an introductory investigation on how load balancing should be performed, eventually preparing us for a future investigation of the general case.

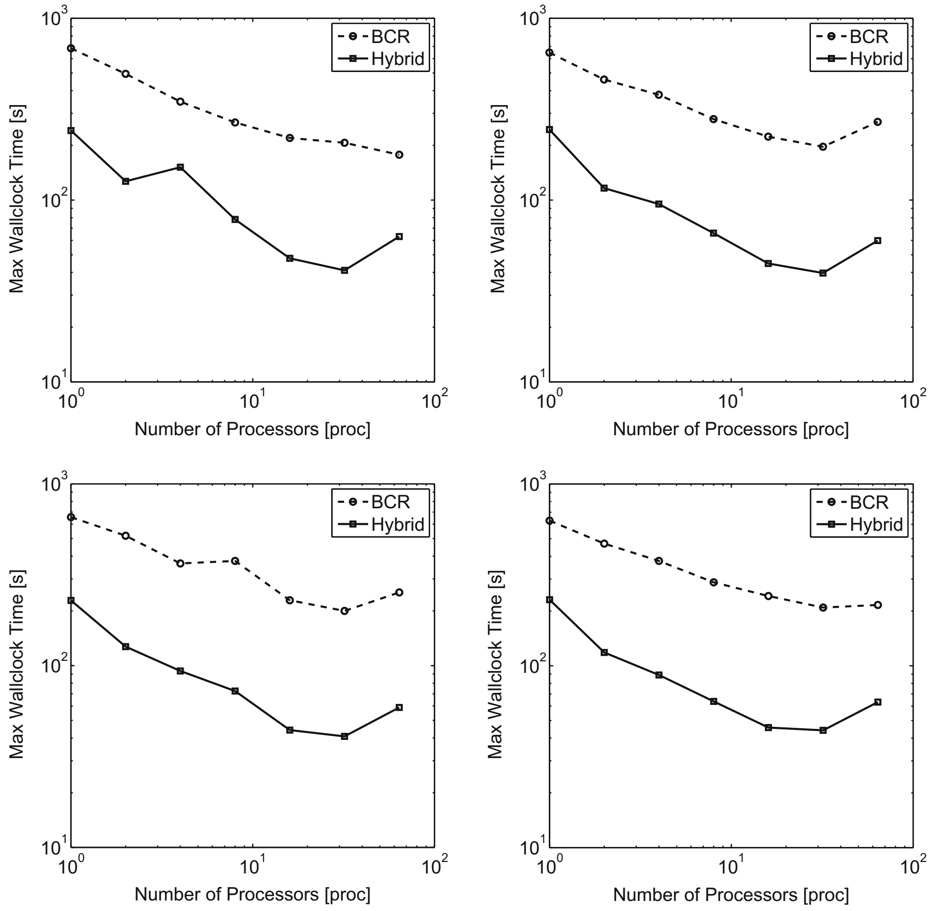


Fig. 10. Walltime for our hybrid algorithm and pure BCR for different αs as a basic load balancing parameter. A block tridiagonal matrix **A** with $n = 512$ diagonal blocks, each of dimension $m = 256$, was used for these tests.

There are essentially two sorts of execution profiles for a process: one for corner processes (p_0 and $p_{\mathcal{P}-1}$) and for central processes. The corner processes perform operations described in Section 5.1.1, while the center processes perform those outlined in 5.1.2.

By performing an operation count under the assumption of equal block sizes throughout **A**, and assuming an LU factorization cost equal to $\frac{2}{3}d^3$ operations and a matrix–matrix multiplication cost of $2d^3$ operations (for a matrix of dimension d, cf. [21]), we estimate the ratio α of number of rows for a corner process to number of rows for central processes. An analysis of our algorithm predicts $\alpha = 2.636$ to be optimal [22]. For the sake of completeness, we investigate the case of $\alpha = 1$, where each process is assigned the same number of rows, while the values $\alpha = 2$ and $\alpha = 3$ are chosen to bracket the optimal choice.

6.3. Benchmarks

The benchmarking of the algorithms presented in this paper was carried out on a Sun Microsystems Sun Fire E25K server. This machine comprises 72 UltraSPARC IV+ dual-core CPUs, yielding a total of 144 CPU cores, each running at 1.8 GHz. Each dual-core CPU had access to 2 MB of shared L2-cache and 32 MB shared L3-cache, with a final layer of 416 GB of RAM.

We estimated that in all probability each participating process in our calculations had exclusive right to a single CPU core. There was no guarantee however that the communication network was limited to handling requests from our benchmarked algorithms. It is in fact highly likely that the network was loaded with other users' applications, which played a role in reducing the performance of our applications. We were also limited to a maximum number of CPU cores of 64.

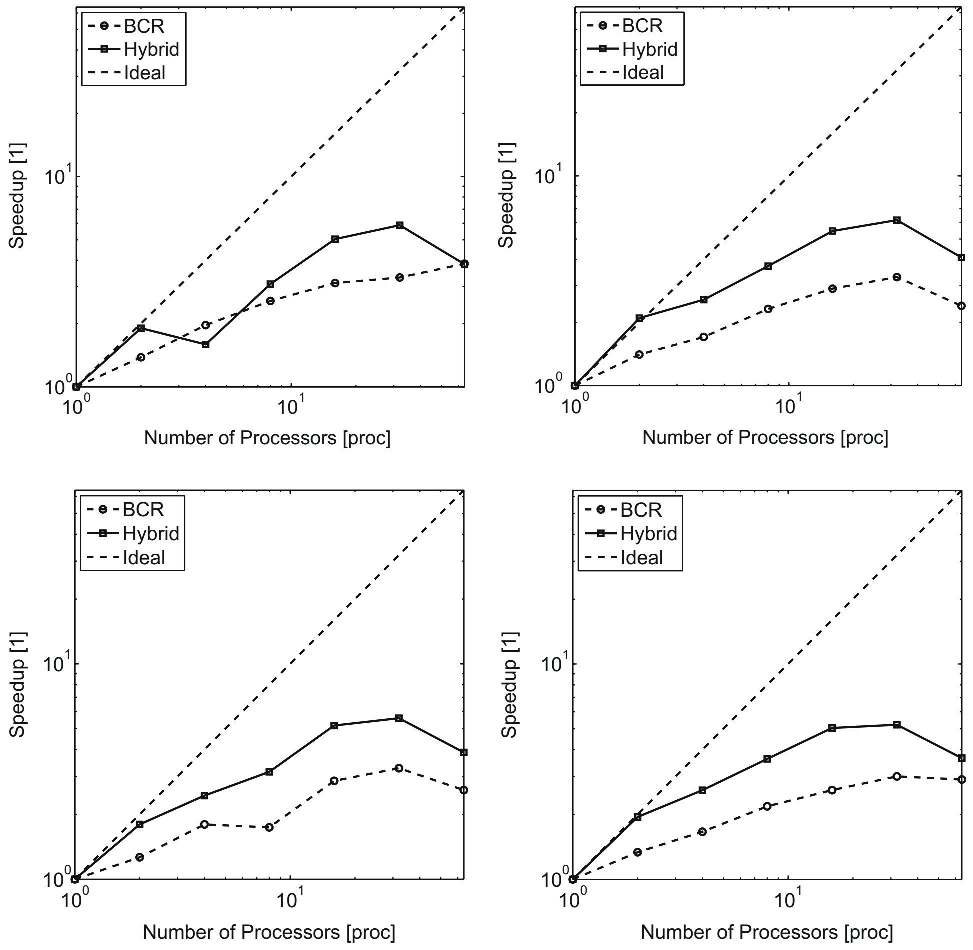


Fig. 11. Speedup curves for our hybrid algorithm and pure BCR for different αs. A block tridiagonal matrix **A** with $n = 512$ diagonal blocks, each of dimension $m = 256$, was used.

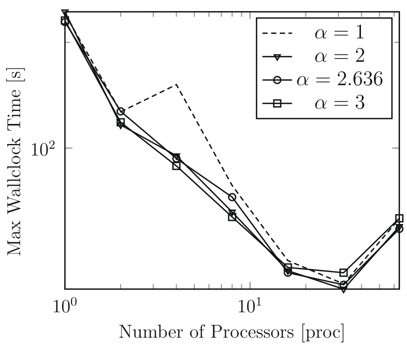


Fig. 12. The total execution time for our hybrid algorithm is plotted against the number of processes $\mathcal{P}$. The choice $\alpha = 1$ gives poor results except when the number of processes is large. Other choices for α give similar results. The same matrix **A** as before was used.

The execution time was measured for a pure block cyclic reduction algorithm (cf. Algorithm 1) in comparison with our hybrid algorithm (cf. Algorithm 2). The walltime measurements for running these algorithms on a block tridiagonal matrix **A** with $n = 512$ block rows with blocks of dimension $m = 256$ is given in Fig. 10 for four different load balancing values $\alpha = \{1, 2, 2.636, 3\}$. The total number of processes used for execution was $\mathcal{P} = \{1, 2, 4, 8, 16, 32, 64\}$ in all cases.

The speedup results corresponding to these walltime measurements are given in Fig. 11 for the four different load balancing values of α. For a uniform distribution where $\alpha = 1$, a serious performance hit is experienced for $\mathcal{P} = 4$. This is due to a poor load balance, as the two corner processes p_0 and p_3 terminate their Schur production/reduction phases much sooner than the central processes p_1 and p_2. It is then observed that choosing α equal 2, 2.636 or 3 eliminates this dip in speedup. The results appear relatively insensitive to the precise choice of α.

It can also be seen that as the number of processes $\mathcal{P}$ increases for a fixed n, a greater portion of execution time is attributed to the BCR phase of our algorithm. Ultimately higher communication and computational costs of BCR over the embarrassingly parallel Schur phase dominate, and the speedup curves level off and drop, regardless of α.

In an effort to determine which load balancing parameter α is optimal, we compare the walltime execution measurements for our hybrid algorithm in Fig. 12. From this figure, we can conclude that a uniform distribution with $\alpha = 1$ leads to poorer execution times; other values of α produce improved but similar results, particularly in the range $\mathcal{P} = 4 \ldots 8$, which is a common core count in modern desktop computers.

7. Conclusion

We have proposed new recurrence formulas to calculate certain entries of the inverse of a sparse matrix. This problem has application in quantum transport and quantum mechanical models, to calculate Green's and lesser Green's function matrices for example. This calculation scales like N^3 for a matrix of size N using a naive algorithm. We have shown that the computational cost can be reduced by orders of magnitude using novel recurrences Eqs. (13) and (20). This is an extension of the work of K. Takahashi [11] and others.

A hybrid algorithm for the parallel implementation of these recurrences has been developed that performs well on many processes. This hybrid algorithm combines Schur complement calculations that are embarrassingly parallel with block cyclic reduction. The performance depends on the problem size and the efficiency of the computer cluster network. On a small test case, we observed scalability up to 32 processes. For electronic structure calculations using density functional theory (DFT), this parallel algorithm can be combined with a parallelization over energy points. This will allow running DFT computations for quantum transport with unprecedented problem sizes.

We note that more efficient parallel implementations of cyclic reduction exist. In particular they can reduce the total number of passes in the algorithm. This will be the object of a future paper.

Acknowledgment

This work was supported by Grant Number 2106-04-0017, "Parallel Algorithms for Computational Nano-Science", under the NABIIT program from the Danish Council for Strategic Research, the Army High Performance Computing and Research Center (AHPCRC) at Stanford University, funded by the United States Army (Grant number W911NF-07-2-0027), and the Stanford School of Engineering.

Appendix A. Algorithms

This section presents the algorithms of block cyclic reduction and the hybrid method used in this paper. The algorithm InverseBCR given in Algorithm 1 calculates the block tridiagonal portion of the inverse of **A**, namely $\text{Trid}_{\mathbf{A}}\{\mathbf{G}\}$, via a pure block cyclic reduction (BCR) approach. The algorithm performs a BCR reduction on **A** on line 2. This leaves us with a single, final block that can be inverted in order to determine the first block of the inverse, $\mathbf{g}_{kk}$. From here, a call on line 4 takes care of reconstructing the block tridiagonal portion of the inverse **G** using this first block of the inverse $\mathbf{g}_{kk}$, and the stored LU factors in **L** and **U**.

Algorithm 1. InverseBCR(**A**)

```
1: $\mathbf{i}^{\text{BCR}} \leftarrow \{1, 2, \dots, n\}$    *BCR is performed over all of* A
2: $\mathbf{A}, \mathbf{L}, \mathbf{U} \leftarrow$ ReduceBCR (A,L,U, $\mathbf{i}^{\text{BCR}}$)
3: $\mathbf{g}_{kk} = \mathbf{a}_{kk}^{-1}$
4: $\mathbf{G} \leftarrow$ Produce BCR($\mathbf{A}, \mathbf{L}, \mathbf{U}, \mathbf{G}, \mathbf{i}^{\text{BCR}}$)
6: return G
```

The algorithm InverseHybrid returns the same result of the block tridiagonal portion of the inverse, but by using the hybrid technique presented in this paper. A significant difference between this algorithm and that of pure BCR is the specification of which row indices $\mathbf{i}^{\text{BCR}}$ the hybrid method should reduce the full block matrix **A** to, before performing BCR on this reduced block tridiagonal system.

This variable of $\mathbf{i}^{\text{BCR}}$ is explicitly used as an argument in the BCR phases, and implicitly in the Schur phases. The implicit form is manifested through the variables *top* and *bot*, that store the value of the top and bottom row indices "owned" by one of the participating parallel processes.

Algorithm 2. InverseHybrid (**A**, $\mathbf{i}^{\text{BCR}}$)

```
1: $\mathbf{A}, \mathbf{L}, \mathbf{U} \leftarrow$ ReduceSchur(A)
2: $\mathbf{A}, \mathbf{L}, \mathbf{U} \leftarrow$ ReduceBCR(A,L,U,$\mathbf{i}^{\text{BCR}}$)
3: $\mathbf{g}_{kk} = \mathbf{a}_{kk}^{-1}$
4: $\mathbf{G} \leftarrow$ ProduceBCR(A,L,U,G,$\mathbf{i}^{\text{BCR}}$)
5: $\mathbf{G} \leftarrow$ ProduceSchur(A,L,U,G)
6: return G
```

A.1. BCR reduction functions

The reduction phase of BCR is achieved through the use of the following two methods. The method ReduceBCR, given in Algorithm 3, takes a block tridiagonal **A** and performs a reduction phase over the supplied indices given in $\mathbf{i}^{\text{BCR}}$. The associated LU factors are then stored appropriately in matrices **L** and **U**.

ReduceBCR loops over each of the levels on line 3, and eliminates the odd-numbered rows given by line 4. This is accomplished by calls to the basic reduction method Reduce on line 6.

Algorithm 3. ReduceBCR(**A**,**L**,**U**,$\mathbf{i}^{\text{BCR}}$)

```
1: $k \leftarrow$ lengthofi$^{\text{BCR}}$    *size of the "reduced" system*
2: $h \leftarrow \log_2(k)$    *height of the binary elimination tree*
3: for $level = 1$ up to $h$ do
4:    $\mathbf{i}^{\text{elim}} \leftarrow$ determine the active rows
5:    for $row = 1$ up to length of $\mathbf{i}^{\text{elim}}$ do    *eliminate odd active rows*
6:       $\mathbf{A}, \mathbf{L}, \mathbf{U} \leftarrow$ Reduce(A,L,U,row,level,$\mathbf{i}^{\text{elim}}$)
7: return A, L, U
```

The core operation of the reduction phase of BCR are the block row updates performed by the method Reduce given in Algorithm 4, where the full system **A** is supplied along with the LU block matrices **L** and **U**. The value *row* is passed along with $\mathbf{i}^{\text{elim}}$, telling us which row of the block tridiagonal matrix given by the indices in $\mathbf{i}^{\text{elim}}$ we want to reduce towards. The method then looks at neighbouring rows based on the level *level* of the elimination tree we are currently at, and then performs block row operations with the correct stride.

Algorithm 4. REDUCE(**A**,**L**,**U**,*row*,*level*,$\mathbf{i}^{\text{elim}}$)

```
1: $h, i, j, k, l \leftarrow$ get the working indices
2: if $i \geqslant \mathbf{i}^{\text{elim}}[1]$ then                    if there is a row above
3:     $\mathbf{u}_{ij} \leftarrow \mathbf{a}_{ii}^{-1}\mathbf{a}_{ij}^{\text{BCR}}$
4:     $\ell_{ji} \leftarrow \mathbf{a}_{ji}\mathbf{a}_{ii}^{-1}$
5:     $\mathbf{a}_{jj} \leftarrow \mathbf{a}_{jj} - \ell_{ji}\mathbf{a}_{ij}$
6:     if $\mathbf{a}_{ih}$ exists then                      if the row above is not the "top" row
7:         $\mathbf{a}_{jh} \leftarrow -\ell_{ji}\mathbf{a}_{ih}$
8: if $k \leqslant \mathbf{i}^{\text{elim}}[\mathbf{end}]$ then                 if there is a row below
9:     $\mathbf{u}_{kj} \leftarrow \mathbf{a}_{kk}^{-1}\mathbf{a}_{kj}$
10:    $\ell_{jk} \leftarrow \mathbf{a}_{jk}\mathbf{a}_{kk}^{-1}$
11:    $\mathbf{a}_{jj} \leftarrow \mathbf{a}_{jj} - \ell_{jk}\mathbf{a}_{kj}$
12:    if $\mathbf{a}_{kl}$ exists then                      if the row below is not the "bottom" row
13:        $\mathbf{a}_{jl} \leftarrow -\ell_{jk}\mathbf{a}_{kl}$
14: return A,L,U
```

A.2. BCR production functions

The production phase of BCR is performed via a call to PRODUCEBCR given in Algorithm 5. The algorithm takes as input an updated matrix **A**, associated LU factors, an inverse matrix **G** initialized with the first block inverse $\mathbf{g}_{kk}$, and a vector of indices $\mathbf{i}^{\text{BCR}}$ defining the rows/columns of **A** on which BCR is to be performed.

The algorithm works by traversing each level of the elimination tree (line 1), where an appropriate striding length is determined and an array of indices $\mathbf{i}^{\text{prod}}$ is generated. This array is a subset of the overall array of indices $\mathbf{i}^{\text{BCR}}$, and is determined by considering which blocks of the inverse have been computed so far. The rest of the algorithm then works through each of these production indices $\mathbf{i}^{\text{prod}}$, and calls the auxiliary methods CORNERPRODUCE and CENTERPRODUCE.

Algorithm 5. PRODUCEBCR(**A**,**L**,**U**,**G**,$\mathbf{i}^{\text{BCR}}$)

```
1: for $level = h$ down to 1 do
2:     $stride \leftarrow 2^{level} - 1$
3:     $\mathbf{i}^{\text{prod}} \leftarrow$ determine the rows to be produced
4:     for $i = 1$ up to length of $\mathbf{i}^{\text{prod}}$ do
5:         $k_{\text{to}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i]]$
6:         if $i = 1$ then
7:             $k_{\text{from}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i] + stride]$
8:             $\mathbf{G} \leftarrow$ CORNERProduce(A,L,U,G,$k_{\text{from}}$,$k_{\text{to}}$)
9:         if $i \neq 1$ and $i =$ length of $\mathbf{i}^{\text{prod}}$ then
10:            if $\mathbf{i}^{\text{prod}}[\mathbf{end}] \leqslant$ length of $\mathbf{i}^{\text{BCR}} - stride$ then
11:                $k_{\text{above}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i] - stride]$
12:                $k_{\text{below}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i] + stride]$
13:                $\mathbf{G} \leftarrow$ CENTERPRODUCE(A,L,U,G,$k_{\text{above}}$, $k_{\text{to}}$,$k_{\text{below}}$)
14:            else
15:                $k_{\text{from}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i] - stride]$
16:                $\mathbf{G} \leftarrow$ CORNERPRODUCE(A,L,U,G, $k_{\text{from}}, k_{\text{to}}$)
17:        if $i \neq 1$ and $i \neq$ length of $\mathbf{i}^{\text{prod}}$ then
18:            $k_{\text{above}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i] - stride]$
19:            $k_{\text{below}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i] + stride]$
20:            $\mathbf{G} \leftarrow$ CENTERPRODUCE(A,L,U,G, $k_{\text{above}}, k_{\text{to}}, k_{\text{below}}$)
21: return G
```

The auxiliary methods CORNERPRODUCE and CENTERPRODUCE are given in Algorithms 6 and 7.

Algorithm 6. CORNERPRODUCE(**A**,**L**,**U**,**G**, $k_{\text{from}}, k_{\text{to}}$)

```
1: $\mathbf{g}_{k_{\text{from}},k_{\text{to}}} \leftarrow -\mathbf{g}_{k_{\text{from}},k_{\text{from}}}\ell_{k_{\text{from}},k_{\text{to}}}$
2: $\mathbf{g}_{k_{\text{to}},k_{\text{from}}} \leftarrow -\mathbf{u}_{k_{\text{to}},k_{\text{from}}}\mathbf{g}_{k_{\text{from}},k_{\text{from}}}$
3: $\mathbf{g}_{k_{\text{to}},k_{\text{to}}} \leftarrow \mathbf{a}_{k_{\text{to}},k_{\text{to}}}^{-1} - \mathbf{g}_{k_{\text{to}},k_{\text{from}}}\ell_{k_{\text{from}},k_{\text{to}}}$
4: return G
```

Algorithm 7. CenterProduce ($\mathbf{A}, \mathbf{L}, \mathbf{U}, \mathbf{G}, k_{\text{above}}, k_{\text{to}}, k_{\text{below}}$)

```
1: $\mathbf{g}_{k_{\text{above}},k_{\text{to}}} \leftarrow -\mathbf{g}_{k_{\text{above}},k_{\text{above}}}\boldsymbol{\ell}_{k_{\text{above}},k_{\text{to}}} - \mathbf{g}_{k_{\text{above}},k_{\text{below}}}\boldsymbol{\ell}_{k_{\text{below}},k_{\text{to}}}$
2: $\mathbf{g}_{k_{\text{below}},k_{\text{to}}} \leftarrow -\mathbf{g}_{k_{\text{below}},k_{\text{above}}}\boldsymbol{\ell}_{k_{\text{above}},k_{\text{to}}} - \mathbf{g}_{k_{\text{below}},k_{\text{below}}}\boldsymbol{\ell}_{k_{\text{below}},k_{\text{to}}}$
3: $\mathbf{g}_{k_{\text{to}},k_{\text{above}}} \leftarrow -\mathbf{u}_{k_{\text{to}},k_{\text{above}}}\mathbf{g}_{k_{\text{above}},k_{\text{above}}} - \mathbf{u}_{k_{\text{to}},k_{\text{below}}}\mathbf{g}_{k_{\text{below}},k_{\text{above}}}$
4: $\mathbf{g}_{k_{\text{to}},k_{\text{below}}} \leftarrow -\mathbf{u}_{k_{\text{to}},k_{\text{above}}}\mathbf{g}_{k_{\text{above}},k_{\text{below}}} - \mathbf{u}_{k_{\text{to}},k_{\text{below}}}\mathbf{g}_{k_{\text{below}},k_{\text{below}}}$
5: $\mathbf{g}_{k_{\text{to}},k_{\text{to}}} \leftarrow \mathbf{a}^{-1}_{k_{\text{to}},k_{\text{to}}} - \mathbf{g}_{k_{\text{to}},k_{\text{above}}}\boldsymbol{\ell}_{k_{\text{above}},k_{\text{to}}} - \mathbf{g}_{k_{\text{to}},k_{\text{below}}}\boldsymbol{\ell}_{k_{\text{below}},k_{\text{to}}}$
6: return G
```

A.3. Hybrid auxiliary functions

Finally, this subsection deals with the auxiliary algorithms introduced by our hybrid method. Prior to any BCR operation, the hybrid method applies a Schur reduction to **A** in order to reduce it to a smaller block tridiagonal system. This reduction is handled by ReduceSchur given in Algorithm 8, while the final production phase to generate the final block tridiagonal $\text{Trid}_{\mathbf{A}}\{\mathbf{G}\}$ is done by Produceschur in Algorithm 9.

Algorithm 8. ReduceSchur(**A**)

```
1: if myPID=0 and $\mathcal{P} > 1$ then        corner eliminate downwards
2:    for $i = top + 1$ up to $bot$ do
3:       $\boldsymbol{\ell}_{i,i-1} \leftarrow \mathbf{a}_{i,i-1}\mathbf{a}^{-1}_{i-1,i-1}$
4:       $\mathbf{u}_{i-1,i} \leftarrow \mathbf{a}^{-1}_{i-1,i-1}\mathbf{a}_{i-1,i}$
5:       $\mathbf{a}_{ii} \leftarrow \mathbf{a}_{ii} - \boldsymbol{\ell}_{i,i-1}\mathbf{a}_{i-1,i}$
6: if myPID $= \mathcal{P} - 1$ then        corner eliminate upwards
7:    for $i = bot - 1$ down to $top$ do
8:       $\boldsymbol{\ell}_{i,i+1} \leftarrow \mathbf{a}_{i,i+1}\mathbf{a}^{-1}_{i+1,i+1}$
9:       $\mathbf{u}_{i+1,i} \leftarrow \mathbf{a}^{-1}_{i+1,i+1}\mathbf{a}_{i+1,i}$
10:      $\mathbf{a}_{ii} \leftarrow \mathbf{a}_{ii} - \boldsymbol{\ell}_{i,i+1}\mathbf{a}_{i+1,i}$
11: if myPID $\neq 0$ and myPID $\neq \mathcal{P} - 1$ and $\mathcal{P} > 1$ then        center elim. down
12:   for $i = top + 2$ down to $bot$ do
13:      $\boldsymbol{\ell}_{i,i-1} \leftarrow \mathbf{a}_{i,i-1}\mathbf{a}^{-1}_{i-1,i-1}$
14:      $\boldsymbol{\ell}_{top,i-1} \leftarrow \mathbf{a}_{top,i-1}\mathbf{a}^{-1}_{i-1,i-1}$
15:      $\mathbf{u}_{i-1,i} \leftarrow \mathbf{a}^{-1}_{i-1,i-1}\mathbf{a}_{i-1,i}$
16:      $\mathbf{u}_{i-1,top} \leftarrow \mathbf{a}^{-1}_{i-1,i-1}\mathbf{a}_{i-1,top}$
17:      $\mathbf{a}_{ii} \leftarrow \mathbf{a}_{ii} - \boldsymbol{\ell}_{i,i-1}\mathbf{a}_{i-1,i}$
18:      $\mathbf{a}_{top,top} \leftarrow \mathbf{a}_{top,top} - \boldsymbol{\ell}_{top,i-1}\mathbf{a}_{i-1,top}$
19:      $\mathbf{a}_{i,top} \leftarrow -\boldsymbol{\ell}_{i,i-1}\mathbf{a}_{i-1,top}$
20:      $\mathbf{a}_{top,i} \leftarrow -\boldsymbol{\ell}_{top,i-1}\mathbf{a}_{i-1,i}$
21: return A,L,U
```

The Schur reduction algorithm takes the full initial block tridiagonal matrix **A**, and through the implicit knowledge of how the row elements of **A** have been assigned to processes, proceeds to reduce **A** into a smaller block tridiagonal system. This implicit knowledge is provided by the *top* and *bot* variables, which specify the topmost and bottommost row indices for this process.

The reduction Algorithm 8 is then split into three cases. If the process owns the topmost rows of **A**, it performs a corner elimination downwards. If it owns the bottommost rows, it then performs a similar operation, but in an upwards manner. Finally, if it owns rows that reside in the middle of **A**, it performs a center reduction operation.

Finally, once the hybrid method has performed a full reduction and a subsequent BCR portion of production, the algorithm ProduceSchur handles the production of the remaining elements of the inverse **G**.

Algorithm 9. ProduceSchur(**A**,**L**,**U**,**G**)

```
if myPID $= 0$ and $\mathcal{P} > 1$ then        corner produce upwards
   for $i = bot$ down to $top + 1$ do
      $\mathbf{g}_{i,i-1} \leftarrow -\mathbf{g}_{ii}\boldsymbol{\ell}_{i,i-1}$
      $\mathbf{g}_{i-1,i} \leftarrow -\mathbf{u}_{i-1,i}\mathbf{g}_{ii}$
      $\mathbf{g}_{i-1,i-1} \leftarrow \mathbf{a}^{-1}_{i-1,i-1} - \mathbf{u}_{i-1,i}\mathbf{g}_{i,i-1}$
if myPID $= \mathcal{P} - 1$ then        corner produce downwards
   for $i = top$ up to $bot - 1$ do
```

```
        $\mathbf{g}_{i,i+1} \leftarrow -\mathbf{g}_{ii}\boldsymbol{\ell}_{i,i+1}$
        $\mathbf{g}_{i+1,i} \leftarrow -\mathbf{u}_{i+1,i}\mathbf{g}_{ii}$
        $\mathbf{g}_{i+1,i+1} \leftarrow \mathbf{a}^{-1}_{i+1,i+1} - \mathbf{u}_{i+1,i}\mathbf{g}_{i,i+1}$
if myPID $\neq 0$ and myPID $\neq \mathcal{P}-1$ and $\mathcal{P} > 1$ then        center produce up
    $\mathbf{g}_{bot,bot-1} \leftarrow -\mathbf{g}_{bot,top}\boldsymbol{\ell}_{top,bot-1} - \mathbf{g}_{bot,bot}\boldsymbol{\ell}_{bot,bot-1}$
    $\mathbf{g}_{bot-1,bot} \leftarrow -\mathbf{u}_{bot-1,bot}\mathbf{g}_{bot,bot} - \mathbf{u}_{bot-1,top}\mathbf{g}_{top,bot}$
    for $i = bot-1$ up to $top+1$ do
        $\mathbf{g}_{top,i} \leftarrow -\mathbf{g}_{top,top}\boldsymbol{\ell}_{top,i} - \mathbf{g}_{top,i+1}\boldsymbol{\ell}_{i+1,i}$
        $\mathbf{g}_{i,top} \leftarrow -\mathbf{u}_{i,i+1}\mathbf{g}_{i+1,top} - \mathbf{u}_{i,top}\mathbf{g}_{top,top}$
    for $i = bot-1$ up to $top+2$ do
        $\mathbf{g}_{ii} \leftarrow \mathbf{a}^{-1}_{ii} - \mathbf{u}_{i,top}\mathbf{g}_{top,i} - \mathbf{u}_{i,i+1}\mathbf{g}_{i+1,i}$
        $\mathbf{g}_{i-1,i} \leftarrow -\mathbf{u}_{i-1,top}\mathbf{g}_{top,i} - \mathbf{u}_{i-1,i}\mathbf{g}_{ii}$
        $\mathbf{g}_{i,i-1} \leftarrow -\mathbf{g}_{i,top}\boldsymbol{\ell}_{top,i-1} - \mathbf{g}_{ii}\boldsymbol{\ell}_{i,i-1}$
    $\mathbf{g}_{top+1,top+1} \leftarrow \mathbf{a}^{-1}_{top+1,top+1} - \mathbf{u}_{top+1,top}\mathbf{g}_{top,top+1} - \mathbf{u}_{top+1,top+2}\mathbf{g}_{top+2,top+1}$
return G
```

References

[1] International technology roadmap for semiconductors, <http://www.itrs.net/Links/2007ITRS/Home2007.htm>, 2007.

[2] S. Shankar, H. Simka, M. Haverty, Density functional theory and beyond – opportunities for quantum methods in materials modeling semiconductor technology, J. Phys.: Condens. Matter 20 (2008) 64232–64240.

[3] R.S. Friedman, M.C. McAlpine, D.S. Ricketts, D. Ham, C.M. Lieber, Nanotechnology: high-speed integrated nanowire circuits, Nature 434 (2005) 1085.

[4] J. Appenzeller, Y.-M. Lin, J. Knoch, P. Avouris, Band-to-band tunneling in carbon nanotube field-effect transistors, Phys. Rev. Lett. 93 (19) (2004) 196805, doi:10.1103/PhysRevLett.93.196805. <http://dx.doi.org/10.1103/PhysRevLett.93.196805>.

[5] E. Ben-Jacob, Z. Hermon, S. Caspi, DNA transistor and quantum bit element: realization of nano-biomolecular logical devices, Phys. Lett. A 263 (1999) 199–202.

[6] S. Kubatkin, A. Danilov, M. Hjort, J. Cornil, J.L. Bredas, N. Stuhr-Hansen, P. Hedegard, T. Björnholm, Single-electron transistor of a single organic molecule with access to several redox states, Nature 425 (2003) 698.

[7] S. Datta, Electronic Transport in Mesoscopic Systems, Cambridge University Press, Cambridge, UK, 1997.

[8] H. Haug, A.-P. Jauho, Quantum Kinetics in Transport and Optics of Semiconductors, Springer-Verlag, Berlin, 1996.

[9] M. Brandbyge, J.-L. Mozos, P. Ordejón, J. Taylor, K. Stokbro, Density-functional method for nonequilibrium electron transport, Phys. Rev. B 65 (16) (2002) 165401, doi:10.1103/PhysRevB.65.165401.

[10] A. Svizhenko, M.P. Anantram, T.R. Govindan, B. Biegel, R. Venugopal, Two-dimensional quantum mechanical modeling of nanotransistors, J. Appl. Phys. 91 (4) (2002) 2343–2354.

[11] K. Takahashi, J. Fagan, M.-S. Chin, Formation of a sparse bus impedance matrix and its application to short circuit study, in: Eigth PICA Conference on Proceedings, Minneapolis, MN, 1973, pp. 63–69.

[12] A.M. Erisman, W.F. Tinney, On computing certain elements of the inverse of a sparse matrix, Numer. Math. 18 (3) (1975) 177–179.

[13] H. Niessner, K. Reichert, On computing the inverse of a sparse matrix, Int. J. Numer. Meth. Eng. 19 (1983) 1513–1526.

[14] S. Li, S. Ahmed, G. Klimeck, E. Darve, Computing entries of the inverse of a sparse matrix using the FIND algorithm, J. Comput. Phys. 227 (22) (2008) 9408–9427. doi: http://dx.doi.org/10.1016/j.jcp.2008.06.033.

[15] J. Schröder, Zur lösung von potentialaufgaben mit hilfe des differenzenverfahrens, Angew. Math. Mech. 34 (1954) 241–253.

[16] L.A. Hageman, R.S. Varga, Block iterative methods for cyclically reduced matrix equations, Numerische Mathematik 6 (1964) 106–119.

[17] A. George, Nested dissection of a regular finite-element mesh, SIAM J. Numer. Anal. 10 (2) (1973) 345–363.

[18] S. Li, E. Darve, Opimation of the FIND algorithm to compute the inverse of a sparse matrix, in: 13th International Workshop on Computational Electronics, 27–29 May, Beijing, China, 2009.

[19] W. Gander, G.H. Golub, Cyclic reduction – history and applications, in: Scientific Computing: Proceedings of the Workshop 10–12 March 1997, pp. 73–85.

[20] D. Heller, Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems, SIAM J. Num. Anal. 13 (4) (1976) 484–496.

[21] G.H. Golub, C.F.V. Loan, Matrix Computations, third ed., Johns Hopkins University Press, 1996.

[22] D.E. Petersen, Block tridiagonal matrices in electronic structure calculations, Ph.D. Thesis, Dept. of Computer Science, Copenhagen University, 2008.