FAST ALGORITHMS FOR SPARSE MATRIX INVERSE COMPUTATIONS

A DISSERTATION
SUBMITTED TO THE INSTITUTE FOR
COMPUTATIONAL AND MATHEMATICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Song Li
September 2009

Abstract

An accurate and efficient algorithm, called Fast Inverse using Nested Dissection (FIND), has been developed for certain sparse matrix computations. The algorithm reduces the computation cost by an order of magnitude for 2D problems. After discretization on an $N_x \times N_y$ mesh, the previously best-known algorithm Recursive Green's Function (RGF) requires $O(N_x^3 N_y)$ operations, whereas ours requires only $O(N_x^2 N_y)$. The current major application of the algorithm is to simulate the transport of electrons in nano-devices, but it may be applied to other problems as well, e.g., data clustering, imagery pattern clustering, image segmentation.

The algorithm computes the diagonal entries of the inverse of a sparse of finite-difference, finite-element, or finite-volume type. In addition, it can be extended to computing certain off-diagonal entries and other inverse-related matrix computations. As an example, we focus on the retarded Green's function, the less-than Green's function, and the current density in the non-equilibrium Green's function approach for transport problems. Special optimizations of the algorithm are also be discussed. These techniques are applicable to 3D regular shaped domains as well.

Computing inverse elements for a large matrix requires a lot of memory and is very time-consuming even using our efficient algorithm with optimization. We propose several parallel algorithms for such applications based on ideas from cyclic reduction, dynamic programming, and nested dissection. A similar parallel algorithm is discussed for solving sparse linear systems with repeated right-hand sides with significant speedup.

Acknowledgements

The first person I would like to thank is Professor Walter Murray. My pleasant study here at Stanford and the newly established Institute for Computational and Mathematical Engineering would not have been possible without him. I would also very much like to thank Indira Choudhury, who helped me on numerous administrative issues with lots of patience.

I also want to thank Professor Tien-Wen Wiedmann, who has been constantly providing mental support and giving me advice on all aspects of my life here.

I am grateful to Professors Tze Leung Lai, Parviz Moin, George Papanicolaou, and Michael Saunders for willing to be my committee members. In particular, Michael gave lots of comments on my dissertation draft. I believe that his comments not only helped me improve my dissertation, but also will benefit my future academic writing.

I would very much like to thank my advisor Eric Darve for choosing a great topic for me at the beginning and all his help afterwards in the past five years. His constant support and tolerance provided me a stable environment with more than enough freedom to learn what interests me and think in my own pace and style, while at the same time, his advice led my vague and diverse ideas to rigorous and meaningful results. This is almost an ideal environment in my mind for academic study and research—one that I have never actually had before. In particular, the discussions with him in the past years have been very enjoyable. I am very lucky to meet him and have him as my advisor here at Stanford. Thank you, Eric!

Lastly, I thank my mother and my sister for always encouraging me and believing in me.

Contents

List of Tables

List of Figures

List of Notations

All the symbols used in this dissertation are listed below unless they are defined elsewhere.

Table I: Symbols in the physical problems

$\mathcal{A}$	spectral function
b	conduction band valley index
$\mathcal{E}$	energy level
$f(\cdot)$	Fermi distribution function
G	Green's function
G^r	retarded Green's function
G^a	advanced Green's function
$G^<$	less-than Green's function
$\hbar$	Planck constant
$\mathcal{H}$	Hamiltonian
$k, \boldsymbol{k}$	wavenumber
ρ	density matrix
Σ	self-energy (reflecting the coupling effects)
$\mathcal{U}$	potential
$\boldsymbol{v}$	velocity

Table II: Symbols for matrix operations

$\mathbf{A}, \mathbf{A}^T, \mathbf{A}^\dagger$	the sparse matrix from the discretization of the device, its transpose, and its transpose conjugate
$\mathbf{A}(i_1 : i_2, j)$	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 $\mathbf{A}$
$\mathbf{a}_{ij}$	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.
$\mathbf{G}$	Green's function matrix
$\mathbf{G}^r$	retarded Green's function matrix
$\mathbf{G}^a$	advanced Green's function matrix
$\mathbf{G}^<$	lesser Green's function matrix
$\mathbf{\Sigma}$ and $\mathbf{\Gamma}$	self-energy matrix, $\mathbf{\Gamma}$ is used only in Chapter 6 to avoid confusion
$\mathbf{I}$	identity matrix
$\mathbf{L}, \mathbf{U}$	the LU factors of the sparse matrix $\mathbf{A}$
$\mathbf{L}$, $\mathbf{D}$, $\mathbf{U}$	block LDU factorization of $\mathbf{A}$ (used only in Chapter 6)
d	block size (used only in Chapter 6)
n	number of blocks in matrix (used only in Chapter 6)
$g, i, j, k, m, n, p, q, r$	matrix indices
$\mathbf{0}$	zero matrix

Table III: Symbols for mesh description

B	set of boundary nodes of a cluster
C	a cluster of mesh nodes after partitioning
$\bar{\mathsf{C}}$	complement set of C
I	set of inner nodes
M	the set of all the nodes in the mesh
N_x, N_y	size of the mesh from the discretization of 2D device
N	the total number of nodes in the mesh
S	set of private inner nodes
T	tree of clusters

Chapter 1

Introduction

The study of sparse matrix inverse computations come from the modeling and computation of the transport problem of nano-devices, but could be applicable to other problems. I will be focusing on the nano-device transport problem in this chapter for simplicity.

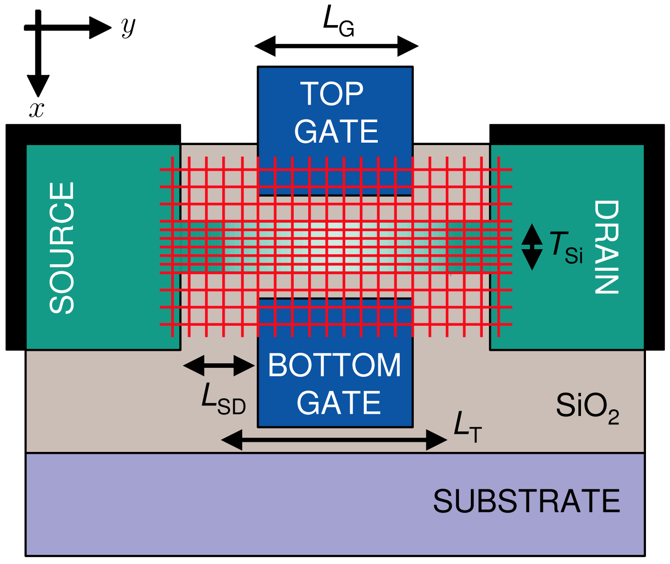
1.1 The transport problem of nano-devices

1.1.1 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 [Wong, 2002]. 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 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 [Wong, 2002; Welser et al., 1992; Vasileska and Ahmed, 2005]), 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 percent performance gain over bulk CMOS [Shahidi, 2002]. Today there is an extensive research in double-gate structures, and FinFET transistors [Wong, 2002], which have better electrostatic integrity and theoretically have better transport properties than single-gated FETs. A number of non-classical and revolutionary technologies 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.

Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) is an example of such devices. 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 [Walls et al., 2004; Hasan et al., 2004] and Fig 1.1. 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 percent 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 [SIA, 2001]. Intrinsic device speed may exceed 1 THz and integration densities will be more than 1 billion transistors/cm^2.



$T_{ox} = 1$ nm
$T_{si} = 5$ nm
$L_G = 9$ nm
$L_T = 17$ nm
$L_{sd} = 10$ nm
$N_{sd} = 10^{20}$ cm^{-3}
$N_b = 0$
$\Phi_G = 4.4227$ eV
(Gate workfunction)

Figure 1.1: The model of a widely-studied double-gate SOI MOSFET with ultra-thin intrinsic channel. Typical values of key device parameters are also shown.

For devices of such size, we have to take the quantum effect into consideration. Although some approximation theories of quantum effects with semiclassical MOSFET modeling tools or other approaches like Landauer-Buttiker formalism are numerically less expensive, they are either not accurate enough or not applicable to realistic devices with interactions (e.g., scattering) that break quantum mechanical phase and cause energy relaxation. (For such devices, the transport problem is between diffusive and phase-coherent.) Such devices include nano-transistors, nanowires, and molecular electronic devices. Although the transport issues for these devices are very different from one another, they can be treated with the common formalism provided by the Non-Equilibrium Green's Function (NEGF) approach [Datta, 2000, 1997]. For example, a successful utilization of the Green's function approach commercially is the NEMO (Nano-Electronics Modeling) simulator [Lake et al., 1997], which is effectively 1D and is primarily applicable to resonant tunneling diodes. This approach, however, is computationally very intensive, especially for 2D devices [Venugopal et al., 2002] (cf. real vs. mode-space approaches). Several methods have been proposed to improve the efficiency but all the currently available algorithms are prohibitively expensive for 2D problem of large size (e.g., a 2D device with 500×1000 grid points after discretization). Our method significantly reduces the computation cost and thus has made most 2D device simulation based on NEGF feasible and some 3D device simulation possible as well.

In the next section, we briefly discuss the NEGF approach for the system in Fig. 1.1 described by the Hamiltonian $\mathcal{H}(\boldsymbol{r}) = \sum_b \mathcal{H}_b(\boldsymbol{r})$ with

$$\mathcal{H}_b(\mathbf{r}) = -\frac{\hbar^2}{2}\left[\frac{d}{dx}\left(\frac{1}{m_x^b}\frac{d}{dx}\right) + \frac{d}{dy}\left(\frac{1}{m_y^b}\frac{d}{dy}\right) + \frac{d}{dz}\left(\frac{1}{m_z^b}\frac{d}{dz}\right)\right] + \mathcal{V}(\mathbf{r}), \tag{1.1}$$

where m_x^b, m_y^b and m_z^b are the components of effective mass in conduction band valley b.

We also have the following assumptions for the system:

1. The Hamiltonian of the device is translate invariant in z-direction (not shown in Fig.1.1) [Venugopal et al., 2002].

2. The conduction band valleys are independent of each other [Svizhenko et al., 2002].

3. There are only outgoing waves at the ends of the device.[Datta, 2000].

With these assumptions, we can treat the Hamiltonian corresponding to each valley separately.

1.1.2 NEGF approach

The NEGF approach is based on the coupled solution of the Schrödinger and Poisson equations. The first step in the NEGF method to model nanoscale devices such as the double-gate SOI MOSFET in Fig. 1.1 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. Since the NEGF approach for problems with or without scattering effect need the same type of computation, for simplicity, we only consider ballistic transport here. After identifying the Hamiltonian matrix and the self-energies, the third step is to compute the retarded Green's function. Once that is known, one can calculate other Green's functions and determine the physical quantities of interest.

Recently, the NEGF approach has been applied in the simulations of 2D MOSFET structures [Svizhenko et al., 2002] as in Fig. 1.1. Below is a summary of the NEGF approach applied to such a device.

I will first introduce the Green's functions used in this approach based on [Datta, 1997; Ferry and Goodnick, 1997].

We start from the simplest case: a 1D wire with a constant potential energy $\mathcal{U}_0$ and zero vector potential. Define $G(x, x')$ as

$$\left(\mathcal{E} - \mathcal{U}_0 + \frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\right)G(x, x') = \delta(x - x'). \tag{1.2}$$

Since $\delta(x-x')$ becomes an identity matrix after discretization, we often write $G(x, x')$ as $\left(\mathcal{E} - \mathcal{U}_0 + \frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\right)^{-1}$.

Since the energy level is degenerate, each $\mathcal{E}$ gives two Green's functions

$$G^r(x, x') = -\frac{i}{\hbar v}e^{+ik|x-x'|} \tag{1.3}$$

and

$$G^a(x, x') = +\frac{i}{\hbar v}e^{-ik|x-x'|}, \tag{1.4}$$

where $k \equiv \frac{\sqrt{2m(\mathcal{E}-\mathcal{U}_0)}}{\hbar}$ is the wavenumber and $v \equiv \frac{\hbar k}{m}$ is the velocity. We will discuss more about the physical interpretation of these two quantities later.

Here $G^r(x, x')$ is called *retarded Green's function* corresponding to the outgoing wave (with respect to excitation $\delta(x, x')$ at $x = x'$) and $G^a(x, x')$ is called *advanced Green's function* corresponding to the incoming wave.

To separate these two states, and more importantly, to help take more complicated boundary conditions into consideration, we introduce an infinitesimal imaginary part

$i\eta$ to the energy, where $\eta \downarrow 0$. We also sometimes write η as 0^+ to make its meaning more obvious. Now we have $G^r = \big(\mathcal{E} + i0^+ - \mathcal{U}_0 + \frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\big)^{-1}$ and $G^a = \big(\mathcal{E} - i0^+ - \mathcal{U}_0 + \frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\big)^{-1}$.

Now we consider the two leads (source and drain) in Fig. 1.1. Our strategy is to still focus on device while summarize the effect of the leads as self-energy. First we consider the effect of the left (source) lead. The Hamiltonian of the system with both the left lead and the device included is given by

$$\begin{pmatrix} \mathbf{H}_L & \mathbf{H}_{LD} \\ \mathbf{H}_{DL} & \mathbf{H}_D \end{pmatrix}. \tag{1.5}$$

The Green's function of the system is then

$$\begin{pmatrix} \mathbf{G}_L & \mathbf{G}_{LD} \\ \mathbf{G}_{DL} & \mathbf{G}_D \end{pmatrix} = \begin{pmatrix} (\mathcal{E} + i0^+)\mathbf{I} - \mathbf{H}_L & -\mathbf{H}_{LD} \\ -\mathbf{H}_{DL} & (\mathcal{E} + i0^+)\mathbf{I} - \mathbf{H}_D \end{pmatrix}^{-1}. \tag{1.6}$$

Eliminating all the columns corresponding to the left lead, we obtain the Schur complement $(\mathcal{E} + i0^+)\mathbf{I} - \mathbf{\Sigma}_L$ and we have

$$\mathbf{G}_D = ((\mathcal{E} + i0^+)\mathbf{I} - \mathbf{\Sigma}_L)^{-1}, \tag{1.7}$$

where $\mathbf{\Sigma}_L = \mathbf{H}_{DL}[(\mathcal{E} + i0^+)\mathbf{I} - \mathbf{H}_L)^{-1}\mathbf{H}_{LD}$ is called the self-energy introduced by the left lead.

Similarly, for the right (drain) lead, we have $\mathbf{\Sigma}_R = \mathbf{H}_{DR}((\mathcal{E} + i0^+)\mathbf{I} - \mathbf{H}_R)^{-1}\mathbf{H}_{RD}$, the self-energy introduced by the right lead. Now we have the Green's function of the system with both the left lead and the right leaded included:

$$\mathbf{G}_D = ((\mathcal{E} + i0^+)\mathbf{I} - \mathbf{\Sigma}_L - \mathbf{\Sigma}_R)^{-1}. \tag{1.8}$$

Now we are ready to compute the Green's function as long as $\mathbf{\Sigma}_L$ and $\mathbf{\Sigma}_R$ are available. Although it seems difficult to compute them since $(\mathcal{E} + i0^+)\mathbf{I} - \mathbf{H}_L$, $(\mathcal{E} + i0^+)\mathbf{I} - \mathbf{H}_R$, $\mathbf{H}_{DL}$, $\mathbf{H}_{DR}$, $\mathbf{H}_{RD}$, and $\mathbf{H}_{LD}$ are of infinite dimension, it turns out that they are often analytically computable. For example, for a semi-infinite discrete wire with grid point distance a in y-direction and normalized eigenfunctions $\{\chi_m(x_i)\}$ as the left lead, the self-energy is given by [Datta, 1997]

$$\mathbf{\Sigma}_L(i,j) = -t\sum_m \chi_m(x_i)e^{+ik_m a}\chi_m(x_j). \tag{1.9}$$

Now we consider the distribution of the electrons. As fermions, the electrons at each energy state $\mathcal{E}$ follow the Fermi-Dirac distribution and the density matrix is given by

$$\rho = \int f_0(\mathcal{E}-\mu)\delta(\mathcal{E}\mathbf{I}-\mathbf{H})dE. \tag{1.10}$$

Since $\delta(x)$ can be written as

$$\delta(x) = \frac{1}{2\pi}\Big(\frac{i}{x+i0^+} - \frac{i}{x-i0^+}\Big), \tag{1.11}$$

we can write the density matrix in terms of the *spectral function* $\mathcal{A}$:

$$\rho = \frac{1}{2\pi}\int f_0(\mathcal{E}-\mu)\mathcal{A}(\mathcal{E})d\mathcal{E}, \tag{1.12}$$

where $\mathcal{A}(\mathcal{E}) \equiv i(G^r - G^a) = iG^r(\Sigma^r - \Sigma^a)G^a$.

We can also define the coupling effect $\Gamma \equiv i(\Sigma^r - \Sigma^a)$. Now, the equations of motion for the retarded Green's function (G^{r}) and less-than Green's function ($G^<$) are found to be (see [Svizhenko et al., 2002] for details and notations)

$$\begin{aligned}\left[\mathcal{E} - \frac{\hbar^2 k_z^2}{2m_z} - \mathcal{H}_b(\boldsymbol{r}_1)\right] G_b^r(\boldsymbol{r}_1,\boldsymbol{r}_2,k_z,\mathcal{E})& \\ - \int \Sigma_b^r(\boldsymbol{r}_1,\boldsymbol{r},k_z,\mathcal{E})\; G_b^r(\boldsymbol{r},\boldsymbol{r}_2,k_z,\mathcal{E})\; d\boldsymbol{r} &= \delta(\boldsymbol{r}_1-\boldsymbol{r}_2)\end{aligned}$$

and

$$G_b^<(\boldsymbol{r}_1,\boldsymbol{r}_2,k_z,\mathcal{E}) = \int G_b^r(\boldsymbol{r}_1,\boldsymbol{r},k_z,\mathcal{E})\; \Sigma_b^<(\boldsymbol{r},\boldsymbol{r}',k_z,\mathcal{E})\; G_b^r(\boldsymbol{r}_2,\boldsymbol{r}',k_z,\mathcal{E})^\dagger\; d\boldsymbol{r}\, d\boldsymbol{r}',$$

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:

$$\begin{aligned}\mathrm{DOS}(\boldsymbol{r},k_z,\mathcal{E}) &= \sum_b N_b(\boldsymbol{r},k_z,\mathcal{E}) = -\frac{1}{\pi}\sum_b \mathrm{Im}[G_b^r(\boldsymbol{r},\boldsymbol{r},k_z,\mathcal{E})] \\ \rho(\boldsymbol{r},k_z,\mathcal{E}) &= \sum_b \rho_b(\boldsymbol{r},k_z,\mathcal{E}) = -i\sum_b G_b^<(\boldsymbol{r},\boldsymbol{r},k_z,\mathcal{E}).\end{aligned}$$

The self-consistent solution of the Green's function is often the most time-intensive step in the simulation of electron density.

To compute the solution numerically, we discretize the intrinsic device 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.

Recursive Green's Function (RGF) [Svizhenko et al., 2002] 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 $N_x^3 N_y$ have been, for the past few years, very desirable. Our newly developed FIND algorithm addresses this particular issue of computing the diagonal of $\mathbf{G}^\mathrm{r}$ and $\mathbf{G}^<$, thereby reducing the simulation time of NEGF significantly compared to the conventional RGF scheme.

1.2 Review of existing methods

1.2.1 The need for computing sparse matrix inverses

The most expensive calculation in the NEGF approach is computing some of (but not all) the entries of the matrix $\mathbf{G}^\mathrm{r}$ [Datta, 2000]:

$$\mathbf{G}^\mathrm{r}(\mathcal{E}) = [\mathcal{E}\mathbf{I} - \mathbf{H} - \mathbf{\Sigma}]^{-1} = \mathbf{A}^{-1} \qquad \text{(retarded Green's function)} \tag{1.13}$$

and $\mathbf{G}^<(\mathcal{E}) = \mathbf{G}^{\mathrm{r}}\, \mathbf{\Sigma}^<\, (\mathbf{G}^{\mathrm{r}})^\dagger$ (less-than Green's function). In these equations, I is the identity matrix, and $\mathcal{E}$ is the energy level. $\dagger$ denotes the transpose conjugate of a matrix. The Hamiltonian matrix $\mathbf{H}$ 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 that may have a non-local coupling (e.g., non-reflecting boundary condition). The matrices $\mathbf{\Sigma}$ and $\mathbf{\Sigma}^<$ correspond to the self energy and can be assumed to be diagonal. See Svizhenko [Svizhenko et al., 2002] for this terminology and notation. In our work, all these matrices are considered to be given and we will focus on the problem of efficiently computing some entries in $\mathbf{G}^{\mathrm{r}}$ and $\mathbf{G}^<$. As an example of entries that must be computed, the diagonal entries of $\mathbf{G}^{\mathrm{r}}$ are required to compute the density of states, while the diagonal entries of $\mathbf{G}^<$ allow computing the electron density. The current can be computed from the super-diagonal entries of $\mathbf{G}^<$.

Even though the matrix $\mathbf{A}$ in (1.13) is, by the usual standards, a mid-size sparse matrix of size typically $10,000 \times 10,000$, computing the entries of $\mathbf{G}^<$ is a major challenge because this operation is repeated at all energy levels for every iteration of the Poisson-Schrödinger solver. Overall, the diagonal of $\mathbf{G}^<(\mathcal{E})$ for the different values of the energy level $\mathcal{E}$ 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-squares fitting In the linear least-square fitting procedure, coefficients a_k are computed so that the error measure

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

is minimal, where (x_i, Y_i) are the data points. It can be shown, under certain assumptions, 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 of the matrix A, where

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

Or we can write them with notation from the literature of linear algebra. Let $\boldsymbol{b} = [Y_1, Y_2, \ldots]^T$, $\mathbf{M}_{ik} = \phi_k(x_i)$, $\boldsymbol{x} = [a_1, a_2, \ldots]^T$, and $\mathbf{A} = \mathbf{M}^T\mathbf{M}$. Then we have the least-squares problem $\min_{\boldsymbol{x}} \|\boldsymbol{b} - \mathbf{M}\boldsymbol{x}\|_2$ and the sensitivity of $\boldsymbol{x}$ depends on the diagonal entries of $\mathbf{A}^{-1}$.

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 [Varah, 1968] and Wilkinson[Peters and Wilkinson, 1971; Wilkinson, 1972; Peters and Wilkinson, 1979] have extensively discussed optimal choices of starting vectors for this method. An important result is that, in general, choosing the vector $\boldsymbol{e}_l$ (l^{th} 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 l^{th} 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 $\boldsymbol{b}$, 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.

1.2.2 Direct methods for matrix inverse related computation

There are several existing direct methods for sparse matrix inverse computations. These methods are mostly based on [Takahashi et al., 1973]'s observation that if the matrix $\mathbf{A}$ is decomposed using an LU factorization $\mathbf{A} = \mathbf{LDU}$, then we have

$$\mathbf{G}^{\mathrm{r}} = \mathbf{A}^{-1} = \mathbf{D}^{-1}\mathbf{L}^{-1} + (\mathbf{I} - \mathbf{U})\mathbf{G}^{\mathrm{r}}, \quad \text{and} \tag{1.14}$$

$$\mathbf{G}^{\mathrm{r}} = \mathbf{U}^{-1}\mathbf{D}^{-1} + \mathbf{G}^{\mathrm{r}}(\mathbf{I} - \mathbf{L}). \tag{1.15}$$

The methods for computing certain entries of the sparse matrix inverse based on Takahashi's observation can lead to methods based on 2-pass computation: one for LU factorization and the other for a special back substitution. We call these methods *2-way methods*. The algorithm of [Erisman and Tinney, 1975]'s algorithm is such a method. Let's define a matrix $\mathbf{C}$ such that

$$\mathbf{C}_{ij} = \begin{cases} 1, & \text{if } \mathbf{L}_{ij} \text{ or } \mathbf{U}_{ij} \neq 0 \\ 0, & \text{otherwise.} \end{cases}$$

Erisman and Tinney showed the following theorem:

> If $\mathbf{C}_{ji} = 1$, any entry $\mathbf{G}^{\mathrm{r}}_{ij}$ can be computed as a function of $\mathbf{L}$, $\mathbf{U}$, $\mathbf{D}$ and entries $\mathbf{G}^{\mathrm{r}}_{pq}$ such that $p \geq i$, $q \geq j$, and $\mathbf{C}_{qp} = 1$.

This implies that efficient recursive equations can be constructed. Specifically, from (2.2), for $i < j$,

$$\mathbf{G}^{\mathrm{r}}_{ij} = -\sum_{k=i+1}^{n} \mathbf{U}_{ik}\mathbf{G}^{\mathrm{r}}_{kj}.$$

The key observation is that if we want to compute $\mathbf{G}^{\mathrm{r}}_{ij}$ with $\mathbf{L}_{ji} \neq 0$ and $\mathbf{U}_{ik} \neq 0$ ($k > i$) then $\mathbf{C}_{jk}$ must be 1. This proves that the theorem holds in that case. A similar result holds for $j < i$ using (2.3). For $i = j$, we get (using (2.2))

$$\mathbf{G}^{\mathrm{r}}_{ii} = (\mathbf{D}_{ii})^{-1} - \sum_{k=i+1}^{n} \mathbf{U}_{ik}\mathbf{G}^{\mathrm{r}}_{ki}.$$

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

An application of Takahashi's observation to computing both $\mathbf{G}^r$ and $\mathbf{G}^<$ on a 1D system is the Recursive Green's Function (RGF) method. RGF is a state-of-the-art method and will be used as a benchmark to compare with our methods. We will discuss it in detail in section 1.2.3.

In RGF, the computation on each way is conducted column by column in sequence, from one end to the other. Such nature will make the method difficult to parallelize. Chapter 6 shows an alternative method based on cyclic reduction for the LU factorization and back substitution.

An alternative approach to the 2-way methods is our FIND-1way algorithm, which we will discuss in detail in Chapter 3. In FIND-1way, we compute the inverse solely through LU factorization without the need for back substitution. This method can be more efficient under certain circumstances and may lead to further parallelization.

In all these methods, we need to conduct the LU factorization efficiently. For 1D systems, a straightforward or simple approach will work well. For 2D systems, however, we need more sophisticated methods. Elimination tree [Schreiber, 1982], min-degree tree [Davis, 2006], etc. are common approaches for the LU factorization of sparse matrices. For a sparse matrix with local connectivity in its connectivity graph, nested dissection has proved to be a simple and efficient method. We will use nested dissection methods in our FIND-1way, FIND-2way, and various parallel FIND algorithms as well.

Independent of the above methods, K. Bowden developed an interesting set of matrix sequences that allows us in principle to calculate the inverse of any block tri-diagonal matrices very efficiently [Bowden, 1989]. Briefly speaking, four sequences of matrices, $\mathbf{K}_p$, $\mathbf{L}_p$, $\mathbf{M}_p$ and $\mathbf{N}_p$, are defined using recurrence relations. Then an expression is found for any block (i, j) of matrix $\mathbf{G}^\mathrm{r}$:

$$\begin{aligned} j \geq i, &\quad \mathbf{G}^\mathrm{r}_{ij} = \mathbf{K}_i \mathbf{N}_0^{-1} \mathbf{N}_j, \\ j \leq i, &\quad \mathbf{G}^\mathrm{r}_{ij} = \mathbf{L}_i \mathbf{L}_0^{-1} \mathbf{M}_j. \end{aligned}$$

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

1.2.3 Recursive Green's Function method

Currently the state-of-the-art is a method developed by Klimeck and Svizhenko et al. [Svizhenko et al., 2002], 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 that are very elongated in one direction and very thin in the two other directions. This algorithm has a running time of order $\mathcal{O}(N_x^3 N_y)$, where N_x and N_y are the number of points on the grid in the x and y direction, respectively.

The key point of this algorithm is that even though it appears that the full knowledge of $\mathbf{G}$ is required in order to compute the diagonal blocks of $\mathbf{G}^<$, in fact we will show that the knowledge of the three main block diagonals of $\mathbf{G}$ is sufficient to calculate the three main block diagonals of $\mathbf{G}^<$. A consequence of this fact is that it is possible to compute the diagonal of $\mathbf{G}^<$ with linear complexity in N_y block operations. If all the entries of $\mathbf{G}$ were required, the computational cost would be $O(N_x^3 N_y^3)$.

Assume that the matrix $\mathbf{A}$ 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 in Fig. 1.2.

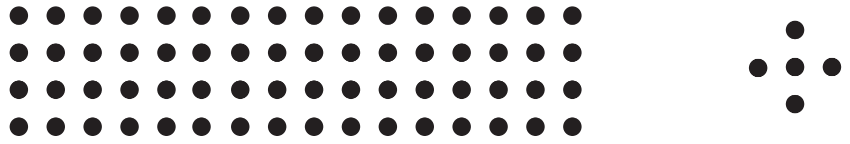


Figure 1.2: Left: example of 2D mesh to which RGF can be applied. Right: 5-point stencil.

For a 5-point stencil, the matrix $\mathbf{A}$ can be written as a tri-diagonal block matrix where blocks on the diagonal are denoted by $\mathbf{A}_q$ $(1 \leq i \leq n)$, on the upper diagonal by $\mathbf{B}_q$ $(1 \leq i \leq n-1)$, and on the lower diagonal by $\mathbf{C}_q$ $(2 \leq i \leq n)$.

RGF computes the diagonal of $\mathbf{A}^{-1}$ by computing recursively two sequences. The first sequence, in increasing order, is defined recursively as [Svizhenko et al., 2002]

$$\mathbf{F}_1 = \mathbf{A}_1^{-1}$$
$$\mathbf{F}_q = (\mathbf{A}_q - \mathbf{C}_q\mathbf{F}_{q-1}\mathbf{B}_{q-1})^{-1}.$$

The second sequence, in decreasing order, is defined as

$$\mathbf{G}_n = \mathbf{F}_n$$
$$\mathbf{G}_q = \mathbf{F}_q + \mathbf{F}_q\mathbf{B}_q\mathbf{G}_{q+1}\mathbf{C}_{q+1}\mathbf{F}_q.$$

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

We first present the algorithm to calculate the three main block diagonals of $\mathbf{G}$ where $\mathbf{AG} = \mathbf{I}$. Then we will discuss how it works for computing $\mathbf{G}^< = \mathbf{A}^{-1}\mathbf{\Sigma}\mathbf{A}^{-\dagger}$. Both algorithms consist of a forward recurrence stage and a backward recurrence stage.

In the forward recurrence stage, we denote

$$\mathbf{A}^q \overset{def}{=} \mathbf{A}_{1:q,1:q} \tag{1.16}$$
$$\mathbf{G}^q \overset{def}{=} (\mathbf{A}^q)^{-1}. \tag{1.17}$$

Proposition 1.1 *The following equations can be used to calculate* $\mathbf{Z}_q := \mathbf{G}^q_{qq}$, $1 \le q \le n$:

$$\mathbf{Z}_1 = \mathbf{A}_{11}^{-1}$$
$$\mathbf{Z}_q = \left(\mathbf{A}_{qq} - \mathbf{A}_{q,q-1}\mathbf{Z}_{q-1}\mathbf{A}_{q-1,q}\right)^{-1}.$$

Sketch of the Proof: Consider the LU factorization of $\mathbf{A}^{q-1} = \mathbf{L}^{q-1}\mathbf{U}^{q-1}$ and

$\mathbf{A}^q = \mathbf{L}^q\mathbf{U}^q$. Because of the block-tridiagonal structure of $\mathbf{A}$, we can eliminate the q^{th} block column of $\mathbf{A}$ based on the (q,q) block entry of $\mathbf{A}$ after eliminating the $(q-1)^{th}$ block column. In equations, we have $\mathbf{U}^q_{qq} = \mathbf{A}_{qq} - \mathbf{A}_{q,q-1}(\mathbf{U}^{q-1}_{q-1,q-1})^{-1}\mathbf{A}_{q-1,q} = \mathbf{A}_{qq} - \mathbf{A}_{q,q-1}\mathbf{Z}_{q-1}\mathbf{A}_{q-1,q}$ and then $\mathbf{Z}_q = (\mathbf{U}_{qq})^{-1} = (\mathbf{A}_{qq} - \mathbf{A}_{q,q-1}\mathbf{Z}_{q-1}\mathbf{A}_{q-1,q})^{-1}$. For more details, please see the full proof in the appendix. □

The last element is such that $\mathbf{Z}_n = \mathbf{G}^n_{nn} = \mathbf{G}_{nn}$. From this one we can calculate all the $\mathbf{G}_{qq}$ and $\mathbf{G}_{q,q\pm1}$. This is described in the next subsection.

Now, we will show that in the backward recurrence stage.

Proposition 1.2 *The following equations can be used to calculate* $\mathbf{G}_{qq}$, $1 \leq q \leq n$ *based on* $\mathbf{Z}_q = \mathbf{G}^q_{qq}$*:*

$$\begin{aligned}\mathbf{G}_{nn} &= \mathbf{Z}_n\\ \mathbf{G}_{qq} &= \mathbf{Z}_q + \mathbf{Z}_q\mathbf{A}_{q,q+1}\mathbf{G}_{q+1,q+1}\mathbf{A}_{q+1,q}\mathbf{Z}_q.\end{aligned}$$

Sketch of the Proof: Consider the LU factorization of $\mathbf{A} = \mathbf{LU} \Rightarrow \mathbf{G}_{qq} = (\mathbf{U}^{-1}\mathbf{L}^{-1})_{qq}$. Since $\mathbf{U}^{-1}$ and $\mathbf{L}^{-1}$ are band-2 block bi-diagonal matrices, respectively, we have $\mathbf{G}_{qq} = (\mathbf{U}^{-1})_{q,q} - (\mathbf{U}^{-1})_{q,q+1}(\mathbf{L}^{-1})_{q+1,q}$. Note that $(\mathbf{U}^{-1}_{qq}) = \mathbf{Z}_q$ from the previous section, $(\mathbf{U}^{-1})_{q,q+1} = -\mathbf{Z}_q\mathbf{A}_{q,q+1}\mathbf{Z}_{q+1}$ by backward substitution, and $(\mathbf{L}^{-1})_{q+1,q} = -\mathbf{A}_{q+1,q}\mathbf{Z}_q$ is the elimination factor when we eliminate the q^{th} block column in the previous section, we have $\mathbf{G}_{qq} = \mathbf{Z}_q + \mathbf{Z}_q\mathbf{A}_{q,q+1}\mathbf{G}_{q+1,q+1}\mathbf{A}_{q+1,q}\mathbf{Z}_q$. □

In the recurrence for the $\mathbf{A}\mathbf{G}^<\mathbf{A}^* = \mathbf{\Sigma}$ stage, the idea is the same as the recurrence for computing $\mathbf{G}^r$. However, since we have to apply similar operations to $\mathbf{\Sigma}$ from *both* sides, the operations are a lot more complicated. Since the elaboration of the operations and the proofs is tedious, we put it in the appendix.

Chapter 2

Overview of FIND Algorithms

Our serial methods, FIND-1way and FIND-2way, are both based on nested dissection. FIND-2way is based on Takahashi's observation but extends to computing $\mathbf{G}^{<}$. It is similar to the existing RGF method and works efficiently in 2D as well. In contrast, FIND-1way is independent of Takahashi's observation but based on three other observations: 1) we can obtain the diagonal entries of the matrix inverse solely based on multiple LU factorizations without the need for back substitution; 2) each of the multiple LU factorizations can be decomposed into partial LU factorizations; 3) the multiple LU factorizations significantly overlap and such overlap reduces cost by two orders of magnitude. Moreover, avoiding the back substitution makes its parallelization more efficient.

2.1 1-way methods based on LU only

The basic idea of the FIND algorithm is to perform many LU factorizations on the given matrix $\mathbf{A}$ to compute the diagonal elements of its inverse. The algorithm is a direct method based on the fact that $\mathbf{G}_{nn} = 1/\mathbf{U}_{nn}$, where $\mathbf{U}$ is the upper triangular matrix in the LU factorization of $\mathbf{A}$, which is a full rank matrix of size $N \times N$ with $N = N_x N_y$ as the total number of nodes in the mesh. If we perform LU factorizations of $\mathbf{A}$ with different orderings, we can compute all the diagonal entries of $\mathbf{G}$.

Because of the local connectivity of the mesh nodes that leads to the sparsity of

A, we can decompose LU factorizations into partial LU factorizations. As a result, we can perform them independently and reuse the partial results multiple times for different orderings. If we reorder **A** properly, each of the partial factorizations is quite efficient (similar to the nested dissection procedure to minimize fill-ins [George, 1973]), and many of them turn out to be identical.

More specifically, we partition the mesh into subsets of mesh nodes first and then merge these subsets to form clusters. Fig. 2.1 shows the partitions of the entire mesh into 2, 4, and 8 clusters.

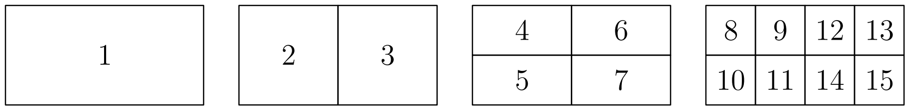


Figure 2.1: Partitions of the entire mesh into clusters

Fig. 2.2 shows the partition tree and complement tree of the mesh formed by the clusters in Fig. 2.1 and their complements.

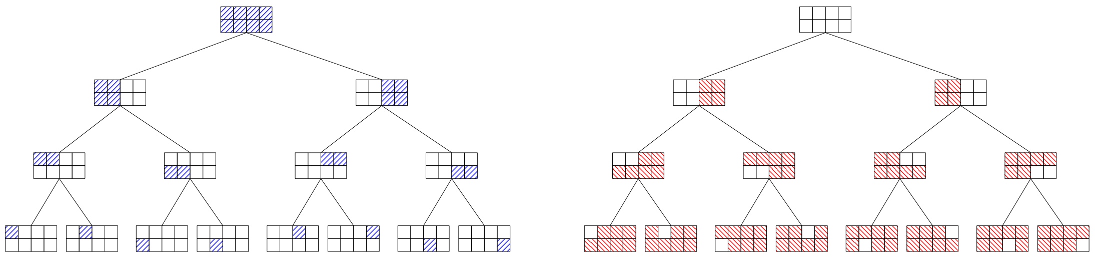


Figure 2.2: The partition tree and the complement tree of the mesh

The two cluster trees in Fig. 2.3 (*augmented tree*) show how two leaf complement clusters $\bar{14}$ and $\bar{15}$ are computed by the FIND algorithm. The path $\bar{3} - \bar{7} - \bar{14}$ in the left tree and the path $\bar{3} - \bar{7} - \bar{15}$ in the right tree are part of the complement-cluster tree; the other subtrees are copies from the basic cluster tree.

At each level of the two trees, two clusters merge into a larger one. Each merge eliminates all the inner nodes of the resulting cluster. For example, the merge in Fig. 2.4 eliminates all the $\circ$ nodes.

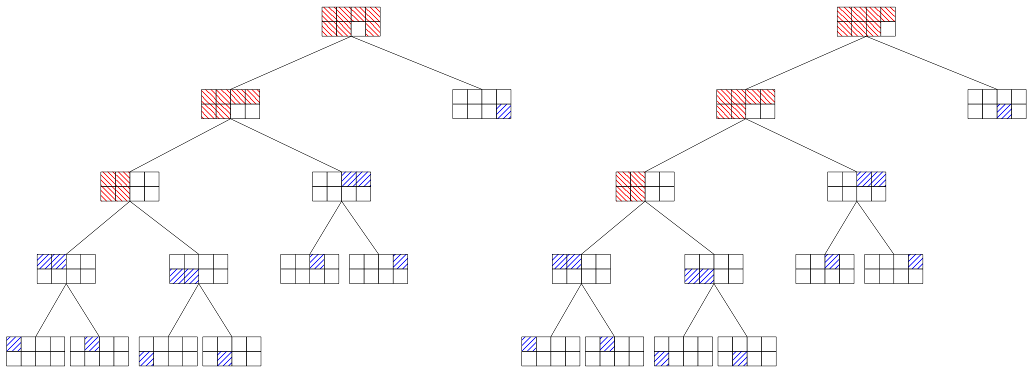


Figure 2.3: Two augmented trees with root clusters $\bar{14}$ and $\bar{15}$.

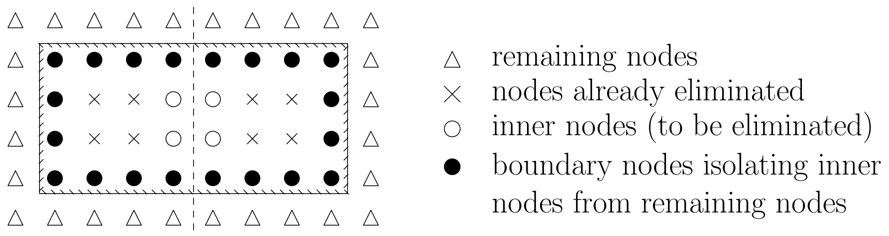

△ remaining nodes
× nodes already eliminated
○ inner nodes (to be eliminated)
● boundary nodes isolating inner nodes from remaining nodes

Figure 2.4: The nodes within the rectangle frame to the left of the dashed line and those to the right form a larger cluster.

Such a merge corresponds to an independent partial LU factorization because there is no connection between the eliminated nodes (∘) and the remaining nodes (△). This is better seen below in the Gaussian elimination in (2.1) for the merge in Fig. 2.4:

$$\begin{pmatrix} \mathbf{A}(\circ,\circ) & \mathbf{A}(\circ,\bullet) & 0 \\ \mathbf{A}(\bullet,\circ) & \mathbf{A}(\bullet,\bullet) & \mathbf{A}(\bullet,\triangle) \\ 0 & \mathbf{A}(\triangle,\bullet) & \mathbf{A}(\triangle,\triangle) \end{pmatrix} \Rightarrow \begin{pmatrix} \mathbf{A}(\circ,\circ) & \mathbf{A}(\circ,\bullet) & 0 \\ 0 & \mathbf{A}^*(\bullet,\bullet) & \mathbf{A}(\bullet,\triangle) \\ 0 & \mathbf{A}(\triangle,\bullet) & \mathbf{A}(\triangle,\triangle) \end{pmatrix},$$

$$\text{where } \mathbf{A}^*(\bullet,\bullet) = \mathbf{A}(\bullet,\bullet) - \mathbf{A}(\bullet,\circ)\mathbf{A}(\circ,\circ)^{-1}\mathbf{A}(\circ,\bullet). \tag{2.1}$$

In (2.1), $\circ$ and $\bullet$ each corresponds to all the $\circ$ nodes and $\bullet$ nodes in Fig. 2.4, respectively. $\mathbf{A}^*(\bullet,\bullet)$ is the result of the partial LU factorization. It will be stored for reuse when we pass through the cluster trees.

To make the most of the overlap, we do not perform the eliminations for each augmented tree individually. Instead, we perform the elimination based on the basic cluster tree and the complement-cluster tree in the FIND algorithm. We start by eliminating all the inner nodes of the leaf clusters of the partition tree in Fig. 2.2 followed by eliminating the inner nodes of their parents recursively until we reach the root cluster. This is called the *upward pass*. Once we have the partial elimination results from upward pass, we can use them when we pass the complement-cluster tree level by level, from the root to the leaf clusters. This is called the *downward pass*.

2.2 2-way methods based on both LU and back substitution

In contrast to the need for only LU factorization in FIND-1way, FIND-2way needs two phases in computing the inverse: LU factorization phase (called *reduction phase*— in Chapter 6) and back substitution phase (called *production phase* in Chapter 6). These two phases are the same as what we need for solving a linear system through Gaussian elimination. In our problem, however, since the given matrix $\mathbf{A}$ is sparse and only the diagonal entries of the matrices $\mathbf{G}^r$ and $\mathbf{G}^<$ are required in the iterative process, we may significantly reduce the computational cost using a number of different methods. Most of these methods are based on Takahashi's observation [Takahashi et al., 1973; Erisman and Tinney, 1975; Niessner and Reichert, 1983; Svizhenko et al., 2002]. In these methods, a block LDU (or LU in some methods) factorization of $\mathbf{A}$ is computed. Simple algebra shows that

$$\mathbf{G}^r = \mathbf{D}^{-1}\mathbf{L}^{-1} + (\mathbf{I} - \mathbf{U})\mathbf{G}^r, \tag{2.2}$$

$$= \mathbf{U}^{-1}\mathbf{D}^{-1} + \mathbf{G}^r(\mathbf{I} - \mathbf{L}), \tag{2.3}$$

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 $\mathbf{G}^r$ 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

$$\mathbf{g}_{ij} = -\sum_{k>i} \mathbf{u}_{ik}\mathbf{g}_{kj}. \tag{2.4}$$

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 $j = i$:

$$\mathbf{g}_{ij} = -\sum_{k>i} \mathbf{g}_{ik}\boldsymbol{\ell}_{kj} \tag{2.5}$$

$$\mathbf{g}_{ii} = \mathbf{d}_{ii}^{-1} - \sum_{k>i} \mathbf{u}_{ik}\mathbf{g}_{ki}. \tag{2.6}$$

This approach leads to fast backward recurrences, as shown in [Erisman and Tinney, 1975].

The generalization of Takahashi's method to computing $\mathbf{G}^<$ is not straightforward. In particular (2.2) and (2.3) do not extend to this case. In this dissertation, we provide a new way of deriving (2.4), (2.5) and (2.6). This derivation is then extended to $\mathbf{G}^<$ for which similar relations will be derived. These recurrences are most efficient when the sparsity graph of $\mathbf{\Gamma}$ is a subset of the graph of $\mathbf{A}$, i.e., $\boldsymbol{\gamma}_{ij} \neq 0 \;\Rightarrow\; \mathbf{a}_{ij} \neq 0$.

Note that for the purpose of computer implementation, an important distinction must be made to distinguish what we term 1D, 2D and 3D problems. Physically, all problems are 3-dimensional. However, we can often simplify the original 3D Schrödinger equation as a 2D equation. In addition, if the mesh or device is elongated in one direction, say y, then the mesh can be split into "slices" along the y direction. This gives rise to a matrix $\mathbf{A}$ with block tri-diagonal structure. The problem is then treated as 1-dimensional. Such treatment (e.g., in Erisman and Tinney's method [Erisman and Tinney, 1975] and RGF [Svizhenko et al., 2002]), however, does not achieve the best efficiency because it does not exploit the sparsity within each block, which itself has a tri-diagonal structure. In contrast, our FIND-2way method takes advantage of the sparsity inside each block and achieves better efficiency through the numbering schemes used by nested dissection.

Chapter 3

Computing the Inverse

FIND improves on RGF by reducing the computational cost to $\mathcal{O}(N_x^2 N_y)$ and the memory to $\mathcal{O}(N_x N_y \log N_x)$. FIND follows some of the ideas of the nested dissection algorithm [George, 1973]. 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 $\mathbf{A}$. Details are described in Section 3.1.

In this chapter, we focus on the calculation of the diagonal of $\mathbf{G}^{\mathrm{r}}$ and will reserve the extension to the diagonal entries of $\mathbf{G}^{<}$ and certain off-diagonal entries of $\mathbf{G}^{\mathrm{r}}$ for Chapter 4.

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 $\mathbf{A}$ 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.

3.1 Brief description of the algorithm

The non-zero entries of a matrix $\mathbf{A}$ can be represented using a graph in which each node corresponds to a row or column of the matrix. If an entry $\mathbf{A}_{ij}$ 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. 3.5). For each cluster, we can define three important sets:

Boundary set B This is the set of all mesh nodes in the cluster which have a connection with a node outside the set.

Inner set I 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 cluster. Such set is equivalent to the boundary set of the complement of the cluster.

These sets are illustrated in Fig. 3.1 where each node is connected to its nearest neighbor as in a 5-point stencil. This can be generalized to more complex connectivities.

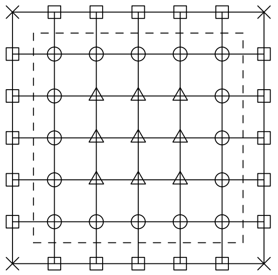


Figure 3.1: Cluster and definition of the sets. The cluster is composed of all the mesh nodes inside the dash line. The triangles form the inner set, the circles the boundary set and the squares the adjacent set. The crosses are mesh nodes outside the cluster that are not connected to the mesh nodes in the cluster.

3.1.1 Upward pass

The first stage of the algorithm, or upward pass, consists of 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. We then keep eliminating the remaining inner mesh nodes level by level in this way. This recursive process is shown in Fig. 3.2.

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

$$\mathsf{M} = [a_{11}\ a_{12}\ \ldots\ a_{1n};\ a_{21}\ a_{22}\ \ldots\ a_{2n};\ \ldots],$$

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

We now define the notation $\mathbf{U}_{\mathsf{C}}$. Assume we eliminate all the inner mesh nodes of cluster C from matrix $\mathbf{A}$. Denote the resulting matrix $\mathbf{A}_{\mathsf{C}+}$ (the notation $\mathsf{C}+$ has a special meaning described in more details in the proof of correctness of the algorithm) and B_{sC} the boundary set of C. Then

$$\mathbf{U}_{\mathsf{C}} = \mathbf{A}_{\mathsf{C}+}(\mathsf{B}_{\mathsf{C}}, \mathsf{B}_{\mathsf{C}}).$$

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

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

The process of elimination of the first p columns in this matrix consists of computing

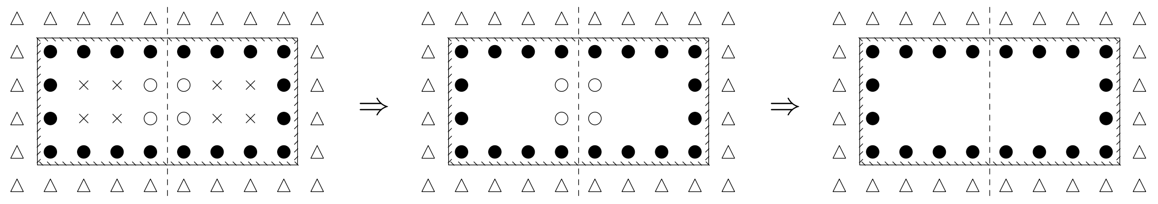


Figure 3.2: The top figure is a 4×8 cluster with two 4×4 child clusters separated by the dash line. The middle figure shows the result of eliminating the inner mesh nodes in the child clusters. The bottom figure shows the result of eliminating the inner mesh nodes in the 4×8 cluster. The elimination from the middle row can be re-used to obtain the elimination at the bottom.

an "updated block D" (denoted $\tilde{D}$) given by the formula

$$\tilde{D} = D - CA^{-1}B.$$

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

The pseudo-code for procedure `eliminateInnerNodes` implements this elimination procedure (see page 27).

3.1.2 Downward pass

The second stage of the algorithm, or downward pass, consists of removing all the mesh nodes that are outside a leaf cluster. This stage re-uses the elimination computed during the first stage. Denote by C_1 and C_2 the two children of the root cluster (which is the entire mesh). Denote by 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 that are outside C_1 and do not belong to its adjacent set, i.e., the inner mesh nodes of $\bar{\mathsf{C}}_1$. This is illustrated in Fig. 3.3.

The process then continues in a similar fashion down to the leaf clusters. A typical situation is depicted in Fig. 3.4. Once we have eliminated all the inner mesh nodes of $\bar{\mathsf{C}}$, we proceed to its children C_1 and C_2. Take C_1 for example. To remove all the

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 $\mathbf{A}$.
Input: cluster $\mathsf{C}$ with $n$ boundary mesh nodes.
Output: all the inner mesh nodes of cluster $\mathsf{C}$ are eliminated by the procedure.
        The $n \times n$ matrix $\mathbf{U}_\mathsf{C}$ with the result of the elimination is saved.
1  if $\mathsf{C}$ is not a leaf then
2      $\mathsf{C1}$ = left child of $\mathsf{C}$ ;
3      $\mathsf{C2}$ = right child of $\mathsf{C}$ ;
4      eliminateInnerNodes($\mathsf{C1}$)    /* The boundary set is denoted $\mathsf{B}_{\mathsf{C1}}$ */ ;
5      eliminateInnerNodes($\mathsf{C2}$)    /* The boundary set is denoted $\mathsf{B}_{\mathsf{C2}}$ */ ;
6      $\mathbf{A}_\mathsf{C} = [\mathbf{U}_{\mathsf{C1}}\ \mathbf{A}(\mathsf{B}_{\mathsf{C1}}, \mathsf{B}_{\mathsf{C2}});\ \mathbf{A}(\mathsf{B}_{\mathsf{C2}}, \mathsf{B}_{\mathsf{C1}})\ \mathbf{U}_{\mathsf{C2}}]$;
       /* $\mathbf{A}(\mathsf{B}_{\mathsf{C1}}, \mathsf{B}_{\mathsf{C2}})$ and $\mathbf{A}(\mathsf{B}_{\mathsf{C2}}, \mathsf{B}_{\mathsf{C1}})$ are values from the original
          matrix $\mathbf{A}$.                                                    */
7  else
8      $\mathbf{A}_\mathsf{C} = \mathbf{A}(\mathsf{C}, \mathsf{C})$ ;
9  if $\mathsf{C}$ is not the root then
10     Eliminate from $\mathbf{A}_\mathsf{C}$ the mesh nodes that are inner nodes of $\mathsf{C}$ ;
11     Save the resulting matrix $\mathbf{U}_\mathsf{C}$;
```

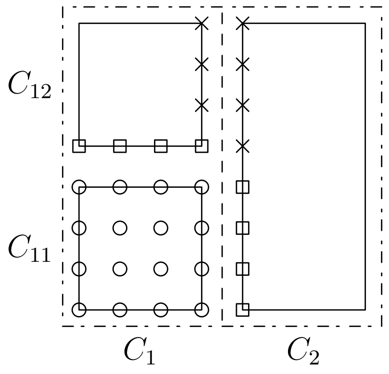


Figure 3.3: The first step in the downward pass of the algorithm. Cluster C_1 is on the left and cluster C_2 on the right. The circles are mesh nodes in cluster C_{11}. The mesh nodes in the adjacent set of C_{11} are denoted by squares; they are not eliminated at this step. The crosses are mesh nodes that are in the boundary set of either 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 that 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}.

inner mesh nodes of $\bar{\mathsf{C}}_1$, similar to the elimination in the upward pass, we simply need to remove some nodes in the boundary sets of $\bar{\mathsf{C}}$ and C_2 because $\bar{\mathsf{C}}_1 = \bar{\mathsf{C}} \cup \mathsf{C}_2$. The complete algorithm is given in procedure `eliminateOuterNodes`.

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 $\mathbf{A}$; the upward pass
      [eliminateInnerNodes()] should have been completed.
Input: cluster $\mathsf{C}$ with $n$ adjacent mesh nodes.
Output: all the inner mesh nodes of cluster $\bar{\mathsf{C}}$ are eliminated by the procedure.
        The $n \times n$ matrix $\mathbf{U}_{\bar{\mathsf{C}}}$ with the result of the elimination is saved.
1  if $\mathsf{C}$ is not the root then
2      $\mathsf{D}$ = parent of $\mathsf{C}$        /* The boundary set of $\bar{\mathsf{D}}$ is denoted $\mathsf{B}_{\bar{\mathsf{D}}}$ */;
3      $\mathsf{D1}$ = sibling of $\mathsf{C}$      /* The boundary set of $\mathsf{D1}$ is denoted $\mathsf{B}_{\mathsf{D1}}$ */;
4      $\mathbf{A}_{\bar{\mathsf{C}}} = [\mathbf{U}_{\bar{\mathsf{D}}}\ \mathbf{A}(\mathsf{B}_{\bar{\mathsf{D}}}, \mathsf{B}_{\mathsf{D1}});\ \mathbf{A}(\mathsf{B}_{\mathsf{D1}}, \mathsf{B}_{\bar{\mathsf{D}}})\ \mathbf{U}_{\mathsf{D1}}]$;
       /* $\mathbf{A}(\mathsf{B}_{\bar{\mathsf{D}}}, \mathsf{B}_{\mathsf{D1}})$ and $\mathbf{A}(\mathsf{B}_{\mathsf{D1}}, \mathsf{B}_{\bar{\mathsf{D}}})$ are values from the original matrix
          $\mathbf{A}$.                                                        */
       /* If $\mathsf{D}$ is the root, then $\bar{\mathsf{D}} = \emptyset$ and $\mathbf{A}_{\bar{\mathsf{C}}} = \mathbf{U}_{\mathsf{D1}}$.            */
5      Eliminate from $\mathbf{A}_{\bar{\mathsf{C}}}$ the mesh nodes which are inner nodes of $\bar{\mathsf{C}}$; $\mathbf{U}_{\bar{\mathsf{C}}}$ is the
       resulting matrix;
6      Save $\mathbf{U}_{\bar{\mathsf{C}}}$;
7  if $\mathsf{C}$ is not a leaf then
8      $\mathsf{C1}$ = left child of $\mathsf{C}$ ;
9      $\mathsf{C2}$ = right child of $\mathsf{C}$ ;
10     eliminateOuterNodes($\mathsf{C1}$);
11     eliminateOuterNodes($\mathsf{C2}$);
12 else
13     Calculate $[\mathbf{A}^{-1}](\mathsf{C}, \mathsf{C})$ using (6.8);
```

For completeness we give the list of subroutines to call to perform the entire calculation:

```
1 treeBuild($\mathsf{M}$)                 /* Not described in this dissertation */;
2 eliminateInnerNodes(root);
3 eliminateOuterNodes(root);
```

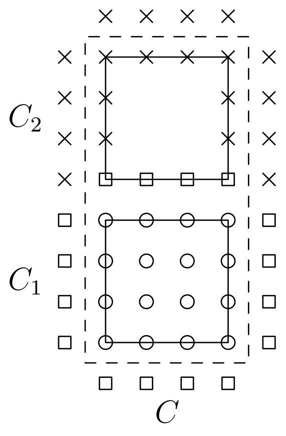


Figure 3.4: A further step in the downward pass of the algorithm. Cluster C has two children C_1 and C_2. As previously, the circles are mesh nodes in cluster C_1. The mesh nodes in the adjacent set of C_1 are denoted by squares; they are not eliminated at this step. The crosses are nodes that need to be eliminated. They belong to either the adjacent set of C or the boundary set of C_2.

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 $\mathbf{U}_{\bar{\mathsf{C}}}$. Denote by $\mathbf{A}_{\bar{\mathsf{C}}+}$ the matrix obtained by eliminating all the inner mesh nodes of $\bar{\mathsf{C}}$ (all the nodes except the squares and circles in Fig. 3.4); $\mathbf{A}_{\bar{\mathsf{C}}+}$ contains mesh nodes in the adjacent set of C (i.e., the boundary set of $\bar{\mathsf{C}}$) and the mesh nodes of C:

$$\mathbf{A}_{\bar{\mathsf{C}}+} = \begin{pmatrix} \mathbf{U}_{\bar{\mathsf{C}}} & \mathbf{A}(\mathsf{B}_{\bar{\mathsf{C}}}, \mathsf{C}) \\ \mathbf{A}(\mathsf{C}, \mathsf{B}_{\bar{\mathsf{C}}}) & \mathbf{A}(\mathsf{C}, \mathsf{C}) \end{pmatrix}.$$

The diagonal block of $\mathbf{A}^{-1}$ corresponding to cluster C is then given by

$$[\mathbf{A}^{-1}](\mathsf{C}, \mathsf{C}) = \left[\mathbf{A}(\mathsf{C}, \mathsf{C}) - \mathbf{A}(\mathsf{C}, \mathsf{B}_{\bar{\mathsf{C}}})\,(\mathbf{U}_{\bar{\mathsf{C}}})^{-1}\,\mathbf{A}(\mathsf{B}_{\bar{\mathsf{C}}}, \mathsf{C})\right]^{-1}. \tag{3.1}$$

3.1.3 Nested dissection algorithm of A. George et al.

The algorithm in this chapter uses a nested dissection-type approach similar to the nested dissection algorithm of George et al. [George, 1973]. 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 subroutine `eliminateInnerNodes`, the Gaussian elimination for cluster C reuses the result of the elimination for $\mathsf{C}1$ and $\mathsf{C}2$ by using as input $\mathbf{U}_{\mathsf{C}1}$ and $\mathbf{U}_{\mathsf{C}2}$. See Procedure 3.1, page 27. The matrix $\mathbf{U}_{\mathsf{C}}$ produced by the elimination is then saved. Note that Procedure `eliminateInnerNodes` could be used, with small modifications, to implement George's algorithm.

The second subroutine `eliminateOuterNodes` reuses results from both of the subroutines `eliminateInnerNodes` and `eliminateOuterNodes`. See Procedure 3.2, page 28. The matrices $\mathbf{U}_{\bar{\mathsf{D}}}$ (produced by `eliminateOuterNodes`) and $\mathbf{U}_{\mathsf{D}1}$ (produced by `eliminateInnerNodes`) are being reused. The matrix obtained after elimination, $\mathbf{U}_{\bar{\mathsf{C}}}$, 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" [George, 1973] with double the width. This is required for reusing the partial elimination results, in particular during the downward pass.

3.2 Detailed description of the algorithm

In this section, we describe the algorithm in detail and derive a formal proof of the correctness of the algorithm. The proof relies primarily on the properties of 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 that essentially contains the original tree and in addition a tree associated with the complement of each cluster ($\overline{\mathsf{C}}$ in our notation). The reason 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 (perhaps making things more complicated), from a theoretical and formal standpoint, this is actually a natural graph to consider.

3.2.1 The definition and properties of mesh node sets and trees

We start this section by defining M as the set of all 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 a tree by $\mathsf{T}_0 = \{\mathsf{C}_i\}$. The subsets C_i are defined recursively in the following way: Let $\mathsf{C}_1 = \mathsf{M}$, 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 $\mathsf{C}_i \cup \mathsf{C}_j = \mathsf{C}_k$ and $\mathsf{C}_i \cap \mathsf{C}_j = \emptyset$, i.e., $\{\mathsf{C}_i, \mathsf{C}_j\}$ is a partition of C_k. Fig. 3.5 shows the partitioning of the mesh and the binary tree T_0, where for notational simplicity we use the subscripts of the clusters to stand for the clusters.

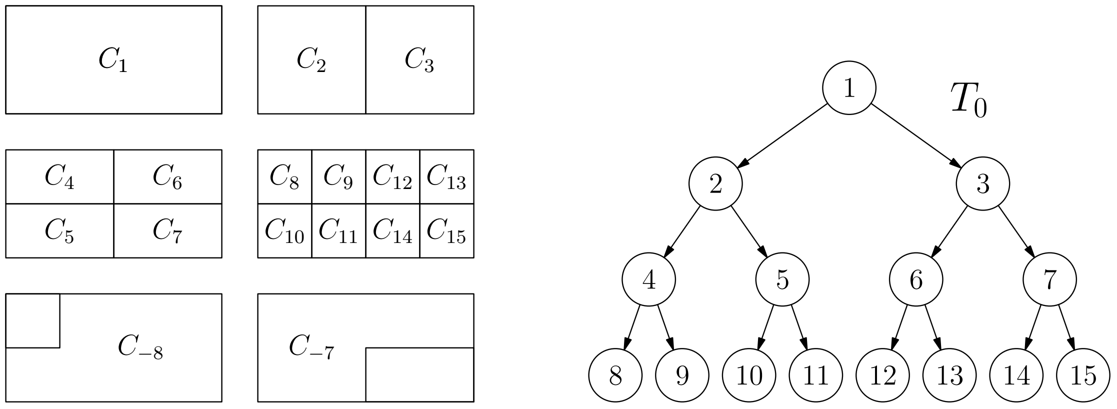


Figure 3.5: The mesh and its partitions. $\mathsf{C}_1 = \mathsf{M}$.

Let $\mathsf{C}_{-i} = \bar{\mathsf{C}}_i = \mathsf{M}\backslash\mathsf{C}_i$. Now we can define an *augmented tree* T_r^+ with respect to a leaf node $\mathsf{C}_r \in \mathsf{T}_0$ as $\mathsf{T}_r^+ = \{\mathsf{C}_j|\mathsf{C}_r \subseteq \mathsf{C}_{-j}, j < -3\} \cup (\mathsf{T}_0\backslash\{\mathsf{C}_j|\mathsf{C}_r \subseteq \mathsf{C}_j, j > 0\})$. Such an 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 $\mathsf{C}_i \cup \mathsf{C}_j = \mathsf{C}_k$ and $\mathsf{C}_i \cap \mathsf{C}_j = \emptyset$. In addition, since $\mathsf{C}_2 = \mathsf{C}_{-3}$, $\mathsf{C}_3 = \mathsf{C}_{-2}$ and $\mathsf{C}_{-1} = \emptyset$, the tree nodes $\mathsf{C}_{\pm 1}$, C_{-2}, and C_{-3} are removed from T_r^+ to avoid redundancy. Two examples of such augmented tree are shown in Fig. 3.6.

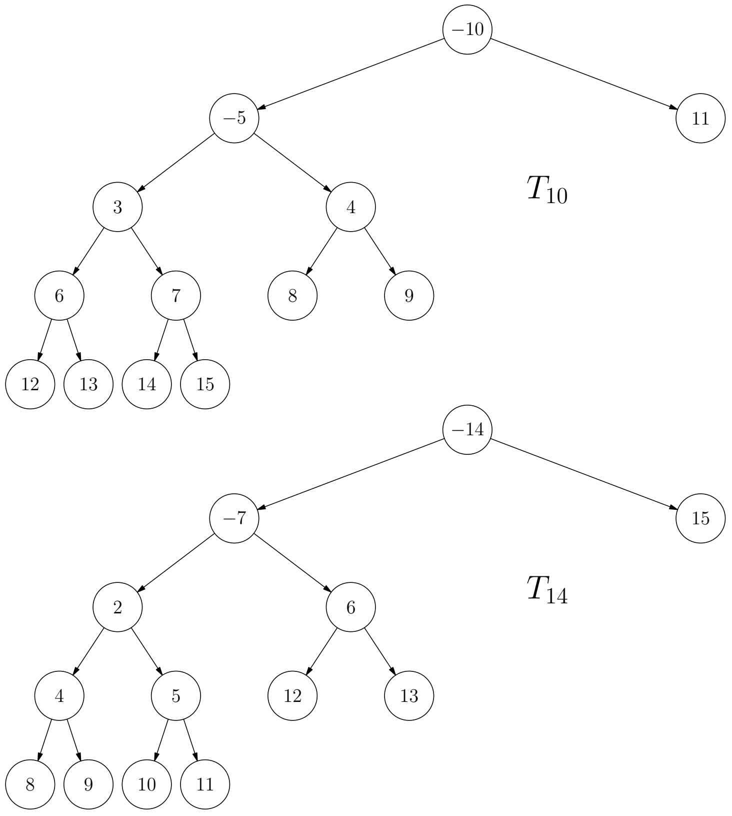


Figure 3.6: Examples of augmented trees.

We denote by I_i the inner nodes of cluster C_i and by B_i the boundary nodes as defined in section 5. Then we recursively define the set of *private inner nodes* of C_i as $\mathsf{S}_i = \mathsf{I}_i\backslash \cup_{\mathsf{C}_j \subset \mathsf{C}_i} \mathsf{S}_j$ with $\mathsf{S}_i = \mathsf{I}_i$ if C_i is a leaf node in T_r^+, where C_i and $\mathsf{C}_j \in \mathsf{T}_r^+$. Fig. 3.7 shows these subsets for a 4×8 cluster.

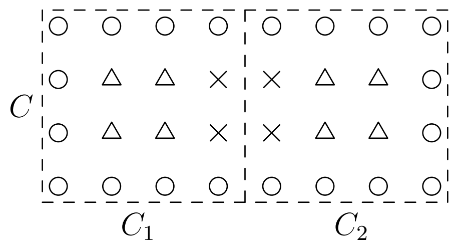


Figure 3.7: Cluster C has two children C_1 and C_2. The inner nodes of cluster C_1 and C_2 are shown using triangles. The private inner nodes of C are shown with crosses. The boundary set of C is shown using circles.

Now 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 B.1.

Property 3.1 shows two different ways of looking at the same subset. This change of view happens repeatedly in our algorithm.

Property 3.1 *If C_i and C_j are the two children of C_k, then $\mathsf{S}_k \cup \mathsf{B}_k = \mathsf{B}_i \cup \mathsf{B}_j$ and $\mathsf{S}_k = (\mathsf{B}_i \cup \mathsf{B}_j)\backslash\mathsf{B}_k$.*

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 a property makes it possible to define a consistent ordering.

Property 3.2 *For any given augmented tree T_r^+ and all $\mathsf{C}_i \in \mathsf{T}_r^+$, S_i, B_{-r}, and C_r are all disjoint and $\mathsf{M} = (\cup_{\mathsf{C}_i \in \mathsf{T}_r^+} \mathsf{S}_i) \cup \mathsf{B}_{-r} \cup \mathsf{C}_r$.*

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

1. The indices of nodes in S_i are contiguous;
2. $\mathsf{C}_i \subset \mathsf{C}_j$ implies $\mathsf{S}_i < \mathsf{S}_j$; and
3. The indices corresponding to B_{-r} and C_r appear at the end.

Since it is possible that $\mathsf{C}_i \not\subset \mathsf{C}_j$ and $\mathsf{C}_j \not\subset \mathsf{C}_i$, we can see that the consistent ordering of $\mathbf{A}$ 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 $\mathsf{S}_j < \mathsf{S}_k < \mathsf{S}_i$ and $\mathsf{S}_k < \mathsf{S}_j < S_i$ are consistent. When we discuss the properties of the Gaussian elimination of $\mathbf{A}$ 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.

3.2.2 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 $\mathbf{A}$ with consistent ordering.

Notation: For a given $\mathbf{A}$, the order of S_i is determined so we can write the indices of S_i as $i_1, i_2, \ldots$, etc. For notational convenience, we write $\cup_{\mathsf{S}_j<\mathsf{S}_i}\mathsf{S}_j$ as $\mathsf{S}_{<i}$ and $(\cup_{\mathsf{S}_i<\mathsf{S}_j}\mathsf{S}_j) \cup \mathsf{B}_{-r} \cup \mathsf{C}_r$ as $\mathsf{S}_{>i}$. If $g = i_j$ then we denote $\mathsf{S}_{i_{j+1}}$ 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 $\mathbf{A}$ 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 $\mathbf{A}^{-1}$. Starting from $\mathbf{A}_{i_1} = \mathbf{A}$, we define the following intermediate matrices for each $g = i_1, i_2, \ldots, -r$ as the results of each step of Gaussian elimination:

$$\mathbf{A}_{g+} = \text{ the result of eliminating the } \mathsf{S}_g \text{ columns in } \mathbf{A}_g.$$

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

Example. Let us consider Figure 3.6 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 $i_1 = 12$, $i_2 = 13$, $i_3 = 14$, $\ldots$. Pick $i = 15$, then $\mathsf{S}_{<i} = \mathsf{S}_{12} \cup \mathsf{S}_{13} \cup \mathsf{S}_{14}$. Pick $i = -5$, then $\mathsf{S}_{>i} = \mathsf{S}_{11} \cup \mathsf{S}_{-10} \cup \mathsf{B}_{-10} \cup \mathsf{C}_{10}$. For $g = i_j = 15$, $\mathsf{S}_{g+} = \mathsf{S}_{6}$.

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 $(\mathsf{B}_i, \mathsf{B}_i)$ entries. The precise statement of the theorem is listed in the appendix as Theorem B.1 with its proof.

The following two matrices show one step of the elimination, with the right pattern of 0's:

$$\mathbf{A}_i = \begin{pmatrix} \mathbf{A}_i(\mathsf{S}_{<i}, \mathsf{S}_{<i}) & \mathbf{A}_i(\mathsf{S}_{<i}, \mathsf{S}_i) & \mathbf{A}_i(\mathsf{S}_{<i}, \mathsf{B}_i) & \mathbf{A}_i(\mathsf{S}_{<i}, \mathsf{S}_{>i} \backslash \mathsf{B}_i) \\ \mathbf{0} & \mathbf{A}_i(\mathsf{S}_i, \mathsf{S}_i) & \mathbf{A}_i(\mathsf{S}_i, \mathsf{B}_i) & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_i(\mathsf{B}_i, \mathsf{S}_i) & \mathbf{A}_i(\mathsf{B}_i, \mathsf{B}_i) & \mathbf{A}_i(\mathsf{B}_i, \mathsf{S}_{>i} \backslash \mathsf{B}_i) \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_i(S_{>i} \backslash \mathsf{B}_i, \mathsf{B}_i) & \mathbf{A}_i(\mathsf{S}_{>i} \backslash \mathsf{B}_i, S_{>i} \backslash \mathsf{B}_i) \end{pmatrix}$$

$\Rightarrow$

$$\mathbf{A}_{i+} = \begin{pmatrix} \mathbf{A}_i(\mathsf{S}_{<i}, \mathsf{S}_{<i}) & \mathbf{A}_i(\mathsf{S}_{<i}, \mathsf{S}_i) & \mathbf{A}_i(\mathsf{S}_{<i}, \mathsf{B}_i) & \mathbf{A}_i(\mathsf{S}_{<i}, \mathsf{S}_{>i} \backslash \mathsf{B}_i) \\ \mathbf{0} & \mathbf{A}_i(\mathsf{S}_i, \mathsf{S}_i) & A_i(\mathsf{S}_i, \mathsf{B}_i) & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_{i+}(\mathsf{B}_i, \mathsf{B}_i) & \mathbf{A}_i(\mathsf{B}_i, \mathsf{S}_{>i} \backslash \mathsf{B}_i) \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_i(\mathsf{S}_{>i} \backslash \mathsf{B}_i, \mathsf{B}_i) & \mathbf{A}_i(\mathsf{S}_{>i} \backslash \mathsf{B}_i, \mathsf{S}_{>i} \backslash \mathsf{B}_i) \end{pmatrix}$$

where $\mathbf{A}_{i+}(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{A}_i(\mathsf{B}_i, \mathsf{B}_i) - \mathbf{A}_i(\mathsf{B}_i, \mathsf{S}_i)\mathbf{A}_i(\mathsf{S}_i, \mathsf{S}_i)^{-1}\mathbf{A}_i(\mathsf{S}_i, \mathsf{B}_i)$.

Note: in the above matrices, $\mathbf{A}_i(\bullet, \mathsf{B}_i)$ is written as a block. In reality, however, it is usually not a block for any $\mathbf{A}$ with consistent ordering.

Because the matrix sparsity pattern is preserved during the elimination process, we know that certain entries remain unchanged. More specifically, Corollaries 3.1 and 3.2 can be used to determine when the entries $(\mathsf{B}_i \cup \mathsf{B}_j, \mathsf{B}_i \cup \mathsf{B}_j)$ remain unchanged. Corollary 3.3 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 B.1.

Corollary 3.1 *If C_i and C_j are the two children of C_k, then $\mathbf{A}_k(\mathsf{B}_i, \mathsf{B}_j) = \mathbf{A}(\mathsf{B}_i, \mathsf{B}_j)$ and $\mathbf{A}_k(\mathsf{B}_j, \mathsf{B}_i) = \mathbf{A}(\mathsf{B}_j, \mathsf{B}_i)$.*

Corollary 3.2 *If C_i is a child of C_k, then $\mathbf{A}_k(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{A}_{i+}(\mathsf{B}_i, \mathsf{B}_i)$.*

Corollaries 3.1 and 3.2 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 $(\mathsf{B}_i, \mathsf{B}_j)$ and $(\mathsf{B}_j, \mathsf{B}_i)$ from the original matrix $\mathbf{A}$, and the entries $(\mathsf{B}_i, \mathsf{B}_i)$ and $(\mathsf{B}_j, \mathsf{B}_j)$ obtained after elimination of S_i and S_j.

Corollary 3.3 *If C_i is a leaf node in T_r^+, then $\mathbf{A}_i(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{A}(\mathsf{C}_i, \mathsf{C}_i)$.*

Corollary 3.3 tells us that we can use the entries from the original matrix $\mathbf{A}$ for leaf clusters (even though other mesh nodes may have already been eliminated at that point).

Based on Theorem B.1 and Corollaries 3.1–3.3, we can compare the partial elimination results of matrices with different orderings.

Theorem 3.1 *For any r and s such that $\mathsf{C}_i \in \mathsf{T}_r^+$ and $\mathsf{C}_i \in \mathsf{T}_s^+$, we have*

$$\mathbf{A}_{r,i}(\mathsf{S}_i \cup \mathsf{B}_i, \mathsf{S}_i \cup \mathsf{B}_i) = \mathbf{A}_{s,i}(\mathsf{S}_i \cup \mathsf{B}_i, \mathsf{S}_i \cup \mathsf{B}_i).$$

Proof: If C_i is a leaf node, then by Corollary 3.3, we have $\mathbf{A}_{r,i}(\mathsf{S}_i \cup \mathsf{B}_i, \mathsf{S}_i \cup \mathsf{B}_i) = \mathbf{A}_{r,i}(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{A}_r(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{A}_s(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{A}_{s,i}(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{A}_{s,i}(\mathsf{S}_i \cup \mathsf{B}_i, \mathsf{S}_i \cup \mathsf{B}_i)$.

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 B.1, we have $\mathbf{A}_{r,i+}(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{A}_{s,i+}(\mathsf{B}_i, \mathsf{B}_i)$ and $\mathbf{A}_{r,j+}(\mathsf{B}_j, \mathsf{B}_j) = \mathbf{A}_{s,j+}(\mathsf{B}_j, \mathsf{B}_j)$;
- by Corollary 3.2, we have $\mathbf{A}_{r,k}(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{A}_{r,i+}(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{A}_{s,i+}(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{A}_{s,k}(\mathsf{B}_i, \mathsf{B}_i)$ and $\mathbf{A}_{r,k}(\mathsf{B}_j, \mathsf{B}_j) = \mathbf{A}_{r,j+}(\mathsf{B}_j, \mathsf{B}_j) = \mathbf{A}_{s,j+}(\mathsf{B}_j, \mathsf{B}_j) = \mathbf{A}_{s,k}(\mathsf{B}_j, \mathsf{B}_j)$;
- by Corollary 3.1, we have $\mathbf{A}_{r,k}(\mathsf{B}_i, \mathsf{B}_j) = \mathbf{A}_r(\mathsf{B}_i, \mathsf{B}_j) = \mathbf{A}_s(\mathsf{B}_i, \mathsf{B}_j) = \mathbf{A}_{s,k}(\mathsf{B}_i, \mathsf{B}_j)$ and $\mathbf{A}_{r,k}(\mathsf{B}_j, \mathsf{B}_i) = \mathbf{A}_r(\mathsf{B}_j, \mathsf{B}_i) = \mathbf{A}_s(\mathsf{B}_j, \mathsf{B}_i) = \mathbf{A}_{s,k}(\mathsf{B}_j, \mathsf{B}_i)$.

Now we have $\mathbf{A}_{r,k}(\mathsf{B}_i \cup \mathsf{B}_j, \mathsf{B}_i \cup \mathsf{B}_j) = \mathbf{A}_{s,k}(\mathsf{B}_i \cup \mathsf{B}_j, \mathsf{B}_i \cup \mathsf{B}_j)$. By Property B.4, we have $\mathbf{A}_{r,k}(\mathsf{S}_k \cup \mathsf{B}_k, \mathsf{S}_k \cup \mathsf{B}_k) = \mathbf{A}_{s,k}(\mathsf{S}_k \cup \mathsf{B}_k, \mathsf{S}_k \cup \mathsf{B}_k)$. By mathematical induction, the theorem is proved. □

If we go one step further, based on Theorems B.1 and 3.1, we have the following corollary.

Corollary 3.4 *For any r and s such that $\mathsf{C}_i \in \mathsf{T}_r^+$ and $\mathsf{C}_i \in \mathsf{T}_s^+$, we have*

$$\mathbf{A}_{r,i+}(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{A}_{s,i+}(\mathsf{B}_i, \mathsf{B}_i).$$

Theorem 3.1 and Corollary 3.4 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.

3.2.3 The algorithm

Corollary 3.4 shows that $\mathbf{A}_{\bullet,i+}(\mathsf{B}_i, \mathsf{B}_i)$ is the same for all augmented trees, so we can have the following definition for any r:

$$\mathbf{U}_i = \mathbf{A}_{r,i+}(\mathsf{B}_i, \mathsf{B}_i).$$

By Theorem 3.1, Corollary 3.1, and Corollary 3.2, for all i, j, and k such that C_i and C_j are the two children of C_k, we have

$$\mathbf{U}_k = \mathbf{A}_{r,k}(\mathsf{B}_k, \mathsf{B}_k) - \mathbf{A}_{r,k}(\mathsf{B}_k, \mathsf{S}_k)\mathbf{A}_{r,k}(\mathsf{S}_k, \mathsf{S}_k)^{-1}\mathbf{A}_{r,k}(\mathsf{S}_k, \mathsf{B}_k), \tag{3.2}$$

where

$$\mathbf{A}_{r,k}(\mathsf{B}_k,\mathsf{B}_k) = \begin{pmatrix} \mathbf{U}_i(\mathsf{B}_k \cap \mathsf{B}_i, \mathsf{B}_k \cap \mathsf{B}_i) & \mathbf{A}(\mathsf{B}_k \cap \mathsf{B}_i, \mathsf{B}_k \cap \mathsf{B}_j) \\ \mathbf{A}(\mathsf{B}_k \cap \mathsf{B}_j, \mathsf{B}_k \cap \mathsf{B}_i) & \mathbf{U}_j(\mathsf{B}_k \cap \mathsf{B}_j, \mathsf{B}_k \cap \mathsf{B}_j) \end{pmatrix}, \tag{3.3a}$$

$$\mathbf{A}_{r,k}(\mathsf{S}_k,\mathsf{S}_k) = \begin{pmatrix} \mathbf{U}_i(\mathsf{S}_k \cap \mathsf{B}_i, \mathsf{S}_k \cap \mathsf{B}_i) & \mathbf{A}(\mathsf{S}_k \cap \mathsf{B}_i, \mathsf{S}_k \cap \mathsf{B}_j) \\ \mathbf{A}(\mathsf{S}_k \cap \mathsf{B}_j, \mathsf{S}_k \cap \mathsf{B}_i) & \mathbf{U}_j(\mathsf{S}_k \cap \mathsf{B}_j, \mathsf{S}_k \cap \mathsf{B}_j) \end{pmatrix}, \tag{3.3b}$$

$$\mathbf{A}_{r,k}(\mathsf{B}_k,\mathsf{S}_k) = \begin{pmatrix} \mathbf{U}_i(\mathsf{B}_k \cap \mathsf{B}_i, \mathsf{S}_k \cap \mathsf{B}_i) & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_j(\mathsf{B}_k \cap \mathsf{B}_j, \mathsf{S}_k \cap \mathsf{B}_j) \end{pmatrix}, \tag{3.3c}$$

$$\mathbf{A}_{r,k}(\mathsf{S}_k,\mathsf{B}_k) = \begin{pmatrix} \mathbf{U}_i(\mathsf{S}_k \cap \mathsf{B}_i, \mathsf{B}_k \cap \mathsf{B}_i) & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_j(\mathsf{S}_k \cap \mathsf{B}_j, \mathsf{B}_k \cap \mathsf{B}_j) \end{pmatrix}. \tag{3.3d}$$

If C_k is a leaf node, then by Corollary 3.3, we have $\mathbf{A}_{r,k}(\mathsf{B}_k,\mathsf{B}_k) = \mathbf{A}_r(\mathsf{B}_k,\mathsf{B}_k)$, $\mathbf{A}_{r,k}(\mathsf{S}_k,\mathsf{S}_k) = \mathbf{A}_r(\mathsf{S}_k,\mathsf{S}_k)$, $\mathbf{A}_{r,k}(\mathsf{B}_k,\mathsf{S}_k) = \mathbf{A}_r(\mathsf{B}_k,\mathsf{S}_k)$, and $\mathbf{A}_{r,k}(\mathsf{S}_k,\mathsf{B}_k) = \mathbf{A}_r(\mathsf{S}_k,\mathsf{B}_k)$.

By (3.2), we can compute $\mathbf{U}$ for upper level clusters based on i) the original matrix $\mathbf{A}$ and ii) the values of $\mathbf{U}$ for lower level clusters. The values of $\mathbf{U}$ for leaf nodes can be computed directly through Gaussian elimination. The last step of the elimination is shown below:

$$\begin{array}{c|cccc} & \mathsf{S}_{<-r} & \mathsf{S}_{-r} & \mathsf{B}_{-r} & \mathsf{C}_r \\ \mathsf{S}_{<-r} & \times & \times & \times & \mathbf{0} \\ \mathsf{S}_{-r} & \mathbf{0} & \times & \times & \mathbf{0} \\ \mathsf{B}_{-r} & \mathbf{0} & \times & \times & \times \\ \mathsf{C}_r & \mathbf{0} & \mathbf{0} & \times & \times \end{array} \quad \Rightarrow \quad \begin{array}{c|cccc} & \mathsf{S}_{<-r} & \mathsf{S}_{-r} & \mathsf{B}_{-r} & \mathsf{C}_r \\ \mathsf{S}_{<-r} & \times & \times & \times & \mathbf{0} \\ \mathsf{S}_{-r} & \mathbf{0} & \times & \times & \mathbf{0} \\ \mathsf{B}_{-r} & \mathbf{0} & \mathbf{0} & \times & \times \\ \mathsf{C}_r & \mathbf{0} & \mathbf{0} & \times & \times \end{array}$$

In (3.2), 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 and negative tree nodes separately.

Since no negative node is the descendant of any positive node in the augmented trees, we can first compute $\mathbf{U}_i$ 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 $\mathbf{U}_i$ level by level from the bottom up and call it the *upward pass*. This is done in procedure `eliminateInnerNodes` of the algorithm in Section 3.1.1.

Once all the positive tree nodes have been computed, we compute $\mathbf{U}_i$ 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 $\mathbf{U}_i$ 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 3.1.2.

3.3 Complexity analysis

We sketch a derivation of the computational complexity. A more detailed derivation is given in Section 3.3. Consider a mesh of size $N_x \times N_y$ and let $N = N_x N_y$ be the total number of mesh nodes. For simplicity, we assume $N_x = N_y$ and 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 $\mathcal{O}((2^p)^3)$, 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 $\mathcal{O}(N 2^p)$. The total cost is therefore simply $\mathcal{O}(N 2^l) = \mathcal{O}(N^{3/2})$. This is the same computational cost (in the $\mathcal{O}$ sense) as the nested dissection algorithm of George et al. [George, 1973]. It is now apparent that FIND has the same order of computational complexity as a single LU factorization.

3.3.1 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 (3.2), we have the following cost for computing $\mathbf{U}_i$:

$$T \approx \mathsf{b}_i^2\mathsf{s}_i + \mathsf{s}_i^3/3 + \mathsf{b}_i\mathsf{s}_i^2 \text{ flops}, \tag{3.4}$$

where both $\mathsf{b}_i = |\mathsf{B}_i|$ and $\mathsf{s}_i = |\mathsf{S}_i|$ are of order a for clusters of size $a \times a$. For a squared mesh, since the number of clusters in each level is inversely 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 of size either $a \times a$ 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 level L have different behaviors.

In the upward pass, as a typical case, $a \times a$ clusters merge into $a \times 2a$ clusters and then into $2a \times 2a$ 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 $\mathbf{U}_k$ is $T \leq (6^2 \times 2 + 2^3/3 + 2^2 \times 6)a^3 \leq \frac{296}{3}a^3$ 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 $\mathbf{U}_k$ is $T \leq (8^2 \times 4 + 4^3/3 + 4^2 \times 8)a^3 \leq \frac{1216}{3}a^3$ 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:

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

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 but remains the same in the second change, as seen in Fig. 3.8. The upper figure there 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 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.

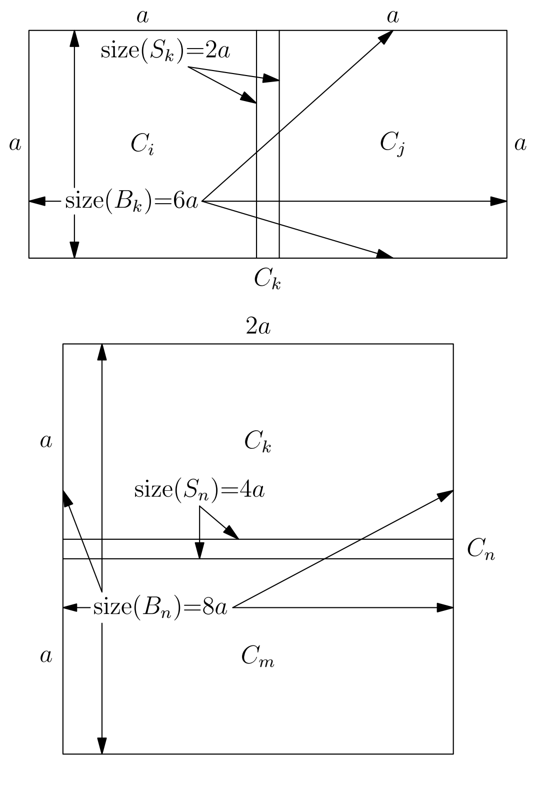


Figure 3.8: Merging clusters below level L.

For clusters above level L, the computational cost for each merging remains the same because 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. 3.9. Since for clusters above level L we have only half merges in the parent level compared to the child level, the cost decreases geometrically for levels above level L.

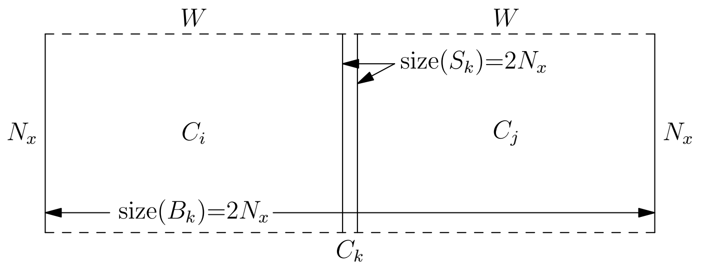


Figure 3.9: Merging rectangular clusters. Two $N_x \times W$ clusters merge into an $N_x \times 2W$ cluster.

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

$$\begin{aligned} T &\leq 151N_xN_y(a + 2a + \cdots + N_x/2) + 19N_x^3(N_y/2N_x + N_y/4N_x + \cdots + 1) \\ &\approx 151_x^2N_y + 19N_x^2N_y = 170N_x^2N_y. \end{aligned}$$

In the downward pass, similar to the upward pass, for clusters above level L, the computational cost for each partitioning remains the same because both the size of B and the size of S are $2N_x$. Since the sizes are 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. 3.10.

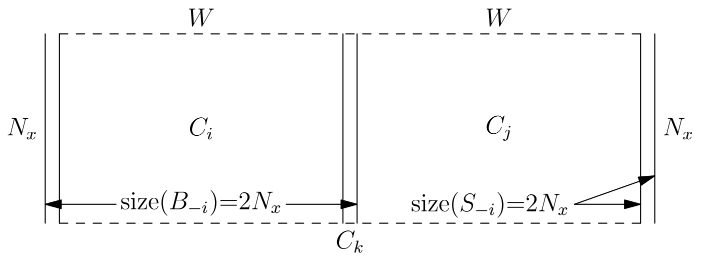


Figure 3.10: Partitioning of clusters above level L.

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 into $a \times 2a$ 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 $\mathbf{U}_k$ is $T \leq 422a \times 4a^2$. When $a \times 2a$ clusters are partitioned into $a \times a$ clusters, the size of B_k is at most $4a$ and the size of S_k is at most $6a$. The time to compute $\mathbf{U}_k$ is $T \leq 312a \times 2a^2$.

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 the sizes of B and S increase relatively smoothly, as shown in Fig. 3.11.

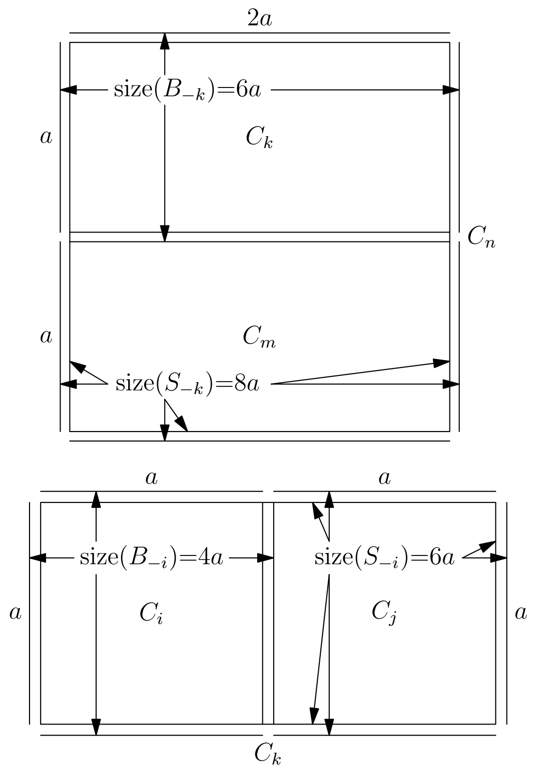


Figure 3.11: Partitioning of clusters below level L.

The computational cost in the downward pass is

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

So the total computational cost is

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

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.

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.

3.3.2 Memory cost analysis

Since the negative tree nodes are the ascendants of the positive tree nodes in the augmented trees, the downward pass needs $\mathbf{U}_i$ 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 $\mathbf{U}_i$ 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 of 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.

3.3.3 Effect of the null boundary set of the whole mesh

As discussed in Section 3.3.1, the computation cost for a cluster is proportional to the cube of its size while the number of clusters is inversely proportional to the square of its size. So the computation cost for the top level clusters dominates. Because of this, the null boundary set of the whole mesh significantly reduces the cost.

Table 3.1 shows the cost of a few top level clusters. The last two rows are the sum of the cost of the rest of the small clusters. Sum the costs for the two passes together and the total computation cost for the algorithm is roughly $147N^{3/2}$ flops with optimization discussed in the early sections.

Table 3.1: Computation cost estimate for different type of clusters

upward pass					*downward pass*				
size & cost per cluster			*clusters*		*size & cost per cluster*			*clusters*	
s	b	*cost*	*level*	*number*	s	b	*cost*	*level*	*number*
1	1	1.167	1	2	0	0	0	1	2
1	1	1.167	2	4	1	1	1.167	2	4
1/2	3/4	0.255	3	4	3/2	3/4	1.828	3	4
1/2	5/4	0.568	3	4	1/2	5/4	0.568	3	4
1/2	1	0.396	4	4	1	1/2	0.542	4	4
1/2	3/4	0.255	4	8	1/2	3/4	0.255	4	4
1/2	1/2	0.146	4	4	3/2	3/4	1.828	4	4
1/4	3/4	0.096	5	8	1	1	1.167	4	4
1/4	5/8	0.071	5	20	1	3/4	0.823	5	32
1/4	3/8	0.032	5	4	3/4	1/2	0.305	6	64
1/4	1/2	0.049	6	64	1	3/4	0.823	7...	<32
1/8	3/8	0.012	7	128	3/4	1/2	0.305	8...	<64

3.4 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. 1.1 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 equal to 1 dec/nm. The doping of the S/D regions equals 1×10^{20} cm^{-3}. According to the ITRS road map [SIA, 2001], 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 the gate metal work function, ϕ_G, directly controls the off-current. The doping gradient, g, affects both on-current and off-current.

Fig. 3.12 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).

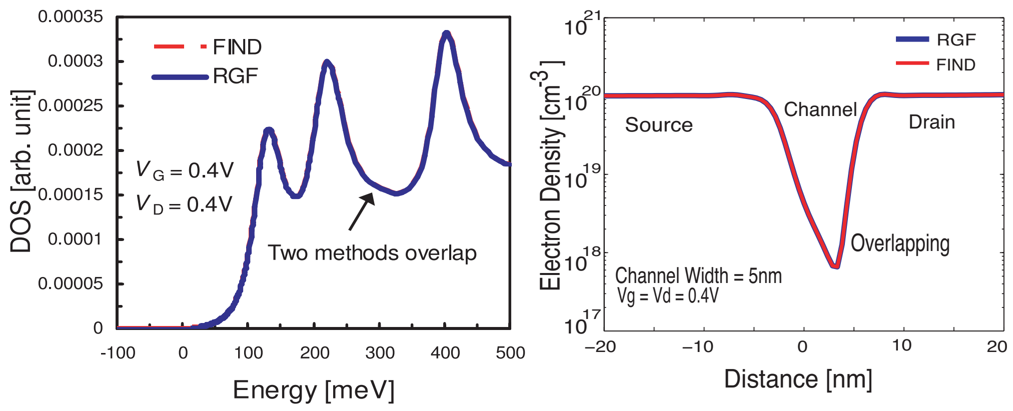


Figure 3.12: Density-of-states (DOS) and electron density plots from RGF and FIND.

Fig. 3.13 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.

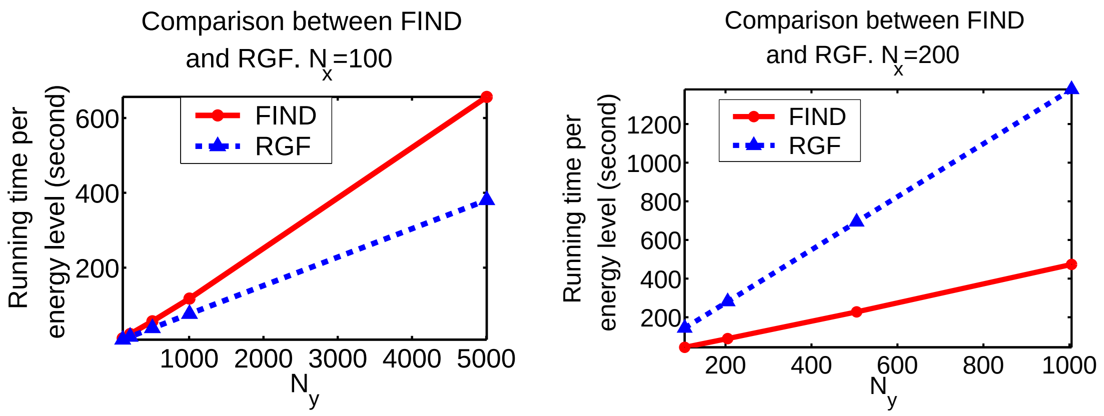


Figure 3.13: Comparison of the running time of FIND and RGF when N_x is fixed.

Fig. 3.14 show the comparisons of running times between FIND and RGF when N_y's are fixed. We can see clearly the speedup of FIND in the figure when N_x increases.

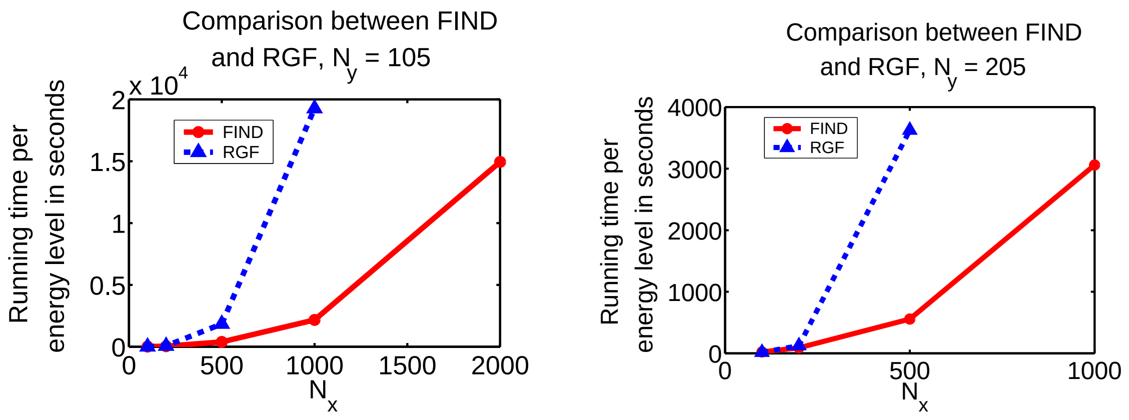


Figure 3.14: Comparison of the running time of FIND and RGF when N_y is fixed.

3.5 Concluding remarks

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 $\mathbf{A}^{-1}$, where $\mathbf{A}$ is a sparse matrix.

In this chapter, we described the algorithm and proved its correctness. We analyzed its computational and memory costs in detail. 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 $\mathcal{O}(N_x^2 N_y)$, which improves over the result of RGF [Svizhenko et al., 2002], which scales as $\mathcal{O}(N_x^3 N_y)$. 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 has been incorporated in an open source code, nanoFET [Anantram et al., 2007], which can be found on the nanohub web portal (`http://www.nanohub.org`).

Chapter 4

Extension of FIND

In addition to computing the diagonal entries of the matrix inverse, the algorithm can be extended to computing the diagonal entries and certain off-diagonal entries of $\mathbf{G}^< = \mathbf{A}^{-1}\mathbf{\Sigma}\mathbf{A}^{-\dagger}$ for the electron and current density in our transport problem.

4.1 Computing diagonal entries of another Green's function: $\mathbf{G}^<$

Intuitively, $\mathbf{A}^{-1}\mathbf{\Sigma}\mathbf{A}^{-\dagger}$ can be computed in the same way as $\mathbf{A}^{-1} = \mathcal{U}^{-1}\mathcal{L}^{-1}$ because we have

$$\begin{aligned} \mathbf{G}^< &= \mathbf{A}^{-1}\mathbf{\Sigma}\mathbf{A}^{-\dagger} \\ \Rightarrow \mathbf{A}\mathbf{G}^<\mathbf{A}^\dagger &= \mathbf{\Sigma} \\ \Rightarrow \mathcal{U}\mathbf{G}^<\mathcal{U}^\dagger &= \mathcal{L}^{-1}\mathbf{\Sigma}\mathcal{L}^{-\dagger} \\ \Rightarrow [\mathbf{G}^<]_{nn} &= (\mathcal{U}_{nn})^{-1}(\mathcal{L}^{-1}\mathbf{\Sigma}\mathcal{L}^{-\dagger})_{nn}(\mathcal{U}_{nn})^{-\dagger}. \end{aligned}$$

It remains to show how to compute the last block of $\mathcal{L}^{-1}\mathbf{\Sigma}\mathcal{L}^{-\dagger}$. We can see from the following updates that when we operate on the matrix $\mathbf{\Sigma}$, in the same way as we do on matrix $\mathbf{A}$, the pattern of $\mathbf{\Sigma}$ will be similar to that of $\mathbf{A}$ during the updates. Similar to the notation and definitions in Chapter 3, starting from $\mathbf{\Sigma}_{i_1} = \mathbf{\Sigma}$ and letting $\mathcal{L}_g^{-1}$

be the matrix corresponding to the g^{th} step of elimination such that $\mathbf{A}_{g+} = \mathcal{L}_g^{-1}\mathbf{A}_g$, we define $\mathbf{\Sigma}_{g+} = \mathcal{L}_g^{-1}\mathbf{\Sigma}_g\mathcal{L}_g^{-\dagger}$. We also write the only non-zero non-diagonal block in $\mathcal{L}_g^{-1}$, $[\mathcal{L}_g^{-1}](\mathsf{B}_g, \mathsf{S}_g)$, as $\mathbf{L}_g^{-1}$ for notation simplicity. Now,

$$\mathbf{\Sigma}_g = \begin{pmatrix} \mathbf{\Sigma}_g(\mathsf{S}_{<g}, \mathsf{S}_{<g}) & \mathbf{\Sigma}_g(\mathsf{S}_{<g}, \mathsf{S}_g) & \underline{\mathbf{\Sigma}_g(\mathsf{S}_{<g}, \mathsf{B}_g)} & \mathbf{\Sigma}_g(\mathsf{S}_{<g}, \mathsf{S}_{>g}\backslash\mathsf{B}_g) \\ \mathbf{\Sigma}_g(\mathsf{S}_g, \mathsf{S}_{<g}) & \mathbf{\Sigma}_g(\mathsf{S}_g, \mathsf{S}_g) & \underline{\mathbf{\Sigma}_g(\mathsf{S}_g, \mathsf{B}_g)} & \mathbf{0} \\ \underline{\mathbf{\Sigma}_g(\mathsf{B}_g, \mathsf{S}_{<g})} & \underline{\mathbf{\Sigma}_g(\mathsf{B}_g, \mathsf{S}_g)} & \underline{\mathbf{\Sigma}_g(\mathsf{B}_g, \mathsf{B}_g)} & \mathbf{\Sigma}_g(\mathsf{B}_g, \mathsf{S}_{>g}\backslash\mathsf{B}_g) \\ \mathbf{\Sigma}_g(\mathsf{S}_{>g}\backslash\mathsf{B}_g) & \mathbf{0} & \mathbf{\Sigma}_g(\mathsf{S}_{>g}\backslash\mathsf{B}_g, \mathsf{B}_g) & \mathbf{\Sigma}_g(\mathsf{S}_{>g}\backslash\mathsf{B}_g, \mathsf{S}_{>g}\backslash\mathsf{B}_g) \end{pmatrix}$$

$\Rightarrow$

$$\mathbf{\Sigma}_{g+} = \begin{pmatrix} \mathbf{\Sigma}_g(\mathsf{S}_{<g}, \mathsf{S}_{<g}) & \mathbf{\Sigma}_g(\mathsf{S}_{<g}, \mathsf{S}_g) & \underline{\mathbf{\Sigma}_{g+}(\mathsf{S}_{<g}, \mathsf{B}_g)} & \mathbf{\Sigma}_g(\mathsf{S}_{<g}, \mathsf{S}_{>g}\backslash\mathsf{B}_g) \\ \mathbf{\Sigma}_g(\mathsf{S}_g, \mathsf{S}_{<g}) & \mathbf{\Sigma}_g(\mathsf{S}_g, \mathsf{S}_g) & \underline{\mathbf{\Sigma}_{g+}(\mathsf{S}_g, \mathsf{B}_g)} & \mathbf{0} \\ \underline{\mathbf{\Sigma}_{g+}(\mathsf{B}_g, \mathsf{S}_{<g})} & \underline{\mathbf{\Sigma}_{g+}(\mathsf{B}_g, \mathsf{S}_g)} & \underline{\mathbf{\Sigma}_{g+}(\mathsf{B}_g, \mathsf{B}_g)} & \mathbf{\Sigma}_g(\mathsf{B}_g, \mathsf{S}_{>g}\backslash\mathsf{B}_g) \\ \mathbf{\Sigma}_g(\mathsf{S}_{>g}\backslash\mathsf{B}_g) & \mathbf{0} & \mathbf{\Sigma}_g(\mathsf{S}_{>g}\backslash\mathsf{B}_g, \mathsf{B}_g) & \mathbf{\Sigma}_g(\mathsf{S}_{>g}\backslash\mathsf{B}_g, \mathsf{S}_{>g}\backslash\mathsf{B}_g) \end{pmatrix},$$

where

$$\mathbf{\Sigma}_{g+}(\mathsf{B}_g, \mathsf{B}_g) = \mathbf{\Sigma}_g(\mathsf{B}_g, \mathsf{B}_g) + \mathbf{L}_g^{-1}\mathbf{\Sigma}_g(\mathsf{S}_g, \mathsf{B}_g) + \mathbf{\Sigma}_g(\mathsf{B}_g, \mathsf{S}_g)\mathbf{L}_g^{-\dagger} + \mathbf{L}_g^{-1}\mathbf{\Sigma}_g(\mathsf{S}_g, \mathsf{S}_g)\mathbf{L}_g^{-\dagger}.$$

The difference between the updates on $\mathbf{A}$ and the updates on $\mathbf{\Sigma}$ is that some entries in $\mathbf{\Sigma}_g(\mathsf{I}_g, \mathsf{B}_g)$ and $\mathbf{\Sigma}_g(\mathsf{B}_g, \mathsf{I}_g)$ (indicated by $\mathbf{\Sigma}_g(\mathsf{S}_{<g}, \mathsf{B}_g)$, $\mathbf{\Sigma}_g(\mathsf{S}_g, \mathsf{B}_g)$, $\mathbf{\Sigma}_g(\mathsf{B}_g, \mathsf{S}_{<g})$, and $\mathbf{\Sigma}_g(\mathsf{B}_g, \mathsf{S}_g)$ above) will be changed when we update the columns corresponding to S_g due to the right multiplication of $\mathcal{L}_g^{-\dagger}$. However, such changes do not affect the result for the final $\mathbf{\Sigma}_{NN}$ because the next update step is independent of the values of $\mathbf{\Sigma}_g(\mathsf{I}_g, \mathsf{B}_g)$ and $\mathbf{\Sigma}_g(\mathsf{B}_g, \mathsf{I}_g)$. To show this rigorously, we need to prove theorems similar to Theorem B.1 and Theorem 3.1 and leave the proofs in Appendix B.2.

4.1.1 The algorithm

Now we consider clusters i, j, and k in the augmented tree rooted at cluster r, with C_i and C_j the two children of C_k. By Theorem B.2, Corollary B.4, and Corollary

B.5, we have

$$\mathbf{\Sigma}_{k+}(\mathsf{B}_k, \mathsf{B}_k) = \mathbf{\Sigma}_k(\mathsf{B}_k, \mathsf{B}_k) + \mathbf{L}_k^{-1}\mathbf{\Sigma}_k(\mathsf{S}_k, \mathsf{B}_k) + \mathbf{\Sigma}_k(\mathsf{B}_k, \mathsf{S}_k)\mathbf{L}_k^{-\dagger} + \mathbf{L}_k^{-1}\mathbf{\Sigma}_k(\mathsf{S}_k, \mathsf{S}_k)\mathbf{L}_k^{-\dagger}, \tag{4.1}$$

where $\mathbf{L}_k^{-1} = -\mathbf{A}_{r,k}(\mathsf{B}_k, \mathsf{S}_k)\mathbf{A}_{r,k}^{-1}(\mathsf{S}_k, \mathsf{S}_k)$, $\mathbf{A}_{r,k}(\mathsf{B}_k \cup \mathsf{S}_k, \mathsf{B}_k \cup \mathsf{S}_k)$ follows the same update rule as in the basic FIND algorithm, and $\mathbf{\Sigma}_k(\mathsf{S}_k \cup \mathsf{B}_k, \mathsf{S}_k \cup \mathsf{B}_k)$ has the following form:

$$\mathbf{\Sigma}_k(\mathsf{B}_k, \mathsf{B}_k) = \begin{pmatrix} \mathbf{\Sigma}_i(\mathsf{B}_k \cap \mathsf{B}_i, \mathsf{B}_k \cap \mathsf{B}_i) & \mathbf{\Sigma}(\mathsf{B}_k \cap \mathsf{B}_i, \mathsf{B}_k \cap \mathsf{B}_j) \\ \mathbf{\Sigma}(\mathsf{B}_k \cap \mathsf{B}_j, \mathsf{B}_k \cap \mathsf{B}_i) & \mathbf{\Sigma}_j(\mathsf{B}_k \cap \mathsf{B}_j, \mathsf{B}_k \cap \mathsf{B}_j) \end{pmatrix},$$

$$\mathbf{\Sigma}_k(\mathsf{S}_k, \mathsf{S}_k) = \begin{pmatrix} \mathbf{\Sigma}_i(\mathsf{S}_k \cap \mathsf{B}_i, \mathsf{S}_k \cap \mathsf{B}_i) & \mathbf{\Sigma}(\mathsf{S}_k \cap \mathsf{B}_i, \mathsf{S}_k \cap \mathsf{B}_j) \\ \Sigma(\mathsf{S}_k \cap \mathsf{B}_j, \mathsf{S}_k \cap \mathsf{B}_i) & \mathbf{\Sigma}_j(\mathsf{S}_k \cap \mathsf{B}_j, \mathsf{S}_k \cap \mathsf{B}_j) \end{pmatrix},$$

$$\mathbf{\Sigma}_k(\mathsf{B}_k, \mathsf{S}_k) = \begin{pmatrix} \mathbf{\Sigma}_i(\mathsf{B}_k \cap \mathsf{B}_i, \mathsf{S}_k \cap \mathsf{B}_i) & 0 \\ 0 & \mathbf{\Sigma}_j(\mathsf{B}_k \cap \mathsf{B}_j, \mathsf{S}_k \cap \mathsf{B}_j) \end{pmatrix},$$

$$\mathbf{\Sigma}_k(\mathsf{S}_k, \mathsf{B}_k) = \begin{pmatrix} \mathbf{\Sigma}_i(\mathsf{S}_k \cap \mathsf{B}_i, \mathsf{B}_k \cap \mathsf{B}_i) & 0 \\ 0 & \mathbf{\Sigma}_j(\mathsf{S}_k \cap \mathsf{B}_j, \mathsf{B}_k \cap \mathsf{B}_j) \end{pmatrix}.$$

Similar to the basic FIND algorithm, if C_k is a leaf node, then by Corollary B.6 we have $\mathbf{\Sigma}_{r,k}(\mathsf{B}_k \cup \mathsf{S}_k, \mathsf{B}_k \cup \mathsf{S}_k) = \mathbf{\Sigma}_r(\mathsf{B}_k \cup \mathsf{S}_k, \mathsf{B}_k \cup \mathsf{S}_k)$ (the entries in the original $\mathbf{\Sigma}$). The recursive process of updating $\mathbf{\Sigma}$ is the same as updating $\mathbf{U}$ in the basic FIND algorithm, but the update rule is different here.

4.1.2 The pseudocode of the algorithm

The pseudocode of the algorithm is an extension of the basic FIND algorithm. In addition to rearranging the matrix and updating the matrix $\mathbf{U}$, we need to do similar things for the matrix $\mathbf{\Sigma}$ as well. Note that even if we do not need $\mathbf{A}^{-1}$, we still need to keep track of the update of $\mathbf{A}$, i.e., the $\mathbf{U}$ matrices. This is because we need $\mathbf{A}_k(\mathsf{B}_k \cup \mathsf{S}_k, \mathsf{B}_k \cup \mathsf{S}_k)$ when we update $\mathbf{\Sigma}_k(\mathsf{B}_k, \mathsf{B}_k)$.

In the algorithm, we first compute $\mathbf{R}_\mathsf{C}$ for all the positive nodes in the upward pass as shown in procedures `updateBoundaryNodes`. We then compute $\mathbf{R}_\mathsf{C}$ for all the negative nodes in the downward pass as shown in procedure `updateAdjacentNodes`. The whole algorithm is shown in procedure `computeG<`.

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

```
   Data: tree decomposition of the mesh; the matrix $\mathbf{A}$.
   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 $\mathbf{U}_C$ for $\mathbf{A}^{-1}$ and $\mathbf{\Sigma}_C$ for $\mathbf{A}^{-1}\mathbf{\Sigma}\mathbf{A}^{-\dagger}$ are
           updated and saved.
1  if C is not a leaf then
2      C1 = left child of C;
3      C2 = right child of C;
4      updateBoundaryNodes(C1)   /* The boundary set is denoted $B_{C1}$ */;
5      updateBoundaryNodes(C2)   /* The boundary set is denoted $B_C$ */;
6  else
7      $\mathbf{A}_C = \mathbf{A}(C, C)$;
8  if C is not the root then
9      $\mathbf{A}_C = [\mathbf{U}_{C1}\ \mathbf{A}(B_{C1}, B_{C2});\ \mathbf{A}(B_{C2}, B_{C1})\ \mathbf{U}_{C2}]$;
       /* A($B_{C1}$,$B_{C2}$) and A($B_{C2}$,$B_{C1}$) are values from the original matrix
          A.                                                                    */
10     Rearrange $\mathbf{A}_C$ such that the inner nodes of $B_{C1} \cup B_{C2}$ appear first;
11     Set $S_k$ in (4.1) equal to the above inner nodes and $B_k$ equal to the rest;
12     Eliminate the inner nodes and set $\mathbf{U}_C$ equal to the Schur complement of
       $\mathbf{A}_C$;
13     $\mathbf{\Sigma}_C = [\mathbf{\Sigma}_{C1}\ \mathbf{\Sigma}(B_{C1}, B_{C2});\ \mathbf{\Sigma}(B_{C2}, B_{C1})\ \mathbf{\Sigma}_{C2}]$;
       /* $\Sigma$($B_{C1}$, $B_{C2}$) and $\Sigma$($B_{C2}$, $B_{C1}$) are values from the original
          matrix $\Sigma$.                                                      */
14     Rearrange $\mathbf{\Sigma}_C$ in the same way as $\mathbf{A}_C$;
15     Compute $\mathbf{\Sigma}_C$ based on (4.1): $S_k$ has been given above and $B_k$ is the
       boundary nodes of C; $\mathbf{\Sigma}_C = \mathbf{\Sigma}_{k+}(B_k, B_k)$;
16     Save $\mathbf{U}_C$ and $\mathbf{\Sigma}_C$
```

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

Data: tree decomposition of the mesh; the matrix $\mathbf{A}$; the upward pass [updateBoundaryNodes()] should have been completed.

Input: cluster C with n adjacent mesh nodes (as the boundary nodes of $\bar{\mathsf{C}}$).

Output: all the outer mesh nodes of cluster C (as the inner nodes of $\bar{\mathsf{C}}$) are eliminated by the procedure. The matrix $\mathbf{R}_{\bar{\mathsf{C}}}$ with the result of the elimination is saved.

```
1  if $\mathsf{C}$ is not the root then
2      $\mathsf{D}$ = parent of $\mathsf{C}$          /* The boundary set of $\bar{\mathsf{D}}$ is denoted $\mathsf{B}_{\bar{\mathsf{D}}}$ */;
3      $\mathsf{D1}$ = sibling of $\mathsf{C}$       /* The boundary set of D1 is denoted $\mathsf{B}_{\mathsf{D1}}$ */;
4      $\mathbf{A}_{\bar{\mathsf{C}}} = [\mathbf{U}_{\bar{\mathsf{D}}}\ \mathbf{A}(\mathsf{B}_{\bar{\mathsf{D}}}, \mathsf{B}_{\mathsf{D1}}); \mathbf{A}(\mathsf{B}_{\mathsf{D1}}, \mathsf{B}_{\bar{\mathsf{D}}})\ \mathbf{U}_{\mathsf{D1}}]$;
       /* A($\mathsf{B}_{\bar{\mathsf{D}}}$,$\mathsf{B}_{\mathsf{D1}}$) and A($\mathsf{B}_{\mathsf{D1}}$,$\mathsf{B}_{\bar{\mathsf{D}}}$) are values from the original matrix
          A.                                                            */
       /* If D is the root, then $\bar{\mathsf{D}} = \emptyset$ and $\mathsf{A}_{\bar{\mathsf{C}}}$ = $\mathsf{R}_{\mathsf{D1}}$.                  */
5      Rearrange $\mathbf{A}_{\bar{\mathsf{C}}}$ such that the inner nodes of $\mathsf{B}_{\bar{\mathsf{D}}} \cup \mathsf{B}_{\mathsf{D1}}$ appear first;
6      Set $\mathsf{S}_{\mathrm{k}}$ in (4.1) equal to the above inner nodes and $\mathsf{B}_{\mathrm{k}}$ equal to the rest;
7      Eliminate the outer nodes; set $\mathbf{U}_{\bar{\mathsf{C}}}$ equal to the schur complement of $\mathbf{A}_{\bar{\mathsf{C}}}$;
8      $\boldsymbol{\Sigma}_{\bar{\mathsf{C}}} = [\mathbf{R}_{\bar{\mathsf{D}}}\ \boldsymbol{\Sigma}(\mathsf{B}_{\bar{\mathsf{D}}}, \mathsf{B}_{\mathsf{D1}}); \boldsymbol{\Sigma}(\mathsf{B}_{\mathsf{D1}}, \mathsf{B}_{\bar{\mathsf{D}}})\ \mathbf{R}_{\mathsf{D1}}]$;
9      Rearrange $\boldsymbol{\Sigma}_{\bar{\mathsf{C}}}$ in the same way as for $\mathbf{A}_{\bar{\mathsf{C}}}$;
10     Compute $\mathbf{R}_{\bar{\mathsf{C}}}$ based on Eq.(4.1); $\mathbf{R}_{\bar{\mathsf{C}}} = \mathbf{R}_{\mathrm{k+}}(\mathsf{B}_{\mathrm{k}}, \mathsf{B}_{\mathrm{k}})$;
11     Save $\mathbf{U}_{\bar{\mathsf{C}}}$ and $\mathbf{R}_{\bar{\mathsf{C}}}$;
12 if $\mathsf{C}$ is not a leaf then
13     $\mathsf{C1}$ = left child of $\mathsf{C}$;
14     $\mathsf{C2}$ = right child of $\mathsf{C}$;
15     updateAdjacentNodes($\mathsf{C1}$);
16     updateAdjacentNodes($\mathsf{C2}$);
```

Procedure `computeG<`(*mesh* M). This procedure should be called by any function that needs $\mathbf{G}^<$ of the whole mesh M.

```
Data: the mesh $\mathsf{M}$; the matrix $\mathbf{A}$; enough memory should be available for
      temporary storage
Input: the mesh $\mathsf{M}$; the matrix $\mathbf{A}$;
Output: the diagonal entries of $\mathbf{G}^<$
1 Prepare the tree decomposition of the whole mesh;
2 for each leaf node $\mathsf{C}$ do
3     Compute $[\mathbf{A}^{-1}](\mathsf{C}, \mathsf{C})$ using Eq. (6.8);
4     Compute $\mathbf{R}_\mathsf{C}$ based on Eq. (7);
5     Save $\mathbf{R}_\mathsf{C}$ together with its indices
6 Collect $\mathbf{R}_\mathsf{C}$ with their indices and output the diagonal entries of $\mathbf{R}$
```

4.1.3 Computation and storage cost

Similar to the computation cost analysis in Chapter 3, the major computation cost comes from computing (4.1).

The cost depends on the size of S_k and B_k. Let $\mathsf{s} = |\mathsf{S}_k|$ and $\mathsf{b} = |\mathsf{B}_k|$ be the sizes, then the computation cost for $\mathbf{L}_k^{-1}\mathbf{R}_k(\mathsf{S}_k, \mathsf{B}_k)$ is $(\mathsf{s}^2\mathsf{b} + \frac{1}{3}\mathsf{s}^3 + \mathsf{sb}^2)$ flops. Since $\mathbf{L}_k^{-1}$ is already given by (3.2) and computed in procedures `eliminateInnerNodes` (page 27) and `eliminateOuterNodes` (page 28), the cost is reduced to sb^2. Similarly, the cost for $\mathbf{R}_k(\mathsf{B}_k, \mathsf{S}_k)\mathbf{L}_k^{-\dagger}$ is also sb^2 and the cost for $\mathbf{L}_k^{-1}\mathbf{R}_k(\mathsf{S}_k, \mathsf{S}_k)\mathbf{L}_k^{-\dagger}$ is $\mathsf{sb}^2 + \mathsf{s}^2\mathsf{b}$. So the total cost for (4.1) is $(3\mathsf{sb}^2 + \mathsf{s}^2\mathsf{b})$ flops. Note that these cost estimates need explicit forms of $\mathbf{A}(\mathsf{B}, \mathsf{S})\mathbf{A}(\mathsf{S}, \mathsf{S})^{-1}$ and we need to arrange the order of computation to have it as an intermediate result in computing (3.2).

Now we consider the whole process. We analyze the cost in the two passes.

For upward pass, when two $a \times a$ clusters merge into one $a \times 2a$ cluster, $\mathsf{b} \le 6a$ and $\mathsf{s} \le 2a$, so the cost is at most $360a^3$ flops; when two $a \times 2a$ clusters merger into one $2a \times 2a$ cluster, $\mathsf{b} \le 8a$ and $\mathsf{s} \le 4a$, so the cost is at most $896a^3$ flops. We have $\frac{N_xN_y}{2a^2}$ $a \times 2a$ clusters and $\frac{N_xN_y}{4a^2}$ $2a \times 2a$ clusters, so the cost for these two levels is $(360a^3\frac{N_xN_y}{2a^2} + 896a^3\frac{N_xN_y}{4a^2} =)$ $404N_xN_ya$. Summing all the levels together up to $a = N_x/2$, the computation cost for upward pass is $404N_x^2N_y$.

For downward pass, when one $2a \times 2a$ cluster is partitioned into two $a \times 2a$ clusters, $\mathsf{b} \leq 6a$ and $\mathsf{s} \leq 8a$, so the cost is at most $1248a^3$ flops; when one $a \times 2a$ cluster is partitioned intoto two $a \times a$ clusters, $\mathsf{b} \leq 4a$ and $|\mathsf{S}_k| \leq 6a$, so the cost is at most $432a^3$ flops. We have $\frac{N_x N_y}{2a^2}$ $a \times 2a$ clusters and $\frac{N_x N_y}{a^2}$ $a \times a$ clusters, so the cost for these two levels is $(1248a^3 \frac{N_x N_y}{2a^2} + 432a^3 \frac{N_x N_y}{a^2} =)$ $1056 N_x N_y a$. Summing all the levels together up to $a = N_x/2$, the computation cost for upward pass is at most $1056 N_x^2 N_y$.

4.2 Computing off-diagonal entries of $\mathbf{G}^r$ and $\mathbf{G}^<$

In addition to computing the diagonal entries of $\mathbf{G}^r$ and $\mathbf{G}^<$, the algorithm can be easily extended to computing the entries in $\mathbf{G}^r$ and $\mathbf{G}^<$ corresponding to neighboring nodes. These entries are often of interest in simulation. For example, the current density can be computed via the entries corresponding to the horizontally neighboring nodes.

With a little more computation, these entries can be obtained at the same time when the algorithm computes the diagonal entries of the inverse. Fig. 4.1 shows the last step of elimination and the inverse computation for the solid circle nodes.

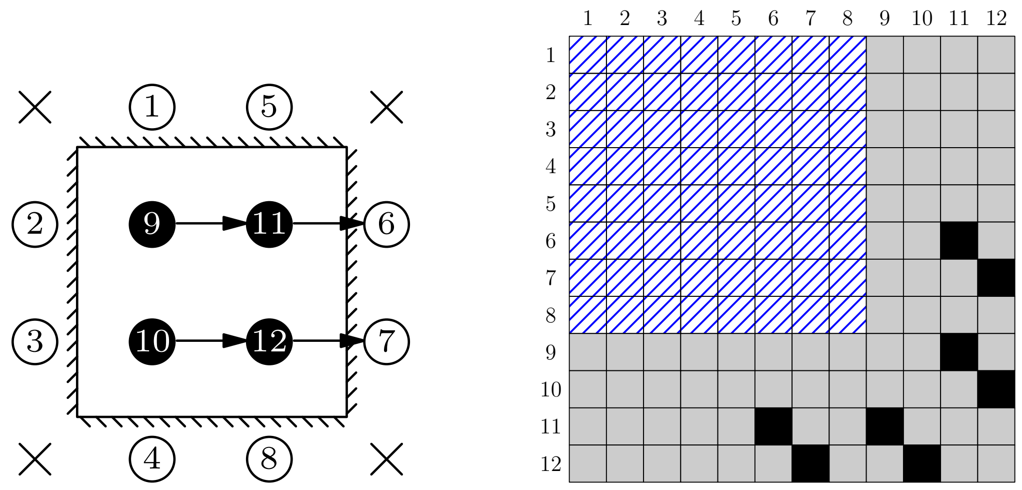


Figure 4.1: Last step of elimination

The entries of $\mathbf{A}^{-1}$ corresponding to nodes in C can be computed by the following equation:

$$[\mathbf{A}^{-1}](\mathsf{C},\mathsf{C}) = [\mathbf{A}(\mathsf{C},\mathsf{C}) - \mathbf{A}(\mathsf{C},\mathsf{B}_{\bar{\mathsf{C}}})(\mathbf{U}_{\bar{\mathsf{C}}})^{-1}\mathbf{A}(\mathsf{B}_{\bar{\mathsf{C}}},\mathsf{C})]^{-1}. \tag{4.2}$$

Performing one step of back substitution, we have

$$[\mathbf{A}^{-1}](\mathsf{B}_{\bar{\mathsf{C}}},\mathsf{C}) = -(\mathbf{U}_{\bar{\mathsf{C}}})^{-1}\mathbf{A}(\mathsf{B}_{\bar{\mathsf{C}}},\mathsf{C})[\mathbf{A}^{-1}](\mathsf{C},\mathsf{C}) \tag{4.3}$$

and

$$[\mathbf{A}^{-1}](\mathsf{C},\mathsf{B}_{\bar{\mathsf{C}}}) = -[\mathbf{A}^{-1}](\mathsf{C},\mathsf{C})\mathbf{A}(\mathsf{C},\mathsf{B}_{\bar{\mathsf{C}}})(\mathbf{U}_{\bar{\mathsf{C}}})^{-1}. \tag{4.4}$$

This could also be generalized to nodes with a distance from each other.

Chapter 5

Optimization of FIND

In our FIND algorithm, we have achieved $\mathcal{O}(N_x^2 N_y)$ computational complexity for a 2D mesh of size $N_x \times N_y$. Even though this complexity has been reduced by an order of magnitude compared to the state-of-the-art RGF method, the matrix inverse computations are still the most time-consuming part and often prohibitively expensive in transport problem simulation. When we tried to make our method faster, we observed that even though the order of complexity cannot be improved by any method for the given sparsity, the constant factor involved in the computation cost can be significantly improved. This chapter discusses how to achieve this.

5.1 Introduction

In the FIND algorithm, the major operations are to perform Gaussian eliminations. All such operations are in the form

$$\mathbf{U} = \mathbf{A}(\mathsf{B},\mathsf{B}) - \mathbf{A}(\mathsf{B},\mathsf{S})\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}\mathbf{A}(\mathsf{S},\mathsf{B}) \tag{5.1}$$

for $\mathbf{G}$ and

$$\mathbf{R} = \mathbf{\Sigma}(\mathsf{B},\mathsf{B}) + \mathbf{L}^{-1}\mathbf{\Sigma}(\mathsf{S},\mathsf{B}) + \mathbf{\Sigma}(\mathsf{B},\mathsf{S})\mathbf{L}^{-\dagger} + \mathbf{L}^{-1}\mathbf{\Sigma}(\mathsf{S},\mathsf{S})\mathbf{L}^{-\dagger} \tag{5.2}$$

for $\mathbf{G}^<$, where

$$\mathbf{L}^{-1} = -\mathbf{A}(\mathsf{B},\mathsf{S})\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}. \tag{5.3}$$

These equations are copied from (3.2) and (4.1) with subscripts r and k skipped for simplicity.[1] As a result, computing (5.1) and (5.2) efficiently becomes critical for our algorithm to achieve good overall performance.

In Chapter 3, we gave the computation cost

$$\frac{1}{3}\mathsf{s}^3 + \mathsf{s}^2 b + \mathsf{s}\mathsf{b}^2 \tag{5.4}$$

for (5.1) and

$$\mathsf{s}^2\mathsf{b} + 3\mathsf{s}\mathsf{b}^2 \tag{5.5}$$

for (5.2), where s and b are the size of sets S and B, respectively. These costs assume that all matrices in (5.1) and (5.2) are general dense matrices. These matrices, however, are themselves sparse for a typical 2D mesh. In addition, due to the characteristics of the physical problem in many real applications, the matrices $\mathbf{A}$ and $\mathbf{\Sigma}$ often have special properties [Svizhenko et al., 2002]. Such sparsities and properties will not change the order of cost, but may reduce the constant factor of the cost, thus achieving significant speed-up. With proper optimization, our FIND algorithm can exceed other algorithms on a much smaller mesh.

In this chapter, we first exploit the extra sparsity of the matrices in (3.2) and (4.1) to reduce the constant factor in the computation and the storage cost. We then consider the symmetry and positive definiteness of the given matrix $\mathbf{A}$ for further performance improvement. We also apply these optimization techniques to $\mathbf{G}^<$ and current density when $\mathbf{\Sigma}$ has similar properties. Finally, we briefly discuss the singularity of $\mathbf{A}$ and how to deal with it efficiently.

[1]Note that some entries of $\mathbf{A}(\bullet,\bullet)$ and $\mathbf{\Sigma}(\bullet,\bullet)$ have been updated from the blocks in the original $\mathbf{A}$ and $\mathbf{\Sigma}$. See Section 3.2.3 for details.

5.2 Optimization for extra sparsity in A

Because of the local connectivity of 2D mesh in our approach,[2] the matrices $\mathbf{A}(\mathsf{B},\mathsf{S})$, $\mathbf{A}(\mathsf{S},\mathsf{S})$, and $\mathbf{A}(\mathsf{S},\mathsf{B})$ in (5.1) are not dense. Such sparsity will not reduce the storage cost because $\mathbf{U}$ in (5.1) is still dense, which is the major component of the storage cost. Due to the sparsity of these matrices, however, the computation cost can be significantly reduced. To clearly observe the sparsity and exploit it, it is convenient to consider these matrices in blocks.

5.2.1 Schematic description of the extra sparsity

As shown earlier in Fig. 3.2, we eliminate the private inner nodes when we merge two child clusters in the tree into their parent cluster. Here in Figs. 5.1 and 5.2, we distinguish between the left cluster and the right cluster in the merge and its corresponding eliminations in matrix form to illustrate the extra sparsity.

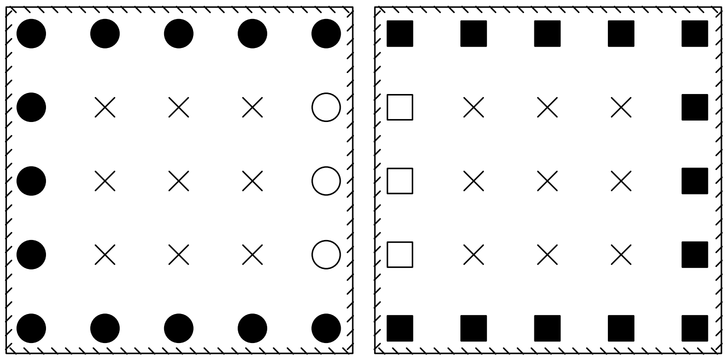


Figure 5.1: Two clusters merge into one larger cluster in the upward pass. The $\times$ nodes have already been eliminated. The ○ and □ nodes remain to be eliminated.

In Fig. 5.1, the three hollow circle nodes and the three hollow square nodes are private inner nodes of the parent cluster that originate from the left child cluster and the right child cluster, respectively. When we merge the two child clusters, these nodes will be eliminated. Fig. 5.2 illustrates how we construct a matrix based

[2]The approach works for 3D mesh as well.

on the original matrix **A** and the results of earlier eliminations for the elimination[3] corresponding to the merge in Fig. 5.1.

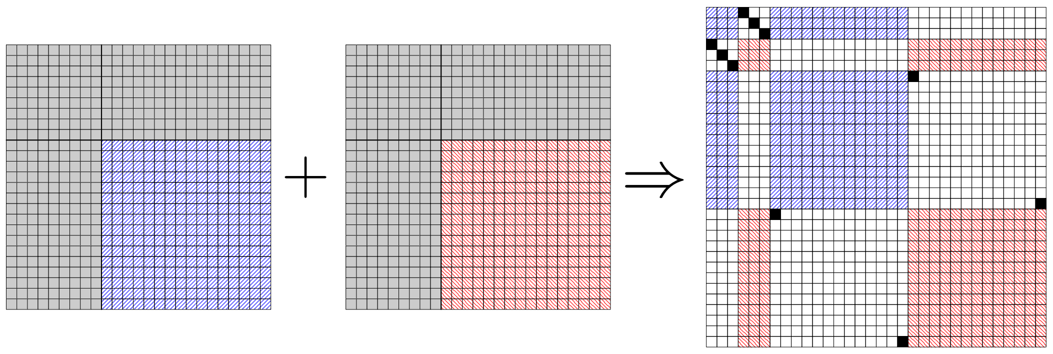


Figure 5.2: Data flow from early eliminations for the left child and the right child to the elimination for the parent cluster

In Fig. 5.2, each row (column) of small blocks corresponds to a node in Fig. 5.1. The blocks with blue and red shading correspond to the boundary nodes of the two child clusters in Fig. 5.1 (the circle and square nodes): those in blue correspond to the left cluster and those in red correspond to the right cluster in Fig. 5.1 (the hollow and solid square nodes). They are rearranged to eliminate the private inner nodes (the six hollow nodes). We can also write the resulting matrix in Fig. 5.2 as

$$\begin{pmatrix} \boldsymbol{U}(\circ,\circ) & \mathbf{A}(\circ,\square) & \boldsymbol{U}(\circ,\bullet) & 0 \\ \mathbf{A}(\square,\circ) & \boldsymbol{U}(\square,\square) & 0 & U(\square,\blacksquare) \\ \boldsymbol{U}(\bullet,\circ) & 0 & \boldsymbol{U}(\bullet,\bullet) & \mathbf{A}(\bullet,\blacksquare) \\ 0 & \boldsymbol{U}(\circ,\bullet) & \mathbf{A}(\blacksquare,\bullet) & \boldsymbol{U}(\blacksquare,\blacksquare) \end{pmatrix}. \tag{5.6}$$

In (5.6), $\circ$, $\square$, $\bullet$, and $\blacksquare$ correspond to all the $\circ$ nodes, $\square$ nodes, $\bullet$ nodes, and $\blacksquare$ nodes in Fig. 5.2, respectively. The **U** blocks originate from the results of earlier eliminations. They are typically dense, since after the elimination of the inner nodes, the remaining nodes become fully connected. The **A** blocks, however, originate from the original **A**, and are often sparse. For example, because each $\circ$ node is connected with only one $\square$ node in Fig. 5.1, $\mathbf{A}(\circ,\square)$ and $\mathbf{A}(\square,\circ)$ are both diagonal. $\mathbf{A}(\circ,\blacksquare)$,

[3]Here *elimination* refers to the partial elimination in Chapter 3.

$\mathbf{A}(\blacksquare, \bigcirc)$, $\mathbf{A}(\square, \bullet)$, and $\mathbf{A}(\bullet, \square)$ are not shown in (5.6) because they are all 0. Note that $\mathbf{A}(\bullet, \blacksquare)$ and $\mathbf{A}(\blacksquare, \bullet)$ are almost all 0 except for a few entries, but such sparsity saves little because they are only involved in floating-point addition operations. We will not exploit such sparsity.

The elimination shown here is one of the steps we discussed in Section 3.2.3, with the left cluster labeled as i, the right one as j, and the parent cluster labeled as k. Below is the list of node types used in this section and the corresponding sets used in Section 3.2.3:

$\bigcirc$: $\mathsf{S}_L = \mathsf{S}_k \cap \mathsf{B}_i$ (private inner nodes from the left cluster)

$\square$: $\mathsf{S}_R = \mathsf{S}_k \cap \mathsf{B}_j$ (private inner nodes from the right cluster)

$\bullet$: $\mathsf{B}_L = \mathsf{B}_k \cap \mathsf{B}_i$ (boundary nodes from the left cluster)

$\blacksquare$: $\mathsf{B}_R = \mathsf{B}_k \cap \mathsf{B}_j$ (boundary nodes from the rightcluster)

The sparsity of the matrices in (3.2) can therefore be demonstrated as follows:

- $\mathbf{A}_{r,k}(\mathsf{B}_k \cap \mathsf{B}_i, \mathsf{S}_k \cap \mathsf{B}_j) = \mathbf{A}_{r,k}(\mathsf{S}_k \cap \mathsf{B}_j, \mathsf{B}_k \cap \mathsf{B}_i) = 0$,
- $\mathbf{A}_{r,k}(\mathsf{B}_k \cap \mathsf{B}_j, \mathsf{S}_k \cap \mathsf{B}_i) = \mathbf{A}_{r,k}(\mathsf{S}_k \cap \mathsf{B}_i, \mathsf{B}_k \cap \mathsf{B}_j) = 0$,
- $\mathbf{A}_{r,k}(\mathsf{S}_k \cap \mathsf{B}_i, \mathsf{S}_k \cap \mathsf{B}_j)$ and $\mathbf{A}_{r,k}(\mathsf{S}_k \cap \mathsf{B}_j, \mathsf{S}_k \cap \mathsf{B}_i)$ are diagonal.

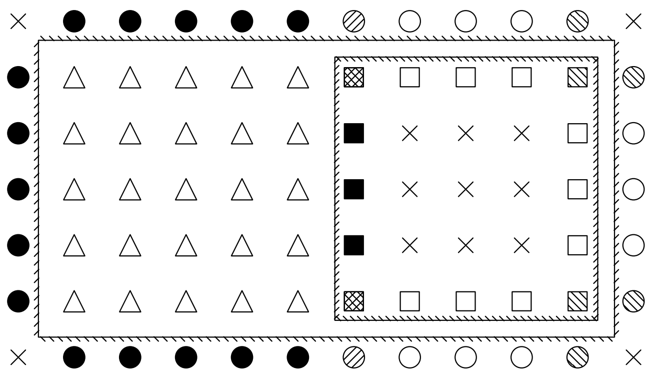


Figure 5.3: Two clusters merge into one larger cluster in the downward pass.

These properties always hold for the upward pass as shown in Figs. 5.1 and Fig. 5.2. They hold for the downward pass as well but with a small number of exceptions, illustrated by the shaded nodes in Fig. 5.3 and by the green nodes and yellow blocks in Fig. 5.4. Since they only appear at the corners, we can ignore them when we analyze the computation and storage cost. Note that such exceptions need special treatment in implementation.

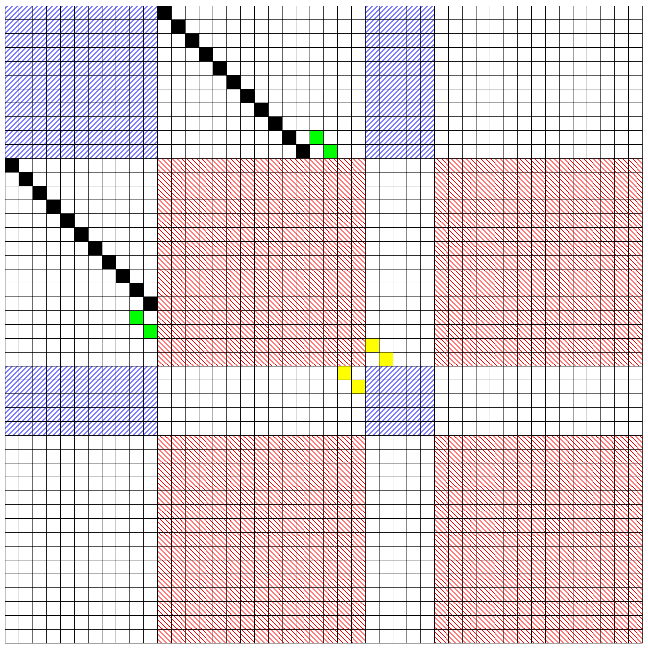


Figure 5.4: Two clusters merge into one larger cluster in the downward pass.

Because typical matrix computation libraries only consider banded matrices and block-diagonal matrices, we utilize special techniques for $\mathbf{A}_{r,k}(\mathsf{S}_k, \mathsf{S}_k)$ (illustrated in Fig. 5.2 as the upper left 6×6 little blocks) to take full advantage of the sparsity. Sections 5.2.2 and 5.2.3 will discuss this procedure in detail.

5.2.2 Preliminary analysis

Before proceeding to further analysis, we list some facts about the cost of the basic matrix computation used in Section 5.2.3. By simple counting, we know that for $n \times n$ full matrices A and B, there are the following costs (in floating-point multiplications):

- $A \Rightarrow LU$: $n^3/3$
- $L \Rightarrow L^{-1}$: $n^3/6$
- $L, B \Rightarrow L^{-1}B$: $n^3/2$
- $U, L^{-1}B \Rightarrow A^{-1}B$: $n^3/2$

Adding them together, computing A^{-1} requires n^3 and computing $A^{-1}B$ requires $\frac{4}{3}n^3$. Note that the order of computation is often important, e.g., if we compute $A^{-1}B = (U^{-1}L^{-1})B$, then the cost is $(n^3+n^3 =)\ 2n^3$, whereas computing $U^{-1}(L^{-1}B)$ only requires $\frac{4}{3}n^3$.

For simple matrix multiplication, if both matrices are full, then the cost is n^3. If one of them is a diagonal matrix, then the cost is reduced to $\mathcal{O}(n^2)$. In addition, for a general 2×2 block matrix, we have the following forms of the inverse:[4]

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} = \begin{pmatrix} \tilde{A}^{-1} & -A^{-1}B\tilde{D}^{-1} \\ -D^{-1}C\tilde{A}^{-1} & \tilde{D}^{-1} \end{pmatrix} \tag{5.7a}$$

$$= \begin{pmatrix} A^{-1} + A^{-1}B\tilde{D}^{-1}CA^{-1} & -A^{-1}B\tilde{D}^{-1} \\ -\tilde{D}^{-1}CA^{-1} & \tilde{D}^{-1} \end{pmatrix} \tag{5.7b}$$

$$= \begin{pmatrix} A & B \\ 0 & \tilde{D} \end{pmatrix}^{-1} \begin{pmatrix} I & 0 \\ CA^{-1} & I \end{pmatrix}^{-1}, \tag{5.7c}$$

where $\tilde{A} = A - BD^{-1}C$ and $\tilde{D} = D - CA^{-1}B$ are the Schur complements of D and A, respectively. In (5.7a) and (5.7b), the two off-diagonal blocks of the inverse are the results of back substitution. In (5.7a), they are independently based on $\tilde{A}^{-1}$ and $\tilde{D}^{-1}$, which is referred to as *one-way back substitution*; whereas in (5.7b), both are based on $\tilde{D}^{-1}$, which is called a *two-way back substitution*. Eq. (5.7c) is based on the block LU factorization of $\mathbf{A}(\mathsf{S},\mathsf{S})$, which is called *block LU inverse*.

For multiplication of 2×2 block matrices with block size $n \times n$, if A and B are both full, the cost is $8n^3$. If one of them is a 2×2 block-diagonal matrix, then the cost is reduced to $4n^3$.

[4]We assume A and D to be nonsingular.

5.2.3 Exploiting the extra sparsity in a block structure

In this section, we discuss several ways of handling $\mathbf{A}(\mathsf{S},\mathsf{S})$ to exploit the sparsity shown in Section 5.2.1. They are not proved to be optimal. Depending on the computer architecture and the implementation of our methods, there are quite likely other techniques to achieve better speed-up. The discussion below demonstrates the speed-up achieved through simple techniques. For notational simplicity, in this section we write $\mathbf{A}(\mathsf{S},\mathsf{S})$ as $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$, with $A = \mathbf{A}(\mathsf{S}_L,\mathsf{S}_L)$, $B = \mathbf{A}(\mathsf{S}_L,\mathsf{S}_R)$, $C = \mathbf{A}(\mathsf{S}_R,\mathsf{S}_L)$, and $D = \mathbf{A}(\mathsf{S}_R,\mathsf{S}_R)$; $\mathbf{A}(\mathsf{S},\mathsf{B})$ as $\begin{pmatrix} W & X \\ Y & Z \end{pmatrix}$, with $W = \mathbf{A}(\mathsf{S}_L,\mathsf{B}_L)$, $X = \mathbf{A}(\mathsf{S}_L,\mathsf{B}_R)$, $Y = \mathbf{A}(\mathsf{S}_R,\mathsf{B}_L)$, and $Z = \mathbf{A}(\mathsf{S}_R,\mathsf{B}_R)$; and $\mathbf{A}(\mathsf{B},\mathsf{S})$ as $\begin{pmatrix} P & Q \\ R & S \end{pmatrix}$, with $P = \mathbf{A}(\mathsf{B}_L,\mathsf{S}_L)$, $Q = \mathbf{A}(\mathsf{B}_L,\mathsf{S}_R)$, $R = \mathbf{A}(\mathsf{B}_R,\mathsf{S}_L)$, and $S = \mathbf{A}(\mathsf{B}_R,\mathsf{S}_R)$. Since the floating-point multiplication operation is much more expensive than addition operations, we ignore the addition with $\mathbf{A}(\mathsf{B},\mathsf{B})$.

The first method is based on (5.7). It computes $\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}$ through the Schur complement of the two diagonal blocks of $\mathbf{A}(\mathsf{S},\mathsf{S})$. Multiplying (5.7) by $\begin{pmatrix} W & X \\ Y & Z \end{pmatrix}$, from Table 5.1 we look at the result at the block level with the proper arrangement of the computation order to minimize the computation cost. This table also lists the required operations that are elaborated in Table 5.2.

Table 5.1: Matrix blocks and their corresponding operations

Block	*Expression*	*Operations*
(1, 1)	$\tilde{A}^{-1}W - A^{-1}B\tilde{D}^{-1}Y$	(5) (12)
(1, 2)	$\tilde{A}^{-1}X - A^{-1}B\tilde{D}^{-1}Z$	(9) (8)
(2, 1)	$-D^{-1}C\tilde{A}^{-1}W + \tilde{D}^{-1}Y$	(7) (10)
(2, 2)	$D^{-1}C\tilde{A}^{-1}X + \tilde{D}^{-1}Z$	(11) (6)

In Table 5.2, we list the operations required in Table 5.1 in sequence. An operation with lower sequence numbers must be performed first to reuse the results because the operations depend on one another.

Since the clusters are typically equally split, we can let $m = |\mathsf{S}_L| = |\mathsf{S}_R| = |\mathsf{S}|/2$

and $n = |\mathsf{B}_L| = |\mathsf{B}_R| = |\mathsf{B}|/2$. Now, the size of matrices A, B, C, and D is $m \times m$; the size of matrices W, X, Y, and Z is $m \times n$, and the size of matrices P, Q, R, and S is $m \times n$. For example, for a merge from two $a \times a$ clusters into one $a \times 2a$ cluster, we have $m = a$ and $n = 3a$.

Table 5.2: The cost of operations and their dependence in the first method. The costs are in flops. The size of A, B, C, and D is $m \times m$; the size of W, X, Y, and Z is $m \times n$.

	Type of matrix blocks				
Operation	all full	X, Y = 0	X, Y = 0; B, C = diag	*Seq. No.*	*Dep.*
$D\backslash C$	$4m^3/3$	$4m^3/3$	m^3	(1)	n/a
$A\backslash B$	$4m^3/3$	$4m^3/3$	m^3	(2)	n/a
$\tilde{A} = A - BD^{-1}C$	m^3	m^3	0	(3)	(1)
$\tilde{D} = D - CA^{-1}B$	m^3	m^3	0	(4)	(2)
$\tilde{A}\backslash W$	$\frac{m^3}{3} + m^2n$	$\frac{m^3}{3} + m^2n$	$\frac{m^3}{3} + m^2n$	(5)	(3)
$\tilde{D}\backslash Z$	$\frac{m^3}{3} + m^2n$	$\frac{m^3}{3} + m^2n$	$\frac{m^3}{3} + m^2n$	(6)	(4)
$-D^{-1}C\tilde{A}^{-1}W$	m^2n	m^2n	m^2n	(7)	(1) (5)
$-A^{-1}B\tilde{D}^{-1}Z$	m^2n	m^2n	m^2n	(8)	(2) (6)
$\tilde{A}^{-1}X$	m^2n	0	0	(9)	(5)
$\tilde{D}^{-1}Y$	m^2n	0	0	(10)	(6)
$-D^{-1}C\tilde{A}^{-1}X$	m^2n	0	0	(11)	(1) (9)
$-A^{-1}B\tilde{D}^{-1}Y$	m^2n	0	0	(12)	(2) (10)
Subtotal	$\frac{16m^3}{3} + 8m^2n$	$\frac{16m^3}{3} + 4m^2n$	$\frac{8m^3}{3} + 4m^2n$	n/a	n/a

Table 5.2 reveals the cost for computing $\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}\mathbf{A}(\mathsf{S},\mathsf{B})$ in (5.1). Now, we take $\mathbf{A}(\mathsf{B},\mathsf{S})$ into consideration. Multiplying $\mathbf{A}(\mathsf{B},\mathsf{S})$ from the left requires $4m^2n$, and then the total computation cost for (5.1) with the above optimization becomes $\frac{8}{3}m^3 + 4m^2n + 4mn^2$. In contrast, the cost without optimization given by (5.4) is $8mn^2 + 8m^2n + \frac{8}{3}m^3$ since $s = 2m$ and $b = 2n$.

Taking two upward cases as examples, $(m, n) = (a, 3a)$ and $(m, n) = (2a, 4a)$, the reduction is $\frac{296}{3}a^3 \to \frac{152}{3}a^3$ and $\frac{1216}{3}a^3 \to \frac{640}{3}a^3$. The cost with optimization lies somewhere between 51% to 53% of the original cost. Similarly, for the downward pass, in the two typical cases $(m, n) = (3a, 2a)$ and $(m, n) = (4a, 3a)$, we have the reduction $312a^3 \to 192a^3$ (62%) and $\frac{2528}{3} \to \frac{1520}{3}a^3$ (60%), respectively. Note that in the downward pass, $|\mathsf{B}_L| \neq |\mathsf{B}_R|$, but in terms of the computation cost estimate, it is equivalent to assuming that they are both $s/2$.

The second method is based on (5.7b). Like the first method, it does not compute $\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}$ explicitly. Instead, it computes $\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}\mathbf{A}(\mathsf{S},\mathsf{B})$ as

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} \begin{pmatrix} W & 0 \\ 0 & Z \end{pmatrix} = \begin{pmatrix} A^{-1}W + A^{-1}B\tilde{D}^{-1}CA^{-1}W & -A^{-1}B\tilde{D}^{-1}Z \\ -\tilde{D}^{-1}CA^{-1}W & \tilde{D}^{-1}Z \end{pmatrix}, \tag{5.8}$$

where $\tilde{D} = D - CA^{-1}B$, B and C are diagonal, and $X = Y = 0$. Table 5.3 shows the required operations with the total cost $\frac{4}{3}m^3 + 5m^2n + 4mn^2$. Compared to the previous method, the cost here is smaller if $m > \frac{3}{4}n$. Therefore, this method is better than the first method for the downward pass where typically (m,n) is $(3a,2a)$ and $(4a,2a)$. In such cases, the costs are $174a^3$ (56%) and $\frac{1408}{3}a^3$ (56%), respectively.

Table 5.3: The cost of operations and their dependence in the second method. The costs are in flops.

Operation	*Cost*	*Seq No.*	*Dependence*
A^{-1}	m^3	(1)	n/a
$\tilde{D}$	0	(2)	(1)
$\tilde{D}^{-1}Z$	$\frac{m^3}{3} + m^2n$	(3)	(2)
$-(A^{-1}B)(\tilde{D}^{-1}Z)$	m^2n	(4)	(1)(3)
$A^{-1}W$	m^2n	(5)	(1)
$-\tilde{D}^{-1}(CA^{-1}W)$	m^2n	(6)	(3)(5)
$(A^{-1}W) + (A^{-1}B)(\tilde{D}^{-1}CA^{-1}W)$	m^2n	(7)	(1)(5)(6)
$\mathbf{A}(\mathsf{S},\mathsf{B})\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}\mathbf{A}(\mathsf{S},\mathsf{B})$	$4mn^2$	(8)	(7)
Total	$\frac{4}{3}m^3 + 5m^2n + 4mn^2$	n/a	n/a

Note that if we compute the second block column of $\begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} \begin{pmatrix} W & 0 \\ 0 & Z \end{pmatrix}$ and $\begin{pmatrix} D & C \\ B & A \end{pmatrix}^{-1} \begin{pmatrix} Z & 0 \\ 0 & W \end{pmatrix}$, then we obtain the same form as in (5.8).

The third method is based on (5.7c). The two factors there will multiply $\mathbf{A}(\mathsf{B},\mathsf{S})$

and $\mathbf{A}(\mathsf{S},\mathsf{B})$ separately:

$$\begin{aligned}
&\mathbf{A}(\mathsf{B},\mathsf{S})\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}\mathbf{A}(\mathsf{S},\mathsf{B}) \\
=\ & \left[\begin{pmatrix} P & 0 \\ 0 & S \end{pmatrix}\begin{pmatrix} A & B \\ 0 & \tilde{D} \end{pmatrix}^{-1}\right]\left[\begin{pmatrix} I & 0 \\ CA^{-1} & I \end{pmatrix}^{-1}\begin{pmatrix} W & 0 \\ 0 & Z \end{pmatrix}\right] \\
=\ & \left[\begin{pmatrix} P & 0 \\ 0 & S \end{pmatrix}\begin{pmatrix} A^{-1} & -A^{-1}B\tilde{D}^{-1} \\ 0 & \tilde{D}^{-1} \end{pmatrix}\right]\left[\begin{pmatrix} I & 0 \\ -CA^{-1} & I \end{pmatrix}\begin{pmatrix} W & 0 \\ 0 & Z \end{pmatrix}\right] \\
=\ & \left[\begin{pmatrix} PA^{-1} & -PA^{-1}B\tilde{D}^{-1} \\ 0 & S\tilde{D}^{-1} \end{pmatrix}\right]\left[\begin{pmatrix} W & 0 \\ -CA^{-1}W & Z \end{pmatrix}\right].
\end{aligned} \tag{5.9}$$

Table 5.4 shows the operations with the total cost $\frac{4}{3}m^3 + 4m^2n + 5mn^2$. We compute the explicit form of A^{-1} there because we need $CA^{-1}B$ to compute $\tilde{D}$ and both B and C are diagonal. $S\tilde{D}^{-1}$ is computed by first LU factorizing $\tilde{D}$ and then solving for $S\tilde{D}^{-1}$. The LU form of $\tilde{D}$ will be used when computing $-(PA^{-1}B)\tilde{D}^{-1}$ as well. The cost for (5.9) results from the multiplication of a block upper triangular matrix and a block lower triangular matrix.

Table 5.4: The cost of operations in flops and their dependence in the third method

Operation	*Cost*	*Seq. No.*	*Dependence*
A^{-1}	m^3	(1)	n/a
$\tilde{D}$	0	(2)	(1)
$\tilde{D}^{-1}Z$	$\frac{m^3}{3} + m^2n$	(3)	(2)
$-(A^{-1}B)(\tilde{D}^{-1}Z)$	m^2n	(4)	(1)(3)
$A^{-1}W$	m^2n	(5)	(1)
$-\tilde{D}^{-1}(CA^{-1}W)$	m^2n	(6)	(3)(5)
$(A^{-1}W) + (A^{-1}B)(\tilde{D}^{-1}CA^{-1}W)$	m^2n	(7)	(1)(5)(6)
$\mathbf{A}(\mathsf{S},\mathsf{B})\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}\mathbf{A}(\mathsf{S},\mathsf{B})$	$4mn^2$	(8)	(7)
Total	$\frac{4}{3}m^3 + 5m^2n + 4mn^2$	n/a	n/a

Finally, our fourth method is more straightforward. It considers $\mathbf{A}(\mathsf{B},\mathsf{S})$ and $\mathbf{A}(\mathsf{S},\mathsf{B})$ as block-diagonal matrices but considers $\mathbf{A}(\mathsf{S},\mathsf{S})$ as a full matrix without exploiting its sparsity. Table 5.5 shows the operations with the total cost. Note that the cost for computing the second column of $\mathbf{L}^{-1}\mathbf{A}(\mathsf{S},\mathsf{B})$ is reduced because $\mathbf{A}(\mathsf{S},\mathsf{B})$ is block diagonal.

Table 5.5: The cost of operations in flops and their dependence in the fourth method. The size of A, B, C, and D is $m \times m$; the size of W, X, Y, and Z is $m \times n$

Operation	*Cost*
LU for $\mathbf{A}(\mathsf{S},\mathsf{S})$	$\frac{8}{3}m^3$
$\mathbf{L}^{-1}\mathbf{A}(\mathsf{S},\mathsf{B})$: 1st column	$2m^2n$
$\mathbf{L}^{-1}\mathbf{A}(\mathsf{S},\mathsf{B})$: 2nd column	$\frac{1}{2}m^2n$
$\mathbf{U}^{-1}[\mathbf{L}^{-1}\mathbf{A}(\mathsf{S},\mathsf{B})]$	$4m^2n$
$\mathbf{A}(\mathsf{S},\mathsf{B})\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}\mathbf{A}(\mathsf{S},\mathsf{B})$	$4mn^2$
Total	$\frac{8}{3}m^3 + \frac{13}{2}m^2n + 4mn^2$

Table 5.6 summarizes the optimization techniques for extra sparsity and the corresponding costs:

Table 5.6: Summary of the optimization methods

Method	*Cost*	*Reduction in four cases*[5]	*Parallelism*
2-way back substitution	$\frac{8}{3}m^3 + 4m^2n + 4mn^2$	51.4, 52.6, 61.5, 60.1	good
1-way back substitution	$\frac{4}{3}m^3 + 5m^2n + 4mn^2$	53.0, 54.0, 55.8, 55.7	little
block LU inverse	$\frac{4}{3}m^3 + 4m^2n + 5mn^2$	59.1, 57.9, 53.9, 54.3	some
naïve LU	$\frac{8}{3}m^3 + \frac{13}{2}m^2n + 4mn^2$	59.0, 62.5, 76.0, 74.4	little

Note again that these methods are not exhaustive. There are many variations of these methods with different forms of $\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}$, different computation orders, and different parts of the computation to share among the operations. For example, (5.10)

[5]The percentage of the cost given by (5.4).

is a variation of (5.9):

$$\begin{aligned}
&\mathbf{A}(\mathsf{B},\mathsf{S})\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}\mathbf{A}(\mathsf{S},\mathsf{B}) \\
&= \left[\begin{pmatrix} P & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} I & A^{-1}B \\ 0 & \tilde{D} \end{pmatrix}^{-1}\right] \left[\begin{pmatrix} A & 0 \\ C & I \end{pmatrix}^{-1} \begin{pmatrix} W & 0 \\ 0 & Z \end{pmatrix}\right] \\
&= \left[\begin{pmatrix} P & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} I & -A^{-1}B\tilde{D}^{-1} \\ 0 & \tilde{D}^{-1} \end{pmatrix}\right] \left[\begin{pmatrix} A & 0 \\ -CA^{-1} & I \end{pmatrix} \begin{pmatrix} W & 0 \\ 0 & Z \end{pmatrix}\right] \\
&= \left[\begin{pmatrix} P & -PA^{-1}B\tilde{D}^{-1} \\ 0 & S\tilde{D}^{-1} \end{pmatrix}\right] \left[\begin{pmatrix} A^{-1}W & 0 \\ -CA^{-1}W & Z \end{pmatrix}\right]. \qquad (5.10)
\end{aligned}$$

The method requires the computation cost $\frac{7}{3}m^3 + 3m^2n + 5mn^2$ as shown in Table 5.7.

Table 5.7: The cost of operations and their dependence for (5.10)

Operation	*Cost*	*Seq. No.*	*Dependence*
$A^{-1}, \tilde{D}$	m^3	(1)	n/a
$A^{-1}W$	m^2n	(2)	(1)
LU for $\tilde{D}$	$\frac{1}{3}m^3$	(3)	(1)
$(A^{-1}B)\tilde{D}^{-1}$	m^3	(4)	(1)(3)
$-P(A^{-1}B\tilde{D}^{-1})$	m^2n	(5)	(4)
$S\tilde{D}^{-1}$	m^2n	(6)	(3)
Eq.(5.9)	$5mn^2$	(7)	(2)(5)(6)
Total	$\frac{7}{3}m^3 + 3m^2n + 5mn^2$	n/a	n/a

Determining which of the above methods is the best depends on the size of S_L, S_R, B_L, and B_R, the exceptions in the downward pass, the flop rate, the cache miss rate, and the implementation complexity.

5.3 Optimization for symmetry and positive definiteness

In real problems, $\mathbf{A}$ is often symmetric (or Hermitian if the problems are complex, which can be treated in a similar way) [Svizhenko et al., 2002]. So it is worthwhile to consider special treatment for such matrices to reduce both computation cost and storage cost. Note that this reduction is independent of the extra sparsity in Section 5.2.

5.3.1 The symmetry and positive definiteness of dense matrix A

If all the given matrices are symmetric, it is reasonable to expect the elimination results to be symmetric as well, because our update rule (5.1) does not break matrix symmetry. This is shown in the following *property of symmetry preservation.*

Property 5.1 (symmetry preservation) *In* (5.1)*,* $\mathsf{U}, \mathbf{A}(\mathsf{B},\mathsf{B})$*, and* $\mathbf{A}(\mathsf{S},\mathsf{S})$ *are all symmetric;* $\mathbf{A}(\mathsf{B},\mathsf{S})$ *and* $\mathbf{A}(\mathsf{S},\mathsf{B})$ *are transpose of each other.*

Proof: This property holds for all the leaf clusters by the symmetry of the original matrix $\mathbf{A}$. For any node in the cluster tree, if the property holds for its two child clusters, then by (3.3), the property holds for the parent node as well. $\square$

In addition, $\mathbf{A}$ is often positive definite. This property is also preserved during the elimination process in our algorithm, as shown in the following *property of positive-definiteness preservation.*

Property 5.2 (positive-definiteness preservation) *If the original matrix* $\mathbf{A}$ *is positive-definite, then the matrix* $\mathbf{U}$ *in* (5.1) *is always positive-definite.*

Proof: Write $n \times n$ symmetric positive definite matrix $\mathbf{A}$ as $\begin{pmatrix} a_{11} & z^T \\ z & \bar{\mathbf{A}}_{11} \end{pmatrix}$ and let $\tilde{\mathbf{A}}_{11} \overset{def}{=} \bar{\mathbf{A}}_{11} - zz^T/a_{11}$ be the Schur complement of a_{11}. Let $\mathbf{A}^{(0)} \overset{def}{=} \mathbf{A}$, $\mathbf{A}^{(1)} \overset{def}{=} \tilde{\mathbf{A}}_{11}^{(0)}$, $\ldots$, $\mathbf{A}^{(n-1)} \overset{def}{=} \tilde{\mathbf{A}}_{11}^{(n-2)}$. Note that since $\mathbf{A}$ is positive definite, $\mathbf{A}_{11}^{(i)} \neq 0$ and $\tilde{\mathbf{A}}^{i+1}$ is well defined for all $i = 0, \ldots, n-2$. By definition, $\tilde{\mathbf{A}}^{i+1}$ are also all symmetric.

Now we show that $\mathbf{A}^{(1)}$, ..., $\mathbf{A}^{(n-1)}$, are all positive definite. Given any $(n-1)$-vector $y \neq 0$, let $x = \begin{pmatrix} -\frac{z^T y}{a_{11}} & y \end{pmatrix}^T \neq 0$. Since $\mathbf{A}$ is symmetric positive definite, we have

$$\begin{aligned}
0 &< x^T \mathbf{A} x \\
&= \begin{pmatrix} -z^T y/a_{11} & y^T \end{pmatrix} \begin{pmatrix} a_{11} & z^T \\ z & \bar{\mathbf{A}}_{11} \end{pmatrix} \begin{pmatrix} -z^T y/a_{11} \\ y \end{pmatrix} \\
&= \begin{pmatrix} 0 & -y^T z z^T/a_{11} + y^T \bar{\mathbf{A}}_{11} \end{pmatrix} \begin{pmatrix} -z^T y/a_{11} \\ y \end{pmatrix} \\
&= -y^T z z^T y/a_{11} + y^T \bar{A}_{11} y \\
&= y^T (\bar{\mathbf{A}}_{11} - z z^T/a_{11}) y \\
&= y^T \tilde{\mathbf{A}}_{11} y.
\end{aligned}$$

As a result, $\mathbf{A}^{(1)}$ is positive definite as well. Repeating this process, we have $\mathbf{A}^{(2)}$, ..., $\mathbf{A}^{(n-1)}$ all positive definite.

Since any principal submatrix of a positive definite matrix is also positive definite [Golub and Loan, 1996a], every $\mathbf{A}(\mathsf{S}_k, \mathsf{S}_k)$ in (3.2) is also positive definite. □

For the symmetry preservation property and the positive definiteness preservation property, we can write the last term in (5.1) as

$$\mathbf{A}(\mathsf{B}, \mathsf{S}) \mathbf{A}(\mathsf{S}, \mathsf{S})^{-1} \mathbf{A}(\mathsf{S}, \mathsf{B}) = (\mathcal{G}_{\mathsf{S}}^{-1} \mathbf{A}(\mathsf{S}, \mathsf{B}))^T (\mathcal{G}_{\mathsf{S}}^{-1} \mathbf{A}(\mathsf{S}, \mathsf{B})),$$

where $\mathbf{A}(\mathsf{S}, \mathsf{S}) = \mathcal{G}_{\mathsf{S}} \mathcal{G}_{\mathsf{S}}^T$. The Cholesky factorization has cost $\frac{1}{6}\mathsf{s}^3$. The solver has cost $\frac{1}{2}\mathsf{s}^2\mathsf{b}$, and the final multiplication has cost $\frac{1}{2}\mathsf{s}\mathsf{b}^2$. The cost for (5.1) is then reduced by half from $\frac{1}{3}\mathsf{s}^3 + \mathsf{s}^2\mathsf{b} + \mathsf{s}\mathsf{b}^2$ to $\frac{1}{6}\mathsf{s}^3 + \frac{1}{2}\mathsf{s}^2\mathsf{b} + \frac{1}{2}\mathsf{s}\mathsf{b}^2$.

Note that even if $\mathbf{A}$ is not positive definite, by the symmetry preservation property alone, we can still write the last term of (5.1) as

$$\mathbf{A}(\mathsf{B}, \mathsf{S}) \mathbf{A}(\mathsf{S}, \mathsf{S})^{-1} \mathbf{A}(\mathsf{S}, \mathsf{B}) = (\mathbf{L}_{\mathsf{S}}^{-1} \mathbf{A}(\mathsf{S}, \mathsf{B}))^T \mathbf{D}_{\mathsf{S}}^{-1} (\mathbf{L}_{\mathsf{S}}^{-1} \mathbf{A}(\mathsf{S}, \mathsf{B})),$$

where $\mathbf{A}(\mathsf{S}, \mathsf{S}) = \mathbf{L}_{\mathsf{S}} \mathbf{D}_{\mathsf{S}} \mathbf{L}_{\mathsf{S}}^T$.

Similarly, the computation cost is reduced by half. However, this method is subject to large errors due to small pivots, since the straightforward symmetric pivoting does not always work [Golub and Loan, 1996b]. The diagonal pivoting method [Bunch and Parlett, 1971; Bunch and Kaufman, 1977] can be used to solve such a problem, but it is more expensive in both implementation and computation. We will discuss this again in more detail in a future article.

5.3.2 Optimization combined with the extra sparsity

For symmetric and positive definite $\mathbf{A}$, we can take advantage of the extra sparsity discussed in Section 5.2 as well. Consider the following block Cholesky factorization of $\mathbf{A}(\mathsf{S},\mathsf{S})$:

$$\mathbf{A}(\mathsf{S},\mathsf{S}) = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{12}^T & \mathbf{A}_{22} \end{pmatrix} = \begin{pmatrix} \mathcal{G}_1 & 0 \\ \mathbf{A}_{12}^T \mathcal{G}_1^{-T} & \tilde{\mathcal{G}}_2 \end{pmatrix} \begin{pmatrix} \mathcal{G}_1^T & \mathcal{G}_1^{-1}\mathbf{A}_{12} \\ 0 & \tilde{\mathcal{G}}_2^T \end{pmatrix} = \mathcal{G}_{\mathsf{S}}\mathcal{G}_{\mathsf{S}}^T, \tag{5.11}$$

where $\mathbf{A}_{11} = \mathcal{G}_1\mathcal{G}_1^T$, $\tilde{\mathbf{A}}_{22} = \mathbf{A}_{22} - \mathbf{A}_{12}^T\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$, and $\tilde{\mathbf{A}}_{22} = \tilde{\mathcal{G}}_2\tilde{\mathcal{G}}_2^T$. Now, we have

$$\begin{aligned} \mathcal{G}_{\mathsf{S}}^{-1}\mathbf{A}(\mathsf{S},\mathsf{B}) &= \begin{pmatrix} \mathcal{G}_1 & 0 \\ \mathbf{A}_{12}^T\mathcal{G}_1^{-T} & \tilde{\mathcal{G}}_2 \end{pmatrix} \begin{pmatrix} \mathbf{A}_i(\mathsf{S}_k \cap \mathsf{B}_i, \mathsf{B}_k \cap \mathsf{B}_i) & 0 \\ 0 & \mathbf{A}_j(\mathsf{S}_k \cap \mathsf{B}_j, \mathsf{B}_k \cap \mathsf{B}_j) \end{pmatrix} \\ &= \begin{pmatrix} \mathcal{G}_1^{-1}\mathbf{A}(\mathsf{S}_L,\mathsf{B}_L) & 0 \\ -\tilde{\mathcal{G}}_2^{-1}\mathbf{A}_{12}^T\mathcal{G}_1^{-T}\mathcal{G}_1^{-1}\mathbf{A}(\mathsf{S}_L,\mathsf{B}_L) & \tilde{\mathcal{G}}_2^{-1}\mathbf{A}(\mathsf{S}_R,\mathsf{B}_R) \end{pmatrix}. \end{aligned} \tag{5.12}$$

Finally, we compute

$$\mathbf{A}(\mathsf{B},\mathsf{S})\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}\mathbf{A}(\mathsf{S},\mathsf{B}) = (\mathcal{G}_{\mathsf{S}}^{-1}\mathbf{A}(\mathsf{S},\mathsf{B}))^T(\mathcal{G}_{\mathsf{S}}^{-1}\mathbf{A}(\mathsf{S},\mathsf{B})). \tag{5.13}$$

Table 5.8 gives the operations with the total cost $\frac{2}{3}m^3 + 2m^2n + \frac{5}{2}mn^2$. Note that the costs for $\mathcal{G}_1^{-1}\mathbf{A}_{12}$ and $\tilde{\mathbf{A}}_{22}$ are both $\frac{1}{6}m^3$ because $\mathbf{A}_{12}$ is diagonal and $\tilde{\mathbf{A}}_{22}$ is symmetric. Compared to the original cost with a block structure, the cost is reduced by more than half when $m = n$ (reduced by exactly half when $m = \frac{\sqrt{3}}{2}n$). Accidentally, the cost with both optimization (for extra sparsity and symmetry) is 25% of the original

Table 5.8: Operation costs in flops.

Operation	*Cost*
$\mathbf{A}_{11} \to \mathcal{G}_1\mathcal{G}_1^T$	$\frac{1}{6}m^3$
$\mathcal{G}_1^{-1}\mathbf{A}_{12}$	$\frac{1}{6}m^3$
$\tilde{A}_{22}$	$\frac{1}{6}m^3$
$\tilde{A}_{22} \to \tilde{\mathcal{G}}_2\tilde{\mathcal{G}}_2^T$	$\frac{1}{6}m^3$
$\mathcal{G}_1^{-1}\mathbf{A}(\mathsf{S}_L, \mathsf{B}_L)$	$\frac{1}{2}m^2n$
$\mathcal{G}_1^{-T}\mathcal{G}_1^{-1}\mathbf{A}(\mathsf{S}_L, \mathsf{B}_L)$	$\frac{1}{2}m^2n$
$-\tilde{\mathcal{G}}_2^{-1}\mathbf{A}_{12}^T\mathcal{G}_1^{-T}\mathcal{G}_1^{-1}\mathbf{A}(\mathsf{S}_L, \mathsf{B}_L)$	$\frac{1}{2}m^2n$
$\tilde{\mathcal{G}}_2^{-1}\mathbf{A}(\mathsf{S}_R, \mathsf{B}_R)$	$\frac{1}{2}m^2n$
Total	$\frac{2}{3}m^3 + 2m^2n + \frac{5}{2}mn^2$

cost as long as the same condition is satisfied: $m = \frac{\sqrt{3}}{2}n$. (This optimization is all that we can do: neither reordering the nodes nor changing the computation sequence works.)

5.3.3 Storage cost reduction

In addition to the computation cost reduction, the storage cost can also be reduced for symmetric **A**. Since the storage cost is mainly for **U**, where **U** is always symmetric, the storage cost is reduced by half. Another part of the storage cost comes from the temporary space needed for (5.1), which can also be reduced by half. Such space is needed for the top level update in the cluster tree. It is about the same as the cost for **U** but we do not have to keep it. So for a typical cluster tree with ten or more levels, this part is not important. When the mesh size is small with a short cluster tree, this part and its reduction may play a significant role though.

5.4 Optimization for computing $\mathbf{G}^<$ and current density

We can also reduce the cost for computing $\mathbf{G}^<$ by optimizing the procedure for (5.14), which is copied from (5.2):

$$\mathbf{R} = \mathbf{\Sigma}(\mathsf{B},\mathsf{B}) + \mathbf{L}^{-1}\mathbf{\Sigma}(\mathsf{S},\mathsf{B}) + \mathbf{\Sigma}(\mathsf{B},\mathsf{S})\mathbf{L}^{-\dagger} + \mathbf{L}^{-1}\mathbf{\Sigma}(\mathsf{S},\mathsf{S})\mathbf{L}^{-\dagger}. \tag{5.14}$$

The computation is by default based on the computation for $\mathbf{G}$.

5.4.1 $\mathbf{G}^<$ sparsity

Since $\mathbf{\Sigma}(\mathsf{S},\mathsf{B})$ and $\mathbf{\Sigma}(\mathsf{B},\mathsf{S})$ are block diagonal, the cost for the second and the third term in (5.14) is reduced by half. Similar to the structure of $\mathbf{A}(\mathsf{S},\mathsf{S})$ in (5.1), the blocks $\mathbf{\Sigma}(\mathsf{S}_k \cap \mathsf{B}_i, \mathsf{S}_k \cap \mathsf{B}_j)$ and $\mathbf{\Sigma}(\mathsf{S}_k \cap \mathsf{B}_j, \mathsf{S}_k \cap \mathsf{B}_i)$ in $\mathbf{\Sigma}(\mathsf{S}_k,\mathsf{S}_k)$ are diagonal. If we use block matrix multiplication for $\mathbf{L}^{-1}\mathbf{\Sigma}(\mathsf{S},\mathsf{S})$, then these blocks can be ignored because multiplication with them are level-2 operations. As a result, the cost for the fourth term $\mathbf{L}^{-1}\mathbf{\Sigma}(\mathsf{S},\mathsf{S})\mathbf{L}^{-\dagger}$ in (5.14) is reduced to $\frac{1}{2}\mathsf{s}^2\mathsf{b}+\mathsf{s}\mathsf{b}^2$ (or $\mathsf{s}^2\mathsf{b}+\frac{1}{2}\mathsf{s}\mathsf{b}^2$, depending on the order of computation). So the total cost for (5.14) becomes $\frac{1}{2}\mathsf{s}^2\mathsf{b} + 2\mathsf{s}\mathsf{b}^2$ (or $\mathsf{s}^\mathsf{b} + \frac{3}{2}\mathsf{s}\mathsf{b}^2$). Compared with the cost without optimization for sparsity, it is reduced by 37.5%.

In addition, $\mathbf{\Sigma}$ is often diagonal or 3-diagonal in real problems. As a result, $\mathbf{\Sigma}(\mathsf{S}_k \cap \mathsf{B}_i, \mathsf{S}_k \cap \mathsf{B}_j)$ and $\mathbf{\Sigma}(\mathsf{S}_k \cap \mathsf{B}_j, \mathsf{S}_k \cap \mathsf{B}_i)$ become zero and $\mathbf{\Sigma}_k(\mathsf{S}_k,\mathsf{S}_k)$ becomes block diagonal. This may lead to further reduction. Please see the appendix for detailed analysis.

5.4.2 $\mathbf{G}^<$ symmetry

For symmetric $\mathbf{\Sigma}$, we have a similar property of symmetry (and positive definiteness) preservation: $\mathbf{\Sigma}_{k+}(\mathsf{B}_k,\mathsf{B}_k)$, $\mathbf{\Sigma}_k(\mathsf{B}_k,\mathsf{B}_k)$, and $\mathbf{\Sigma}_k(\mathsf{S}_k,\mathsf{S}_k)$ in (4.1) are all symmetric. With such symmetry, we can perform the Cholesky factorization $\mathbf{\Sigma}(\mathsf{S},\mathsf{S}) = \mathbf{\Gamma}_\mathsf{S}\mathbf{\Gamma}_\mathsf{S}^T$. Following (5.12) and (5.13), Table 5.9 lists the needed operations and their costs. The total cost for (4.1) is reduced to $\frac{1}{6}\mathsf{s}^3 + \mathsf{s}^2\mathsf{b} + \frac{3}{2}\mathsf{sb}^2$. Note that we can reuse $\mathbf{A}(\mathsf{B},\mathsf{S})\mathcal{G}_\mathsf{S}^{-T}$ from computing symmetric $\mathbf{G}$ but since $\mathbf{L}^{-1} = \mathbf{A}(\mathsf{B},\mathsf{S})\mathbf{A}(\mathsf{S},\mathsf{S})^{-1}$ is not computed explicitly there, we have to do it here. Also note that $\mathbf{L}^{-1}$ is a full dense matrix so computing $\mathbf{L}^{-1}\mathbf{\Sigma}(\mathsf{S},\mathsf{B})$ requires sb^2 flops. Compared to the cost of computation without exploiting the symmetry ($\mathsf{s}^2\mathsf{b} + 3\mathsf{sb}^2$), the cost is reduced by one third when $\mathsf{s} = \mathsf{b}$.

Table 5.9: Operation costs in flops for $\mathbf{G}^<$ with optimization for symmetry.

Operation	*Result*	*Cost*
Cholesky on $\mathbf{\Sigma}(\mathsf{S},\mathsf{S})$	$\mathbf{\Gamma}_\mathsf{S}$	$\frac{1}{6}\mathsf{s}^3$
$(\mathbf{A}(\mathsf{B},\mathsf{S})\mathcal{G}_\mathsf{S}^{-T})\mathcal{G}_\mathsf{S}^{-1}$	$\mathbf{L}^{-1}$	$\frac{1}{2}\mathsf{s}^2\mathsf{b}$
$\mathbf{L}^{-1}\mathbf{\Sigma}(\mathsf{S},\mathsf{B})$	$\mathbf{L}^{-1}\mathbf{\Sigma}(\mathsf{S},\mathsf{B})$	sb^2
$\mathbf{L}^{-1}\mathbf{\Gamma}_\mathsf{S}$	$\mathbf{L}^{-1}\mathbf{\Gamma}_\mathsf{S}$	$\frac{1}{2}\mathsf{s}^2\mathsf{b}$
$[\mathbf{L}^{-1}\mathbf{\Gamma}_\mathsf{S}][\mathbf{L}^{-1}\mathbf{\Gamma}_\mathsf{S}]^T$	$\mathbf{L}^{-1}\mathbf{\Sigma}(\mathsf{S},\mathsf{S})\mathbf{L}^{-T}$	$\frac{1}{2}\mathsf{sb}^2$
Total	$\mathbf{R}$	$\frac{1}{6}\mathsf{s}^3 + \mathsf{s}^2\mathsf{b} + \frac{3}{2}\mathsf{sb}^2$

5.4.3 Current density

We can also exploit the extra sparsity when computing the current density, but it cannot improve much because there is not as much sparsity in (4.2)–(4.4) as in (5.1). This can been seen in Fig. 5.5.

However, we can significantly reduce the cost with another approach. In this approach, in addition to computing the current density from nodes 9-12, we also compute the current density from nodes 1, 2, 5, and 6. This approach is still based on (4.2)–(4.4) with no additional work, but will reduce the number of needed leaf clusters by half if we choose properly which clusters to compute. Fig. 5.6 shows how we choose the leaf clusters to cover the desired off-diagonal entries in the inverse for current density.

Fig. 5.6 shows only the coverage of current in x-direction, but the approach works for the current in y-direction, and more nicely, the same tiling works for the current

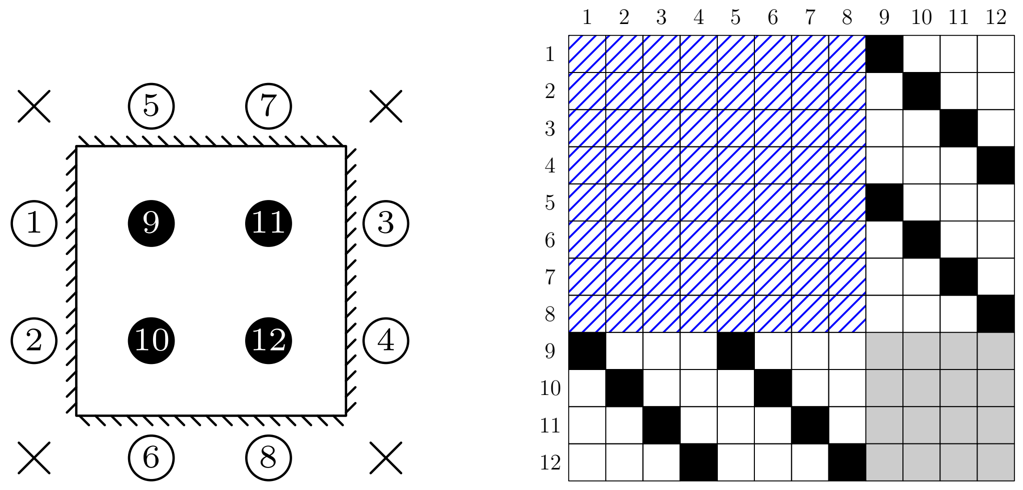


Figure 5.5: The extra sparsity of the matrices used for computing the current density.

in both directions. Readers can check the coverage on their own by drawing arrows between neighboring nodes in the y-direction to see if they are covered by the blue areas in Fig. 5.6 shaded in blue and red.

We can also optimize (4.2)–(4.4) by similar techniques used in Section 5.3 to reduce the cost for symmetric $\mathbf{A}$ and $\mathbf{\Sigma}$. In (4.2), $\mathbf{A}(\mathsf{C},\mathsf{C})-\mathbf{A}(\mathsf{C},\mathsf{B}_{\bar{\mathsf{C}}})(\mathbf{U}_{\bar{\mathsf{C}}})^{-1}\mathbf{A}(\mathsf{B}_{\bar{\mathsf{C}}},\mathsf{C})$ is of the same form as (5.1) so the cost is reduced by half. The cost of computing its inverse is also reduced by half when it is symmetric. Eqs. (4.3) and (4.4) are the transpose of each other so we only need to compute one of them. As a result, the cost for computing the current density is reduce by half in the presence of symmetry.

5.5 Results and comparison

Fig. 5.7 reveals the computation cost reduction for clusters of different size in the basic cluster tree after optimization for extra sparsity during the upward pass. As indicated in section 5.2, the improvements are different for two types of merge: i) two $a\times a$ clusters $\Rightarrow$ one $a\times 2a$ cluster; ii) two $2a\times a$ clusters $\Rightarrow$ one $2a\times 2a$ cluster, so they are shown separately in the figure. Note that the reduction for small clusters is low. This is because the extra second order computation cost (e.g., $\mathcal{O}(m^2)$) introduced in the optimization for extra sparsity is ignored in our analysis but is significant for

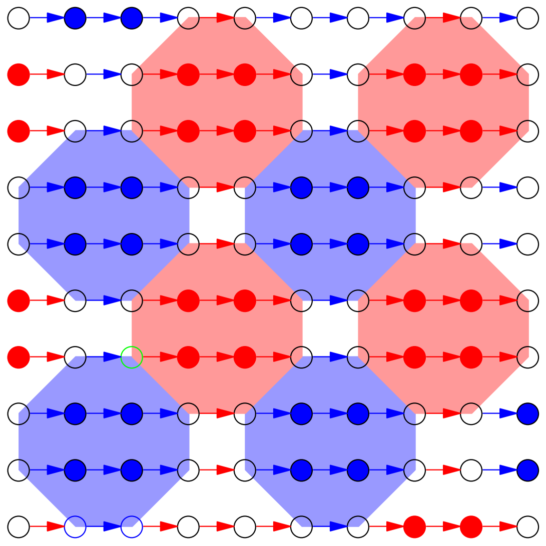


Figure 5.6: Coverage of all the needed current density through half of the leaf clusters. Arrows are the current. Each needed cluster consists of four solid nodes. The circle nodes are skipped.

small clusters.

Fig. 5.8 reveals the computation cost reduction after optimization for symmetry. As in Fig. 5.7, we show the reduction for the merge for clusters at each level in the basic cluster tree. We see that the cost is reduced almost by half for clusters larger than 32×32. The low reduction for small clusters here is mainly due to a larger portion of diagonal entries for small matrices where Cholesky factorization cannot reduce the cost.

Although the cost reduction is low for small clusters in both optimizations, they do not play a significant role for a mesh larger than 32×32 because the top levels of the tree in Fig. 3.5 dominate the computation cost.

Fig. 5.9 compares the computation cost of RGF and optimized FIND on Hermitian and positive-definite $\mathbf{A}$ and reveals the overall effect of the optimization for extra sparsity and symmetry. The simulations were performed on a square mesh of size $N \times N$. The cross-point is around 40×40 according to the two fitted lines, which is

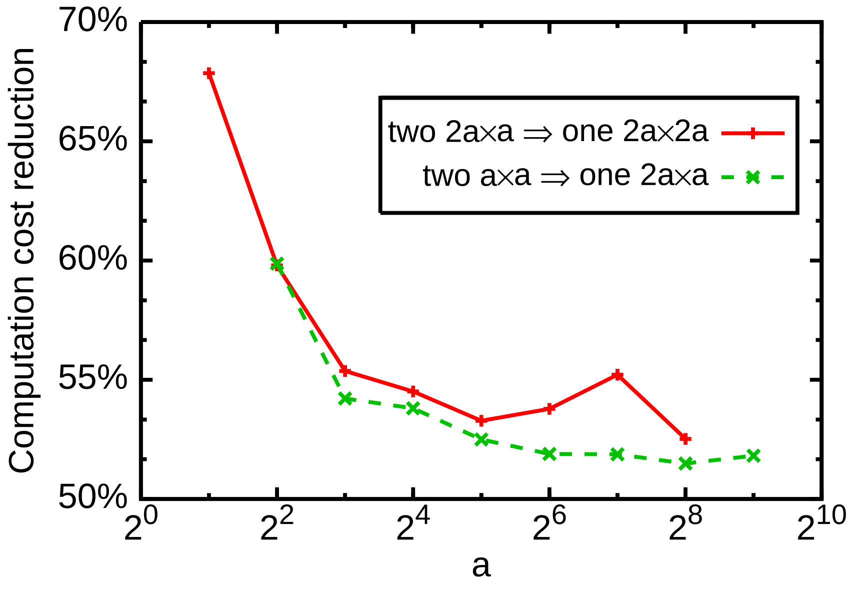


Figure 5.7: Optimization for extra sparsity

also confirmed by a simulation at that point.

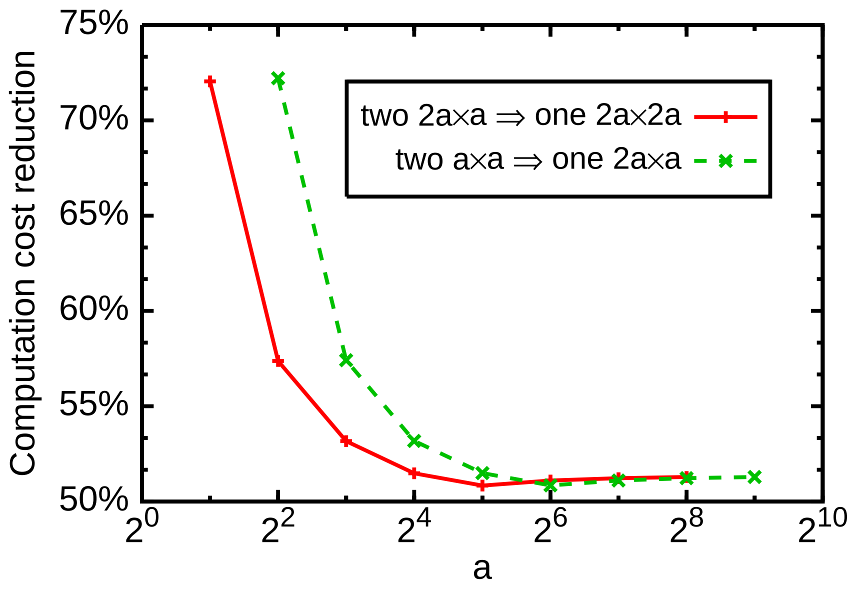


Figure 5.8: Optimization for symmetry

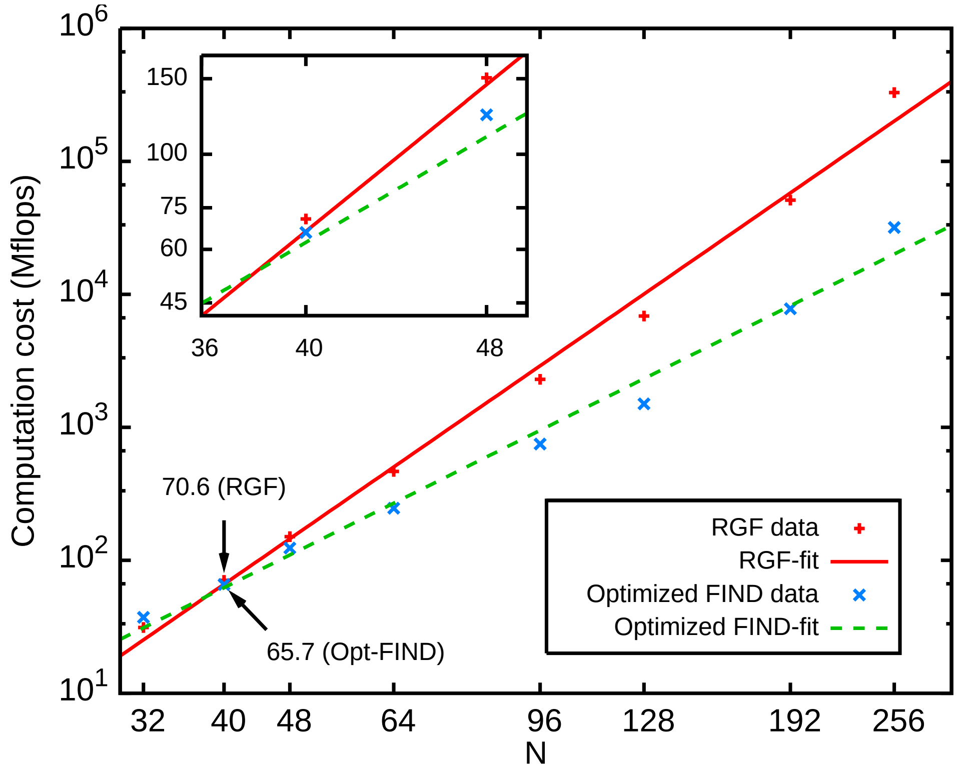


Figure 5.9: Comparison with RGF to show the change of cross-point

Chapter 6

FIND-2way and Its Parallelization

This chapter is organized as follows. We first describe general relations to compute Green's functions (Section 6.1 and 6.2). These are applicable to meshes with arbitrary connectivity. In Section 6.3, 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 the algorithm introduced in Chapter 3.

In Section 6.4, a parallel implementation of the recurrences for 1D problems is proposed. The original algorithms by [Takahashi et al., 1973] and [Svizhenko et al., 2002] are not parallel because 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, [Hageman and Varga, 1964]). Section 6.5 has some numerical results. For notational simplicity, we let $\mathbf{G} = \mathbf{G}^r$ in this chapter.

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

6.1 Recurrence formulas for the inverse matrix

Consider a general matrix $\mathbf{A}$ written in block form with $\mathbf{A}_1 1$ nonsingular:

$$\mathbf{A} \overset{\text{def}}{=} \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix}. \tag{6.1}$$

From the LDU factorization of $\mathbf{A}$, we can form the factors $\mathbf{L}^{\mathrm{S}} \overset{\text{def}}{=} \mathbf{A}_{21}\mathbf{A}_{11}^{-1}$, $\mathbf{U}^{\mathrm{S}} \overset{\text{def}}{=} \mathbf{A}_{11}^{-1}\mathbf{A}_{12}$, 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 $\mathbf{G} = \mathbf{A}^{-1}$:

$$\mathbf{G} = \begin{pmatrix} \mathbf{A}_{11}^{-1} + \mathbf{U}^{\mathrm{S}}\,\mathbf{S}^{-1}\,\mathbf{L}^{\mathrm{S}} & -\mathbf{U}^{\mathrm{S}}\,\mathbf{S}^{-1} \\ -\mathbf{S}^{-1}\,\mathbf{L}^{\mathrm{S}} & \mathbf{S}^{-1} \end{pmatrix}. \tag{6.2}$$

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:

$$\begin{aligned} \mathbf{A} = & \begin{pmatrix} \mathbf{L}(1:i-1,1:i-1) & \mathbf{0} \\ \mathbf{L}(i:n,1:i-1) & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{D}(1:i-1,1:i-1) & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^i \end{pmatrix} \\ & \times \begin{pmatrix} \mathbf{U}(1:i-1,1:i-1) & \mathbf{U}(1:i-1,i:n) \\ \mathbf{0} & \mathbf{I} \end{pmatrix}. \end{aligned} \tag{6.3}$$

The Schur complement matrices $\mathbf{S}^i$ are defined by this equation for all i. From (6.3), the first step of the LDU factorization of $\mathbf{S}^i$ is the same as the $i+1^{\text{th}}$ step in the factorization of $\mathbf{A}$:

$$\mathbf{S}^i = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{L}(i+1:n,i) & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{d}_{ii} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^{i+1} \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{U}(i,i+1:n) \\ \mathbf{0} & \mathbf{I} \end{pmatrix}. \tag{6.4}$$

Combining (6.2) and (6.4) with the substitutions:

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

we arrive at

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

From (6.2), we have

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

We have therefore proved the following backward recurrence relations starting from $\mathbf{g}_{nn} = \mathbf{d}_{nn}^{-1}$:

$$\boxed{\begin{aligned} \mathbf{G}(i+1:n, i) &= -\mathbf{G}(i+1:n, i+1:n)\,\mathbf{L}(i+1:n, i) \\ \mathbf{G}(i, i+1:n) &= -\mathbf{U}(i, i+1:n)\,\mathbf{G}(i+1:n, i+1:n) \\ \mathbf{g}_{ii} &= \mathbf{d}_{ii}^{-1} + \mathbf{U}(i,i+1:n)\,\mathbf{G}(i+1:n, i+1:n)\,\mathbf{L}(i+1:n, i) \end{aligned}} \tag{6.5}$$

These equations are identical to Equations (2.4), (2.5), and (2.6), respectively. A recurrence for $i = n-1$ down to $i = 1$ can therefore be used to calculate $\mathbf{G}$. See Figure 6.1.

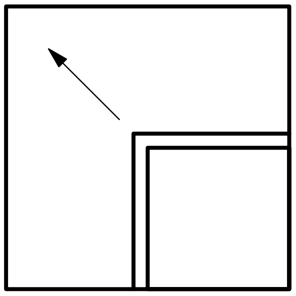


Figure 6.1: Schematics of how the recurrence formulas are used to calculate $\mathbf{G}$.

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 $\mathbf{G}$ is full, we do not need to calculate all the entries. We denote $\mathbf{L}^{\text{sym}}$ and $\mathbf{U}^{\text{sym}}$ the lower and upper triangular factors in the symbolic factorization of $\mathbf{A}$. The non-zero structure of $(\mathbf{L}^{\text{sym}})^T$ and $(\mathbf{U}^{\text{sym}})^T$ is denoted by Q, that is Q is the set of all pairs (i, j) such that $\boldsymbol{\ell}_{ji}^{\text{sym}} \neq \mathbf{0}$ or $\mathbf{u}_{ji}^{\text{sym}} \neq \mathbf{0}$. Then

$$\mathbf{g}_{ij} = \begin{cases} -\displaystyle\sum_{l>j,\,(i,l)\in Q} \mathbf{g}_{il}\,\boldsymbol{\ell}_{lj} & \text{if } i > j \\ -\displaystyle\sum_{k>i,\,(k,j)\in Q} \mathbf{u}_{ik}\,\mathbf{g}_{kj} & \text{if } i < j \\ \mathbf{d}_{ii}^{-1} + \displaystyle\sum_{k>i,\,l>i,\,(k,l)\in Q} \mathbf{u}_{ik}\,\mathbf{g}_{kl}\,\boldsymbol{\ell}_{li} & \text{if } i = j \end{cases}, \tag{6.6}$$

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

***Proof**:* Assume that $(i, j) \in Q$, $i > j$, then $\mathbf{u}_{ji}^{\text{sym}} \neq 0$. Assume also that $\boldsymbol{\ell}_{lj}^{\text{sym}} \neq 0$, $l > j$, then by the properties of 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 $\mathbf{u}_{ik}^{\text{sym}} \neq 0$ and $\boldsymbol{\ell}_{li}^{\text{sym}} \neq 0$, then by the properties of Gaussian elimination, $(k, l) \in Q$. □

This implies that we do not need to calculate all the entries in $\mathbf{G}$ but rather only the entries in $\mathbf{G}$ corresponding to indices in Q. This represents a significant reduction in computational cost. The cost of computing the entries of $\mathbf{G}$ 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

$$\mathbf{G}^< = \mathbf{G}\,\boldsymbol{\Gamma}\,\mathbf{G}^\dagger, \tag{6.7}$$

if we assume that $\boldsymbol{\Gamma}$ has the same sparsity pattern as $\mathbf{A}$, that is, $\boldsymbol{\gamma}_{ij} \neq 0 \Rightarrow \mathbf{a}_{ij} \neq 0$.

The block $\boldsymbol{\gamma}_{ij}$ denotes the block (i,j) of matrix $\mathbf{\Gamma}$.

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

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

Using the same block notations as on page 84, we multiply this equation on the left by $(\mathbf{LD})^{-1}$ and on the right by $(\mathbf{LD})^{-\dagger}$:

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

The equation above can be expanded as

$$\begin{pmatrix} \mathbf{G}^<_{11} + \mathbf{U}^{\mathrm{S}}\mathbf{G}^<_{21} + \mathbf{G}^<_{12}(\mathbf{U}^{\mathrm{S}})^\dagger + \mathbf{U}^{\mathrm{S}}\mathbf{G}^<_{22}(\mathbf{U}^{\mathrm{S}})^\dagger & (\mathbf{G}^<_{12} + \mathbf{U}^{\mathrm{S}}\mathbf{G}^<_{22})\mathbf{S}^\dagger \\ \mathbf{S}(\mathbf{G}^<_{21} + \mathbf{G}^<_{22}(\mathbf{U}^{\mathrm{S}})^\dagger) & \mathbf{S}\mathbf{G}^<_{22}\mathbf{S}^\dagger \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11}^{-1}\mathbf{\Gamma}_{11}\mathbf{A}_{11}^{-\dagger} & \mathbf{A}_{11}^{-1}(\mathbf{\Gamma}_{12} - \mathbf{\Gamma}_{11}(\mathbf{L}^{\mathrm{S}})^\dagger) \\ (\mathbf{\Gamma}_{21} - \mathbf{L}^{\mathrm{S}}\mathbf{\Gamma}_{11})\mathbf{A}_{11}^{-\dagger} & \mathbf{\Gamma}_{22} - \mathbf{L}^{\mathrm{S}}\mathbf{\Gamma}_{12} - \mathbf{\Gamma}_{21}(\mathbf{L}^{\mathrm{S}})^\dagger + \mathbf{L}^{\mathrm{S}}\mathbf{\Gamma}_{11}(\mathbf{L}^{\mathrm{S}})^\dagger \end{pmatrix}. \tag{6.8}$$

Using steps similar to the proof of (6.5) and (6.8) 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 $1 \le i \le n-1$,

$$\mathbf{\Gamma}_{\mathrm{L}}^{i+1} \overset{\text{def}}{=} (\mathbf{\Gamma}^i_{21} - \mathbf{L}(i+1:n,i)\,\boldsymbol{\gamma}^i_{11})\,\mathbf{d}_{ii}^{-\dagger}, \tag{6.9}$$

$$\mathbf{\Gamma}_{\mathrm{U}}^{i+1} \overset{\text{def}}{=} \mathbf{d}_{ii}^{-1}\,(\mathbf{\Gamma}^i_{12} - \boldsymbol{\gamma}^i_{11}\,\mathbf{L}(i+1:n,i)^\dagger), \tag{6.10}$$

$$\begin{aligned} \mathbf{\Gamma}^{i+1} \overset{\text{def}}{=} {} & \mathbf{\Gamma}^i_{22} - \mathbf{L}(i+1:n,i)\,\mathbf{\Gamma}^i_{12} - \mathbf{\Gamma}^i_{21}\,\mathbf{L}(i+1:n,i)^\dagger \\ & + \mathbf{L}(i+1:n,i)\,\boldsymbol{\gamma}^i_{11}\,\mathbf{L}(i+1:n,i)^\dagger, \end{aligned} \tag{6.11}$$

with

$$\begin{pmatrix} \boldsymbol{\gamma}^i_{11} & \mathbf{\Gamma}^i_{12} \\ \mathbf{\Gamma}^i_{21} & \mathbf{\Gamma}^i_{22} \end{pmatrix} \overset{\text{def}}{=} \mathbf{\Gamma}^i,$$

and the following block sizes (in terms of number of blocks of the sub-matrix): Note that the $\mathbf{U}$ factors are not needed in this process. Once this recurrence is complete, we can perform a backward recurrence to find $\mathbf{G}^<$, as in (6.5). This recurrence can be proved using (6.8):

$$\boxed{\begin{aligned}
\mathbf{G}^<(i+1:n,i) &= \mathbf{G}(i+1:n,i+1:n)\,\mathbf{\Gamma}_{\mathrm{L}}^{i+1} \\
&\quad - \mathbf{G}^<(i+1:n,i+1:n)\,\mathbf{U}(i,i+1:n)^\dagger \\
\mathbf{G}^<(i,i+1:n) &= \mathbf{\Gamma}_{\mathrm{U}}^{i+1}\,\mathbf{G}(i+1:n,i+1:n)^\dagger \\
&\quad - \mathbf{U}(i,i+1:n)\,\mathbf{G}^<(i+1:n,i+1:n) \\
\mathbf{g}_{ii}^< &= \mathbf{d}_{ii}^{-1}\,\boldsymbol{\gamma}_{11}^{i}\,\mathbf{d}_{ii}^{-\dagger} \\
&\quad - \mathbf{U}(i,i+1:n)\,\mathbf{G}^<(i+1:n,i) \\
&\quad - \mathbf{G}^<(i,i+1:n)\,\mathbf{U}(i,i+1:n)^\dagger \\
&\quad - \mathbf{U}(i,i+1:n)\,\mathbf{G}^<(i+1:n,i+1:n)\,\mathbf{U}(i,i+1:n)^\dagger
\end{aligned}} \tag{6.12}$$

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 (6.5).

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 assume further that $(i,j) \in Q \Leftrightarrow (j,i) \in Q$.

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 (6.11) leads to the same fill-in as the Schur complement calculations for $\mathbf{S}^{i+1}$.

Second, the set of (6.12) for $\mathbf{G}^<$ can be simplified to

$$\mathbf{g}^<_{ij} = \begin{cases} \displaystyle\sum_{l>j,\,(i,l)\in Q} \mathbf{g}_{il}\,[\boldsymbol{\gamma}_{\mathrm{L}}^{j+1}]_{l-j,1} - \sum_{l>j,\,(i,l)\in Q} \mathbf{g}^<_{il}\,\mathbf{u}^\dagger_{jl} & \text{if } i > j \\ \displaystyle\sum_{k>i,\,(j,k)\in Q} [\boldsymbol{\gamma}_{\mathrm{U}}^{i+1}]_{1,k-i}\,\mathbf{g}^\dagger_{jk} - \sum_{k>i,\,(k,j)\in Q} \mathbf{u}_{ik}\,\mathbf{g}^<_{kj} & \text{if } i < j \\ \displaystyle\mathbf{d}_{ii}^{-1}\boldsymbol{\gamma}^i_{11}\mathbf{d}_{ii}^{-\dagger} - \sum_{k>i,\,(k,i)\in Q} \mathbf{u}_{ik}\,\mathbf{g}^<_{ki} - \sum_{k>i,\,(i,k)\in Q} \mathbf{g}^<_{ik}\,\mathbf{u}^\dagger_{ik} & \\ \qquad \displaystyle - \sum_{k>i,\,l>i,\,(k,l)\in Q} \mathbf{u}_{ik}\,\mathbf{g}^<_{kl}\,\mathbf{u}^\dagger_{il} & \text{if } i = j \end{cases}, \tag{6.13}$$

where $(i,j) \in Q$. (Notation clarification: $\boldsymbol{\gamma}^i_{jl}$, $[\boldsymbol{\gamma}^i_{\mathrm{L}}]_{jl}$, and $[\boldsymbol{\gamma}^i_{\mathrm{U}}]_{jl}$ are respectively blocks of $\boldsymbol{\Gamma}^i$, $\boldsymbol{\Gamma}^i_{\mathrm{L}}$, and $\boldsymbol{\Gamma}^i_{\mathrm{U}}$.) The proof of this result is similar to the one for (6.6) for $\mathbf{G}$. The reduction in computational cost is realized because all the sums are restricted to indices in Q.

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

6.2 Sequential algorithm for 1D problems

In this section, we present a sequential implementation of the relations presented above, for 1D problems. Section 6.4 will discuss the parallel implementation.

In 1D problems, the matrix $\mathbf{A}$ assumes a block tri-diagonal form. The LDU factorization is obtained using the following recurrences:

$$\boldsymbol{\ell}_{i+1,i} = \mathbf{a}_{i+1,i}\,\mathbf{d}_{ii}^{-1} \tag{6.14}$$

$$\mathbf{u}_{i,i+1} = \mathbf{d}_{ii}^{-1}\,\mathbf{a}_{i,i+1} \tag{6.15}$$

$$\begin{aligned} \mathbf{d}_{i+1,i+1} &= \mathbf{a}_{i+1,i+1} - \mathbf{a}_{i+1,i}\,\mathbf{d}_{ii}^{-1}\,\mathbf{a}_{i,i+1} \\ &= \mathbf{a}_{i+1,i+1} - \boldsymbol{\ell}_{i+1,i}\,\mathbf{a}_{i,i+1} \\ &= \mathbf{a}_{i+1,i+1} - \mathbf{a}_{i+1,i}\,\mathbf{u}_{i,i+1}. \end{aligned} \tag{6.16}$$

From (6.6), the backward recurrences for $\mathbf{G}$ are given by

$$\begin{aligned}
\mathbf{g}_{i+1,i} &= -\mathbf{g}_{i+1,i+1}\,\boldsymbol{\ell}_{i+1,i} && (6.17)\\
\mathbf{g}_{i,i+1} &= -\mathbf{u}_{i,i+1}\,\mathbf{g}_{i+1,i+1} && (6.18)\\
\mathbf{g}_{ii} &= \mathbf{d}_{ii}^{-1} + \mathbf{u}_{i,i+1}\,\mathbf{g}_{i+1,i+1}\,\boldsymbol{\ell}_{i+1,i} && (6.19)\\
&= \mathbf{d}_{ii}^{-1} - \mathbf{g}_{i,i+1}\,\boldsymbol{\ell}_{i+1,i}\\
&= \mathbf{d}_{ii}^{-1} - \mathbf{u}_{i,i+1}\,\mathbf{g}_{i+1,i},
\end{aligned}$$

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 (6.9), (6.10), and (6.11), we need to calculate

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

starting from $\boldsymbol{\gamma}^1 \overset{\mathrm{def}}{=} \boldsymbol{\gamma}_{11}$ (the (1,1) block in matrix $\boldsymbol{\Gamma}$); $\boldsymbol{\gamma}^i$, $\boldsymbol{\gamma}_{\mathrm{L}}^i$, and $\boldsymbol{\gamma}_{\mathrm{U}}^i$ are 1×1 blocks. For simplicity, we have shortened the notations. The blocks $\boldsymbol{\gamma}^i$, $\boldsymbol{\gamma}_{\mathrm{L}}^i$, and $\boldsymbol{\gamma}_{\mathrm{U}}^i$ 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 $\mathbf{L}$, $\mathbf{U}$, $\mathbf{G}$, $\boldsymbol{\gamma}^i$, $\boldsymbol{\gamma}_{\mathrm{L}}^i$, and $\boldsymbol{\gamma}_{\mathrm{U}}^i$ have been computed, the backward recurrence (6.12) for $\mathbf{G}^<$ can be computed:

$$\begin{aligned}
\mathbf{g}_{i+1,i}^< &= \mathbf{g}_{i+1,i+1}\,\boldsymbol{\gamma}_{\mathrm{L}}^{i+1} - \mathbf{g}_{i+1,i+1}^<\,\mathbf{u}_{i,i+1}^{\dagger}\\
\mathbf{g}_{i,i+1}^< &= \boldsymbol{\gamma}_{\mathrm{U}}^{i+1}\,\mathbf{g}_{i+1,i+1}^{\dagger} - \mathbf{u}_{i,i+1}\,\mathbf{g}_{i+1,i+1}^<\\
\mathbf{g}_{ii}^< &= \mathbf{d}_{ii}^{-1}\,\boldsymbol{\gamma}^i\,\mathbf{d}_{ii}^{-\dagger} - \mathbf{u}_{i,i+1}\,\mathbf{g}_{i+1,i}^< - \mathbf{g}_{i,i+1}^<\,\mathbf{u}_{i,i+1}^{\dagger} - \mathbf{u}_{i,i+1}\,\mathbf{g}_{i+1,i+1}^<\,\mathbf{u}_{i,i+1}^{\dagger},
\end{aligned}$$

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

These recurrences are the most efficient way to calculate $\mathbf{G}$ and $\mathbf{G}^<$ on a sequential computer. On a parallel computer, this approach is of limited use because there is little room for parallelization. In section 6.4 we describe a scalable algorithm to perform the same calculations in a truly parallel fashion.

6.3 Computational cost analysis

In this section, we determine the computational cost of the sequential algorithms for 1D and 2D Cartesian meshes with nearest neighbor stencils. From the recurrence relations (6.6) and (6.13), one can prove that the computational cost of calculating $\mathbf{G}$ 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 [George, 1973], we obtain the following costs for Cartesian grids assuming a local discretization stencil. We now do a more detailed analysis of the computational cost and a comparison with the FIND algorithm of Li et al. [Li et al., 2008]. 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 (6.6), 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 [George, 1973] and FIND algorithm [Li et al., 2008], we decompose the mesh in a hierarchical way. See Fig. 6.2. First we split the mesh into two parts (see the upper and lower parts of the mesh in the right panel of Fig. 6.2). 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) $\mathbf{a}_{ij} = 0$ 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. 6.2, clusters 17–31 are all separators [George, 1973]. 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.

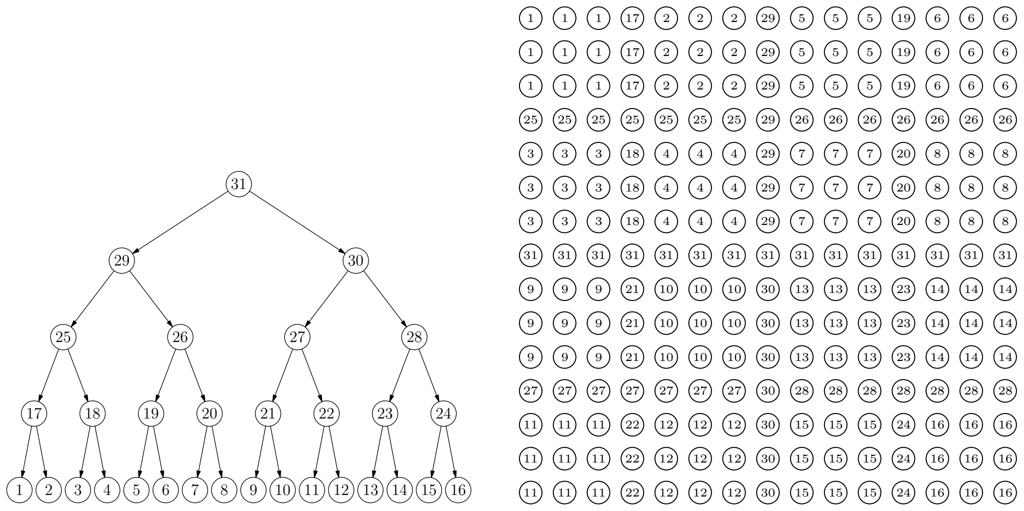


Figure 6.2: The cluster tree (left) and the mesh nodes (right) for mesh of size 15×15. The number in each tree node indicates the cluster number. The number in each mesh node indicates the cluster it belongs to.

Compared to nested dissection [George, 1973], we use vertical and horizontal separators instead of cross-shaped separators to make the estimates of computational cost easier to derive. Compared to FIND-1way, we use single separators $|$ here instead of double separators $\|$ because we do not need additional "independence" among clusters here. Such additional "independence" can be skipped in FIND-1way as well by separating each step of elimination into two part. We will discuss this in our future work.

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. 6.2. For more details, see [George, 1973] and [Li et al., 2008].

Once the separators and boundaries are determined, we see that the computational cost of eliminating a separator is equal to $\mathsf{sb}^2 + 2\mathsf{s}^2\mathsf{b} + \frac{1}{3}\mathsf{s}^3 = \mathsf{s}(\mathsf{b}+\mathsf{s})^2 - \frac{2}{3}\mathsf{s}^3$ 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 into 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 6.1 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 $24.9\,N^{3/2}$ 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

Table 6.1: Estimate of the computational cost for a 2D square mesh for different cluster sizes. The size is in units of $N^{1/2}$. The cost is in units of $N^{3/2}$ flops.

Size of cluster		Cost per cluster			
Separator	Boundary	LDU	Back. Recurr.	Level	Number of clusters
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 ...	≤ 64
1/16	3/8	0.012	0.031	9 ...	≤ 128

in each row increases from b to $\mathsf{s}+\mathsf{b}$. As a result, the cost for computing (6.6) is $3(\mathsf{b}^2\mathsf{s}+\mathsf{b}\mathsf{s}^2+\frac{1}{3}\mathsf{s}^3)$ flops for each step. The total computational cost for the backward recurrence is then $61.3\,N^{3/2}$ flops. The costs for each type of clusters are also listed in Table 6.1.

Adding together the cost of the LDU factorization and the backward recurrence, we see that the total computational cost for the algorithm is $86.2\,N^{3/2}$ flops. For FIND, the cost is $147\,N^{3/2}$ flops [Li and Darve, 2009]. 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.

We now focus on the implementation of these recurrence relations for the calculation of $\mathbf{G}$ 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 discussed in Chapter 7.

6.4 Parallel algorithm for 1D problems

We present a parallel algorithm for the calculation of the Green's function matrix $\mathbf{G}$ typically encountered in electronic structure calculations where the matrix $\mathbf{A}$ (cf. Eq. (1.13)) is assumed to be an $n \times n$ block matrix, and in block tridiagonal form as shown by

$$\mathbf{A} = \begin{pmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} & & & \\ \mathbf{a}_{21} & \mathbf{a}_{22} & \mathbf{a}_{23} & & \\ & \mathbf{a}_{32} & \mathbf{a}_{33} & \mathbf{a}_{34} & \\ & & \ddots & \ddots & \ddots \end{pmatrix}, \tag{6.20}$$

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 $\mathbf{A}$ is then distributed among these processes in a *row-striped* manner, as illustrated in Fig. 6.3.

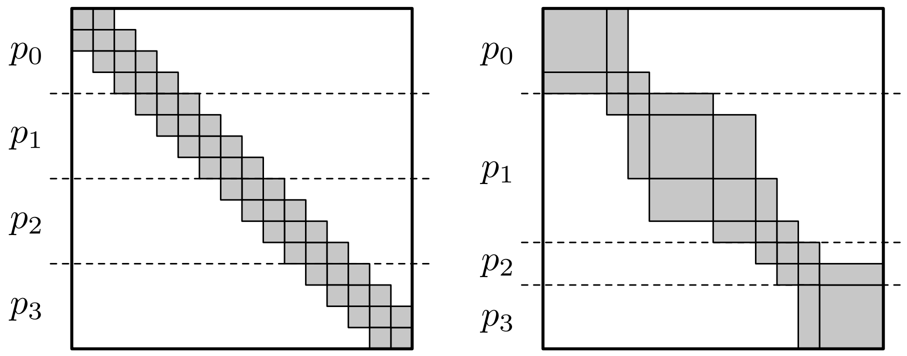


Figure 6.3: Two different block tridiagonal matrices distributed among 4 different processes, labeled p_0, p_1, p_2 and p_3.

Thus each process is assigned ownership of certain contiguous rows of $\mathbf{A}$. This ownership arrangement, however, also extends to the calculated blocks of the inverse

$\mathbf{G}$, as well as any LU factors determined during calculation of $\mathbf{L}$ and $\mathbf{U}$. The manner of distribution is an issue of load balancing, and will be addressed later in the chapter.

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 $\mathbf{A}$, and can be applied to a block tridiagonal matrix as presented on the right in Fig. 6.3. The size of the blocks throughout $\mathbf{A}$, however, does have consequences for adequate load balancing.

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

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 $\text{Trid}_{\mathbf{A}}\left\{\mathbf{G}\right\}$ are outlined in Fig. 6.4.

The algorithm begins with our block tridiagonal matrix $\mathbf{A}$ 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 $\mathbf{G}^{\text{BCR}} = \text{Trid}_{\mathbf{A}^{\text{Schur}}}\left\{\mathbf{G}\right\}$ in what we call the block cyclic production phase. This is done using (6.5). Finally, using $\mathbf{G}^{\text{BCR}}$, we can determine the full tridiagonal structure of $\mathbf{G}$ 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 (6.5).

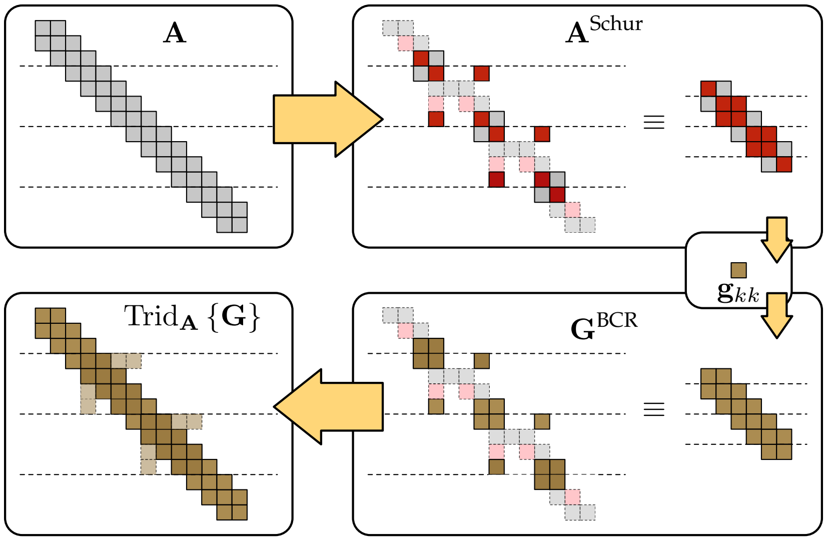

$\mathbf{A}$
$\mathbf{A}^{\text{Schur}}$
$\equiv$
$\mathbf{g}_{kk}$
$\mathbf{G}^{\text{BCR}}$
$\equiv$
$\text{Trid}_{\mathbf{A}}\{\mathbf{G}\}$

Figure 6.4: The distinct phases of operations performed by the hybrid method in determining $\text{Trid}_{\mathbf{A}}\{\mathbf{G}\}$, the block tridiagonal portion of $\mathbf{G}$ with respect to the structure of $\mathbf{A}$. The block matrices in this example are partitioned across 4 processes, as indicated by the horizontal dashed lines.

6.4.1 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 serve to illustrate how our hybrid method is equivalent to unpivoted block Gaussian elimination.

Corner Block Operations

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

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

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

$$\begin{aligned}\boldsymbol{\ell}_{ji} &= \mathbf{a}_{ji}\mathbf{a}_{ii}^{-1}\\ \mathbf{u}_{ij} &= \mathbf{a}_{ii}^{-1}\mathbf{a}_{ij},\end{aligned}$$

and we determine the Schur block

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

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

$$\begin{aligned}\mathbf{g}_{ji} &= -\mathbf{g}_{jj}\boldsymbol{\ell}_{ji}\\ \mathbf{g}_{ij} &= -\mathbf{u}_{ij}\mathbf{g}_{jj},\end{aligned}$$

(see (6.5)) and finally

$$\begin{aligned}\mathbf{g}_{ii} &= \mathbf{a}_{ii}^{-1} + \mathbf{u}_{ij}\mathbf{g}_{jj}\boldsymbol{\ell}_{ji} && (6.21)\\ &= \mathbf{a}_{ii}^{-1} - \mathbf{u}_{ij}\mathbf{g}_{ji} && (6.22)\\ &= \mathbf{a}_{ii}^{-1} - \mathbf{g}_{ij}\boldsymbol{\ell}_{ji}. && (6.23)\end{aligned}$$

This production step is visualized in Fig. 6.5. At the top right of the figure we have the stored LU factors preserved from the BCR elimination phase. At 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 $\boldsymbol{\ell}_{ji}$, corresponding to (6.21), (6.22) and (6.23). 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 which matrix blocks are necessary to calculate

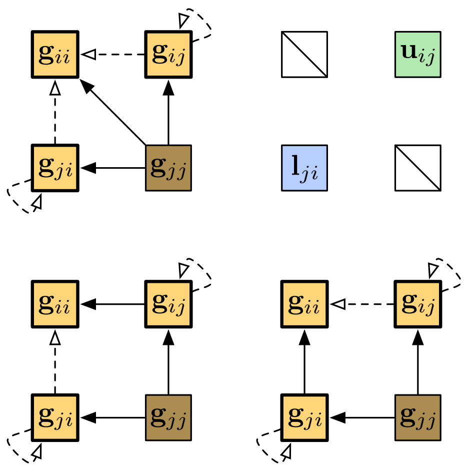


Figure 6.5: Three different schemes that represent a corner production step undertaken in the BCR production phase, where we produce inverse blocks on row/column i using inverse blocks and LU factors from row/column j.

one of the desired inverse blocks. For 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, because the block can be determined via any of the three equations (6.21), (6.22), or (6.23), corresponding respectively to the upper left, lower right, and lower left schemes in Fig. 6.5.

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. (6.23)) beause 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. (6.21)) or bottom right scheme (cf. Eq. (6.22)) where $\mathbf{g}_{jj}$ and $\boldsymbol{\ell}_{ji}$ 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. (6.22)) is preferable to the top left scheme (cf. Eq. (6.21)) because it saves an extra matrix-matrix multiplication. This motivates our choice of using Eq. (6.22) for our hybrid method.

Center Block Operations

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

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

Through a block permutation matrix $\mathbf{P}$, we can transform it to the form

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

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:

$$\text{Part of the L factor (column } j\text{):} \quad \begin{pmatrix} \mathbf{a}_{ij} \\ \mathbf{a}_{kj} \end{pmatrix} \mathbf{a}_{jj}^{-1} = \begin{pmatrix} \boldsymbol{\ell}_{ij} \\ \boldsymbol{\ell}_{kj} \end{pmatrix} \tag{6.24}$$

$$\text{Part of the U factor (row } j\text{):} \quad \mathbf{a}_{jj}^{-1} \big(\mathbf{a}_{ji}\ \mathbf{a}_{jk}\big) = \big(\mathbf{u}_{ji}\ \mathbf{u}_{jk}\big). \tag{6.25}$$

This leads us to the 2×2 block Schur matrix

$$\begin{pmatrix} \mathbf{a}_{ii} & \mathbf{0} \\ \mathbf{0} & \mathbf{a}_{kk} \end{pmatrix} - \begin{pmatrix} \boldsymbol{\ell}_{ij} \\ \boldsymbol{\ell}_{kj} \end{pmatrix} \big(\mathbf{a}_{ji}\ \mathbf{a}_{jk}\big) = \begin{pmatrix} \mathbf{a}_{ii} - \boldsymbol{\ell}_{ij}\mathbf{a}_{ji} & -\boldsymbol{\ell}_{ij}\mathbf{a}_{jk} \\ -\boldsymbol{\ell}_{kj}\mathbf{a}_{ji} & \mathbf{a}_{kk} - \boldsymbol{\ell}_{kj}\mathbf{a}_{jk} \end{pmatrix}. \tag{6.26}$$

Let us assume that we have now computed the following blocks of the inverse $\mathbf{G}$:

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

With this information, we can use the stored LU factors to determine the other blocks of the inverse (cf. Eq. (6.5)), getting

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

and

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

The final block of the inverse is obtained as

$$\begin{aligned} \mathbf{g}_{jj} &= \mathbf{a}_{jj}^{-1} + \begin{pmatrix} \mathbf{u}_{ji} \ \mathbf{u}_{jk} \end{pmatrix} \begin{pmatrix} \mathbf{g}_{ii} & \mathbf{g}_{ik} \\ \mathbf{g}_{ki} & \mathbf{g}_{kk} \end{pmatrix} \begin{pmatrix} \boldsymbol{\ell}_{ij} \\ \boldsymbol{\ell}_{kj} \end{pmatrix} \\ &= \mathbf{a}_{jj}^{-1} - \begin{pmatrix} \mathbf{g}_{ji} \ \mathbf{g}_{jk} \end{pmatrix} \begin{pmatrix} \boldsymbol{\ell}_{ij} \\ \boldsymbol{\ell}_{kj} \end{pmatrix} = \mathbf{a}_{jj}^{-1} - \mathbf{g}_{ji}\boldsymbol{\ell}_{ij} - \mathbf{g}_{jk}\boldsymbol{\ell}_{kj}. \end{aligned} \tag{6.27}$$

The Schur production step for a center block is visualized in Fig. 6.6, where the arrows are given the same significance as for a corner production step shown in Fig. 6.5.

Again, three different schemes arise because $\mathbf{g}_{jj}$ can be determined via one of the

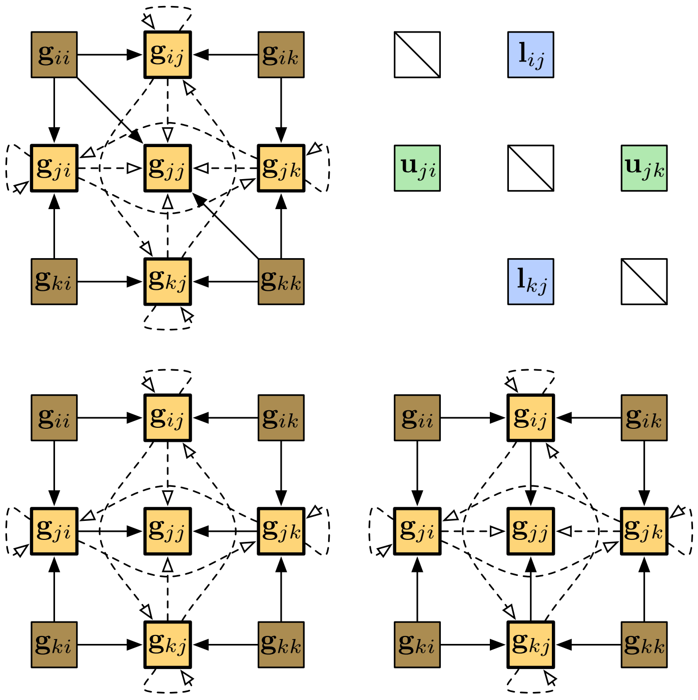


Figure 6.6: The three different schemes that represent a center production step undertaken in the BCR production phase, where we produce inverse block elements on row/column j using inverse blocks and LU factors from rows i and k.

following three equations, depending on how $\mathbf{g}_{jj}$ from Eq. (6.27) is calculated:

$$\mathbf{g}_{jj} = \mathbf{a}_{jj}^{-1} + \mathbf{u}_{ji}\mathbf{g}_{ii}\boldsymbol{\ell}_{ij} + \mathbf{u}_{jk}\mathbf{g}_{ki}\boldsymbol{\ell}_{ij} + \mathbf{u}_{ji}\mathbf{g}_{ik}\boldsymbol{\ell}_{kj} + \mathbf{u}_{jk}\mathbf{g}_{kk}\boldsymbol{\ell}_{kj} \tag{6.28}$$

$$\mathbf{g}_{jj} = \mathbf{a}_{jj}^{-1} - \mathbf{u}_{ji}\mathbf{g}_{ij} - \mathbf{u}_{jk}\mathbf{g}_{kj} \tag{6.29}$$

$$\mathbf{g}_{jj} = \mathbf{a}_{jj}^{-1} - \mathbf{g}_{ji}\boldsymbol{\ell}_{ij} - \mathbf{g}_{jk}\boldsymbol{\ell}_{kj}, \tag{6.30}$$

where (6.28), (6.29) and (6.30) correspond to the upper left, lower right and lower left schemes in Fig. 6.6. Similarly motivated as for the corner Schur production step, we choose the scheme related to (6.30) corresponding to the lower left corner of Fig. 6.6.

6.4.2 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 (6.5) 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 toward 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

$$\begin{pmatrix} \star & \star & \star & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{a}_{ih} & \mathbf{a}_{ii} & \mathbf{a}_{ij} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{a}_{jh}^{\mathrm{BCR}} & \mathbf{0} & \mathbf{a}_{jj}^{\mathrm{BCR}} & \mathbf{0} & \mathbf{a}_{jl}^{\mathrm{BCR}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{a}_{kj} & \mathbf{a}_{kk} & \mathbf{a}_{kl} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \star & \star & \star \end{pmatrix}, \tag{6.31}$$

where the new and updated elements are given as

$$\mathbf{a}_{jj}^{\mathrm{BCR}} \overset{\mathrm{def}}{=} \mathbf{a}_{jj} - \boldsymbol{\ell}_{ji}\mathbf{a}_{ij} - \boldsymbol{\ell}_{jk}\mathbf{a}_{kj}, \tag{6.32}$$

$$\mathbf{a}_{jh}^{\mathrm{BCR}} \overset{\mathrm{def}}{=} -\boldsymbol{\ell}_{ji}\mathbf{a}_{ih}, \tag{6.33}$$

$$\mathbf{a}_{jl}^{\mathrm{BCR}} \overset{\mathrm{def}}{=} -\boldsymbol{\ell}_{jk}\mathbf{a}_{kl}. \tag{6.34}$$

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 (6.5) produces all the other blocks of the inverse. The steps are similar to those in Section 6.4.1. 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.

6.5 Numerical results

6.5.1 Stability

Our elimination algorithm is equivalent to an unpivoted Gaussian elimination on a suitably permuted matrix $\mathbf{PAP}$ for some permutation matrix $\mathbf{P}$ [Gander and Golub, 1997; Heller, 1976].

Fig. 6.7 and Fig. 6.8 show the permutation corresponding to the Schur phase and the cyclic reduction phase for a 31×31 block matrix $\mathbf{A}$.

In the case of our algorithm, both approaches are combined. The process is shown

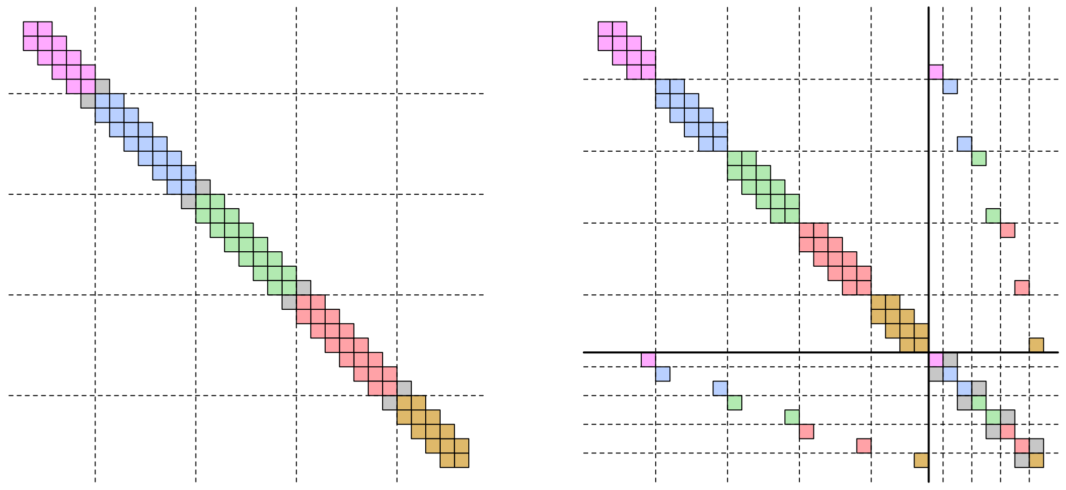


Figure 6.7: Permutation for a 31×31 block matrix **A** (left), which shows that our Schur reduction phase is identical to an unpivoted Gaussian elimination on a suitably permuted matrix **PAP** (right). Colors are associated with processes. The diagonal block structure of the top left part of the matrix (right panel) shows that this calculation is embarrassingly parallel.

on Fig. 6.9. 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 can 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.

6.5.2 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 of relatively identical size. Furthermore, this assumption serves as an introductory investigation on how

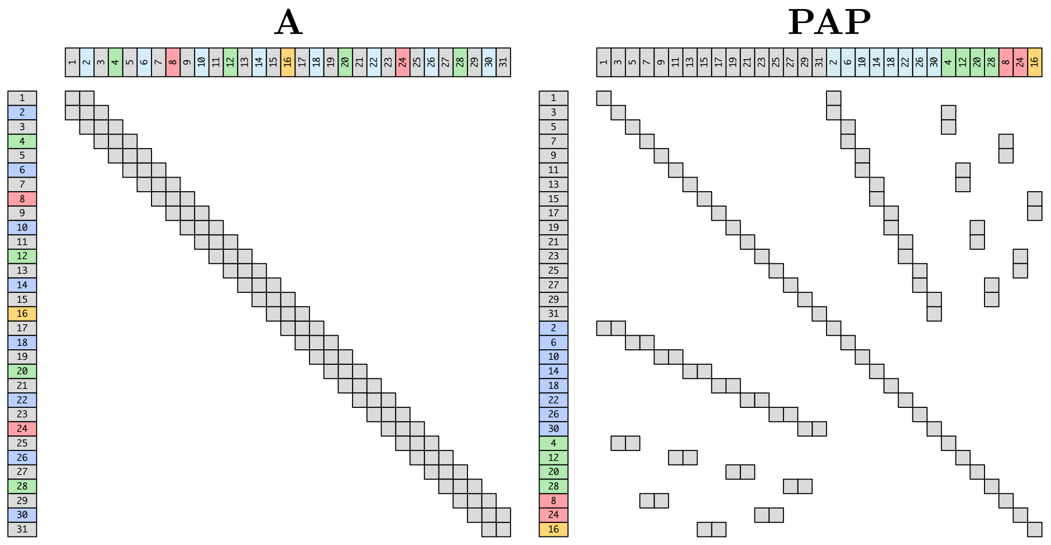


Figure 6.8: BCR corresponds to an unpivoted Gaussian elimination of **A** with permutation of rows and columns.

load balancing should be performed, eventually preparing us for a future investigation of the general case.

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 6.4.1, while the center processes perform those outlined in Section 6.4.1.

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. [Golub and Van Loan, 1989]), 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 [Petersen, 2008]. 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.

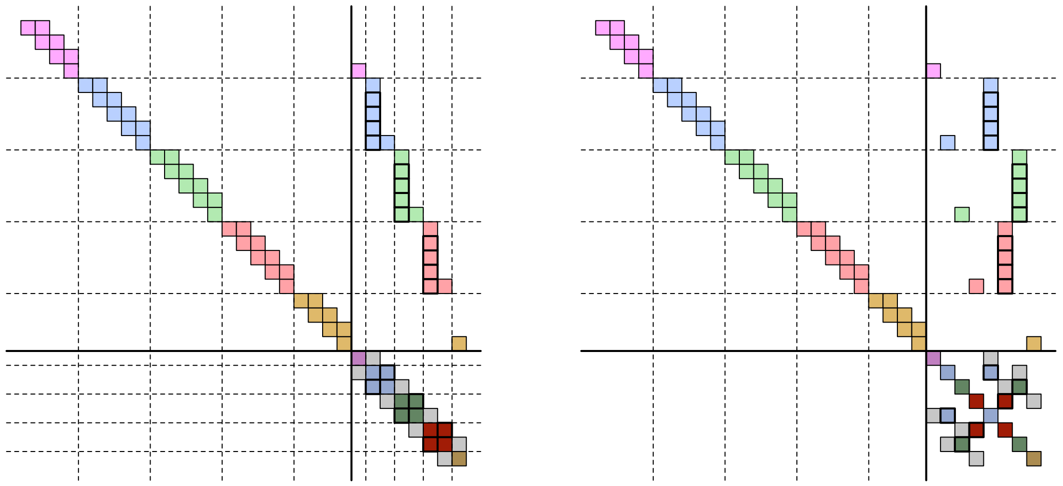


Figure 6.9: Following a Schur reduction phase on the permuted block matrix **PAP**, we obtain the reduced block tridiagonal system in the lower right corner (left panel). This reduced system is further permuted to a form shown on the right panel, as was done for BCR in Fig. 6.8.

6.5.3 Benchmarks

The benchmarking of the algorithms presented in this chapter 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.

The execution time was measured for a pure block cyclic reduction algorithm (Alg. C.1) in comparison with our hybrid algorithm (Alg. C.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. 6.10 for four different load balancing values $\alpha = \{1, 2, 2.636, 3\}$. A total number of processes used for execution was $\mathcal{P} = \{1, 2, 4, 8, 16, 32, 64\}$ in all cases.

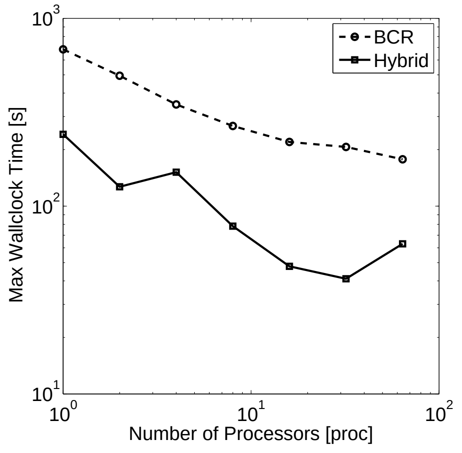
The speedup results corresponding to these walltime measurements are given in Fig. 6.11 for the four different load balancing values of α. For a uniform distribution where $\alpha = 1$, a serious performance hit is experienced for $\mathcal{P} = 4$. This is due to a poor load balance, as the two corner 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 $\alpha = 2$, 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.

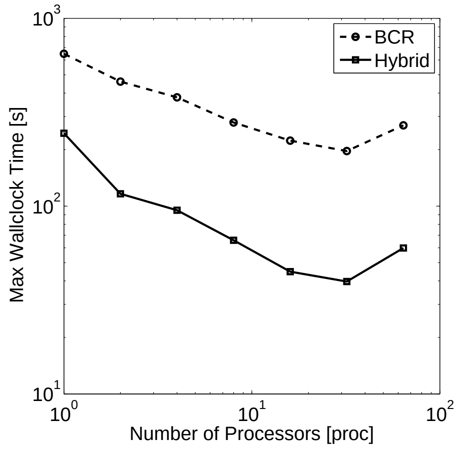
In an effort to determine which load balancing parameter α is optimal, we compare the walltime execution measurements for our hybrid algorithm in Fig. 6.12. From this figure, we can conclude that a uniform distribution with $\alpha = 1$ leads to poorer execution times; other values of α produce improved but similar results, particularly in the range $\mathcal{P} = 4$ to 8, which is a common core count in modern desktop computers.

6.6 Conclusion

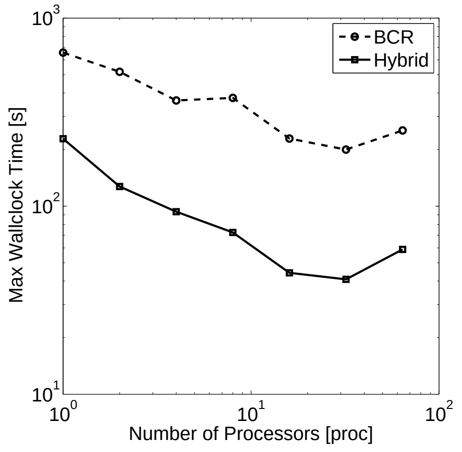
We have proposed new recurrence formulas [Eq. (6.6) and Eq. (6.13)] to calculate certain entries of the inverse of a sparse matrix. Such calculation scales like N^3 for a matrix of size $N \times N$ using a naïve algorithm, while our recurrences have reduced the cost by orders of magnitude. The computation cost using these recurrences is of the same order of magnitude as FIND-1way but with a smaller constant factor. This is an extension of the work by Takahashi [Takahashi et al., 1973] and others.



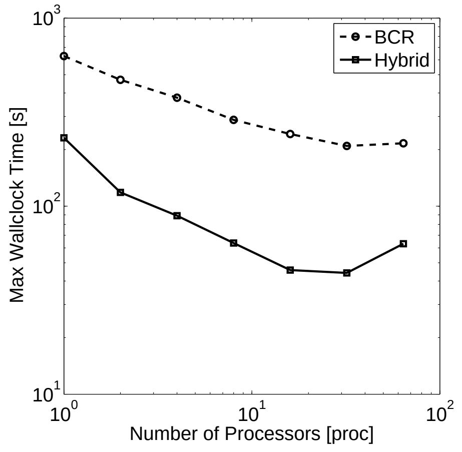
(a) $\alpha = 1$



(b) $\alpha = 2$

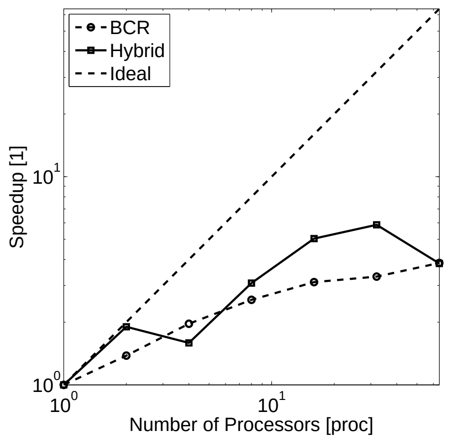


(c) $\alpha = 2.636$

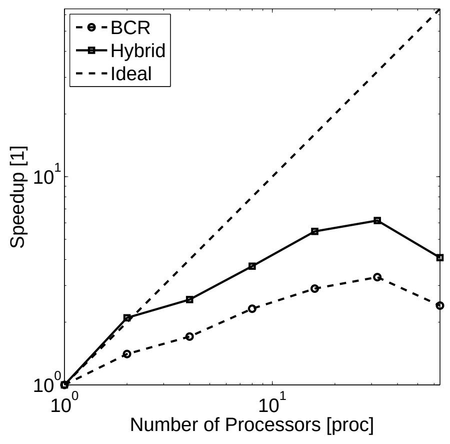


(d) $\alpha = 3$

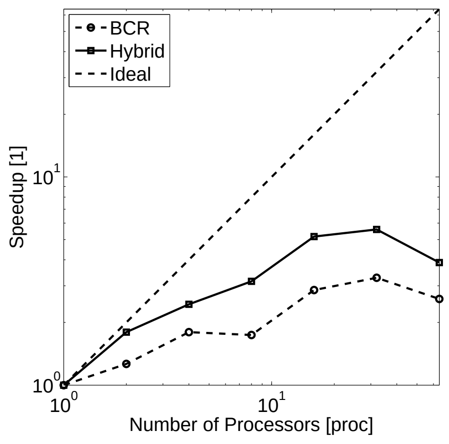
Figure 6.10: Walltime for our hybrid algorithm and pure BCR for different αs as a basic load balancing parameter. A block tridiagonal matrix $\mathbf{A}$ with $n = 512$ diagonal blocks, each of dimension $m = 256$, was used for these tests.



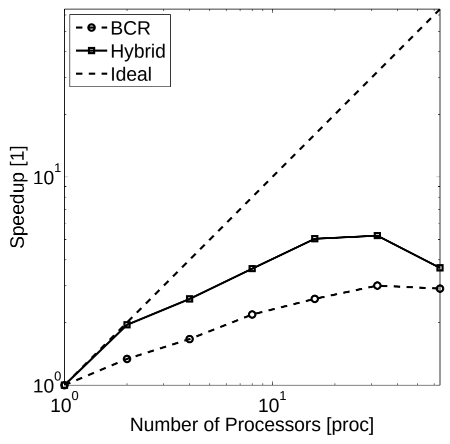
(a) $\alpha = 1$



(b) $\alpha = 2$



(c) $\alpha = 2.636$



(d) $\alpha = 3$

Figure 6.11: Speedup curves for our hybrid algorithm and pure BCR for different αs. A block tridiagonal matrix $\mathbf{A}$ with $n = 512$ diagonal blocks, each of dimension $m = 256$, was used.

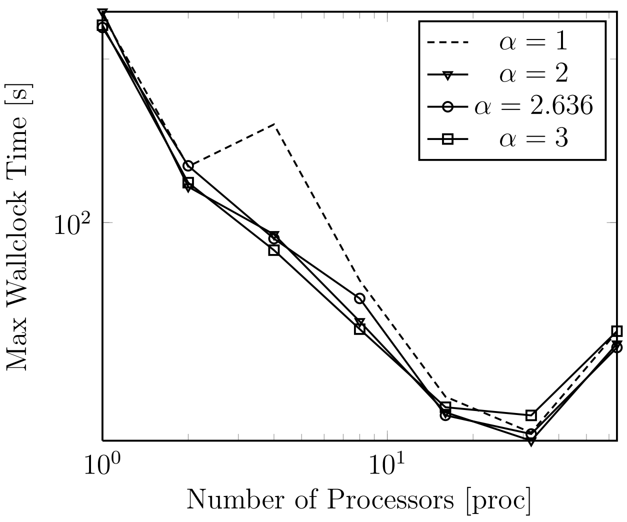


Figure 6.12: The total execution time for our hybrid algorithm is plotted against the number of processes $\mathcal{P}$. The choice $\alpha = 1$ gives poor results except when the number of processes is large. Other choices for α give similar results. The same matrix $\mathbf{A}$ as before was used.

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 discussed in Chapter 7.

Chapter 7

More Advanced Parallel Schemes

In Chapter 6, we discussed a hybrid scheme for parallelizing FIND-2way in 1D case. In the late stage of the reduction process in the scheme, since there are fewer rows left, more processors are idle. We can use these idle processors to do redundant computation and reduce the total running time. To achieve this, we need to return to FIND-1way schemes, where the two passes can be combined. In this chapter, we introduce a few different parallel schemes based on this idea. Some of the schemes work efficiently only for the 1D case while others work well for 2D and 3D cases as well.

7.1 Common features of the parallel schemes

All the schemes in this chapter are based solely on FIND-1way. For each target block, we eliminate its complement with respect to the whole domain by first eliminating the inner nodes of every block (except the target block itself) and then merging these blocks. Since we always eliminate the inner nodes of every block, we will not emphasize this step of elimination and assume that these inner nodes are already eliminated, denoted by the shaded areas (in the north-east and north-west directions).

In all the following sections for 1D parallel schemes, we use 16 processors and 16 blocks in 1D as the whole domain to illustrate how these schemes work. We assign one

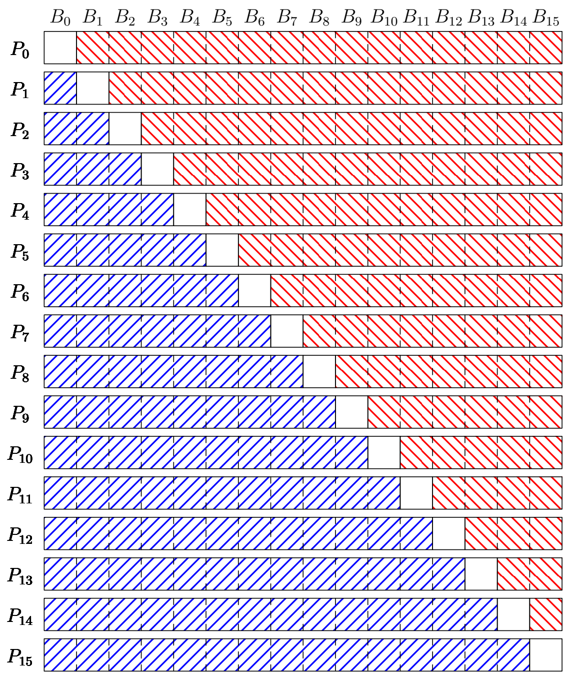

B_0 B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 B_{10} B_{11} B_{12} B_{13} B_{14} B_{15}
P_0
P_1
P_2
P_3
P_4
P_5
P_6
P_7
P_8
P_9
P_{10}
P_{11}
P_{12}
P_{13}
P_{14}
P_{15}

Figure 7.1: The common goal of the 1D parallel schemes in this chapter

target block for each of the processors: $P_0, P_1, \ldots, P_{15}$ with P_i assigned to target block B_i, and in the end, all the complement nodes are eliminated, as shown in Fig. 7.1. We will discuss several schemes to achieve this common goal with different merging processes.

7.2 PCR scheme for 1D problems

In our Parallel Cyclic Reduction (PCR) scheme, each processor starts with its own target block as the initial cluster, denoted as $\mathsf{C}_i^{(0)}$. This is shown in Fig. 7.2(a). On each processor, the initial cluster $\mathsf{C}_i^{(0)}$ is merged with its immediate neighboring cluster $\mathsf{C}_{(i+1) \bmod 16}$. The data for merging $\mathsf{C}_i^{(0)}$ and $\mathsf{C}_i^{(0)}$ on P_i comes from P_i and $P_{(i+1) \bmod 16}$. This is shown in Fig. 7.2(b). Since we always do a mod 16 operation after addition, we will skip it in subscripts for notation simplicity.

The newly formed cluster, denoted $\mathsf{C}_i^{(1)}$, is then merged with $\mathsf{C}_{i+2}^{(1)}$. This is shown in Fig. 7.2(c). Note that $\mathsf{C}_{15}^{(1)}$ has two parts. Generally, when a cluster is cut by the right end of the whole domain, the remaining parts start from the left, as shown by $\mathsf{C}_{15}^{(1)}$ in Fig. 7.2(b), $\mathsf{C}_{13}^{(2)}$, $\mathsf{C}_{14}^{(2)}$, and $\mathsf{C}_{15}^{(2)}$ in Fig. 7.2(c), and similar clusters in Figs. 7.2(d) and 7.2(e). We keep merging in this way until the cluster on each processor covers all the blocks in two groups (Fig. 7.2(e)). In the end, each processor communicates with its neighbor to get the complement blocks and to compute the inverse of the matrix corresponding to the target block (Fig. 7.2(f)).

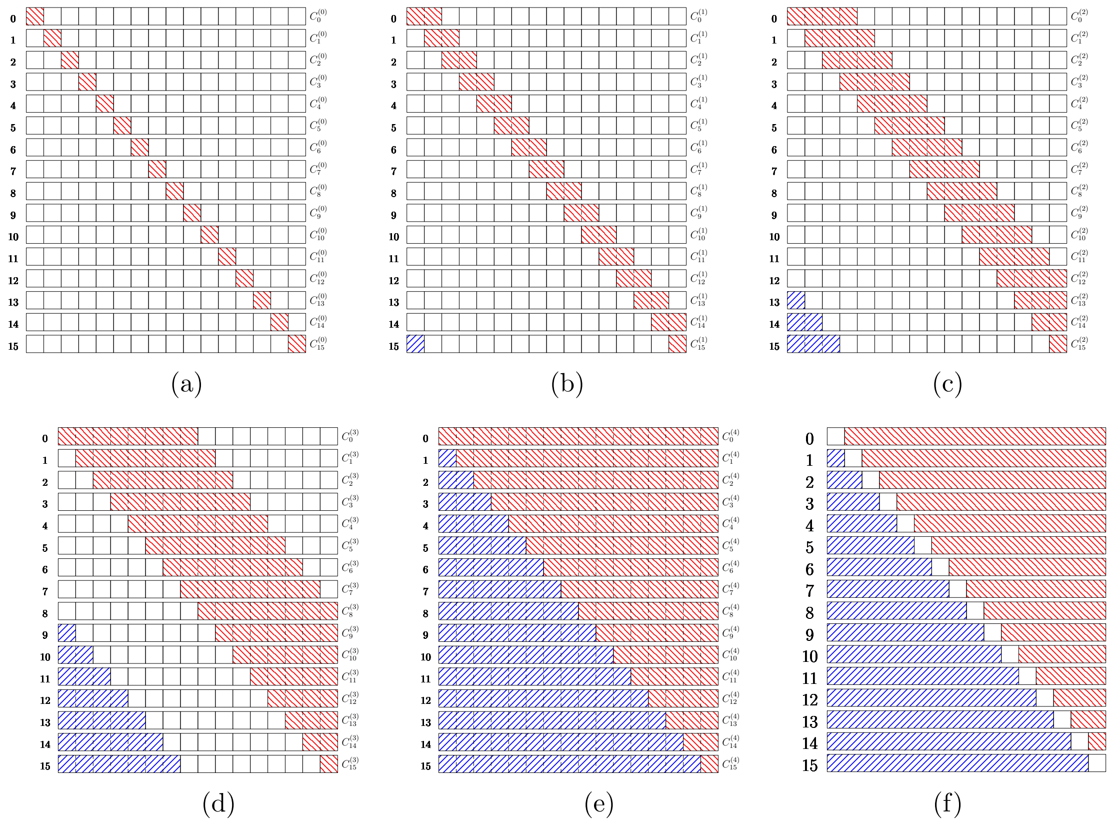


Figure 7.2: The merging process of PCR

Fig. 7.3 shows the communication pattern of this scheme. These patterns correspond to the steps in Fig. 7.2. For example, Fig. 7.3(a) is the communication pattern needed for the merging from Fig. 7.2(a) to Fig. 7.2(b) and Fig. 7.3(e) is the

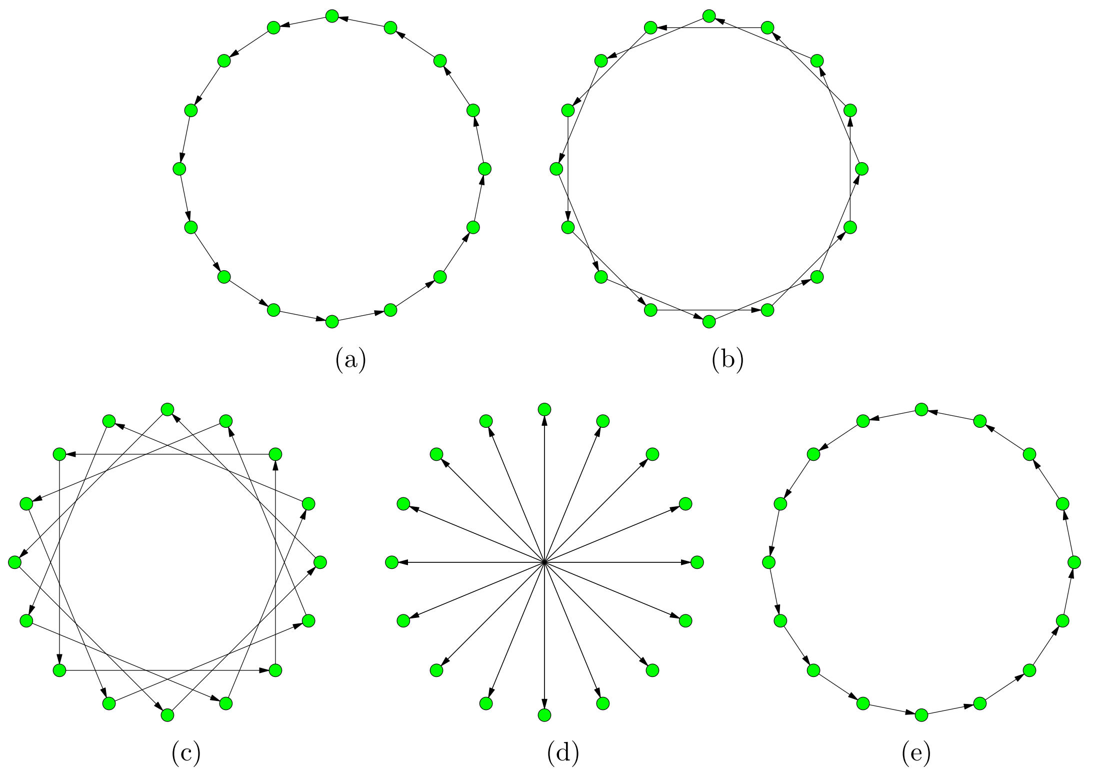


Figure 7.3: The communication patterns of PCR

communication pattern needed for the merging from Fig. 7.2(e) to Fig. 7.2(f).

7.3 PFIND-Complement scheme for 1D problems

In our Parallel-FIND Complement (PFIND-Complement) scheme, *subdomain* is the key concept. A subdomain is a subset of the whole domain consisting of adjacent blocks, which is used when considering the complement of the target block. The subdomain expands when the computation process goes on (usually doubled at each step) until it becomes the whole domain.

Each processor has its own subdomain at each step. We start from subdomains of small size and keep expanding them until they are as large as the whole domain. There are two types of computation involved in the subdomain expansion: one is for computing the complement and the other one for computing the subdomain. These two types of computations have counterparts in FIND-1way: the first one is similar to the downward pass there and the second one is identical to the upward pass. We will treat them as two separate processes in our discussion but they are carried out together in reality.

In the first process, each processor computes the complement of its target block with respect to its subdomain. As the subdomain on each processor expands, the complement expands as well through merging with its neighboring subdomain. Figs. 7.4(a)–7.4(d) show the process of expansion with subdomains of size 2, 4, 8, and 16. In the end, we obtain the complement of every target block and are ready to compute the corresponding inverse entries.

The second process computes the subdomain clusters needed by the above process. At each step, the subdomains are computed through merging smaller subdomain clusters in the previous step of this process. Fig. 7.5 illustrates this merging process. These clusters are needed by the process in Fig. 7.4. For example, the clusters in Fig. 7.5(b) are obtained by merging the clusters in Fig. 7.5(a) and used for the clusters in Fig. 7.4(c).

The above two processes are conceptually separate. In reality, however, the two

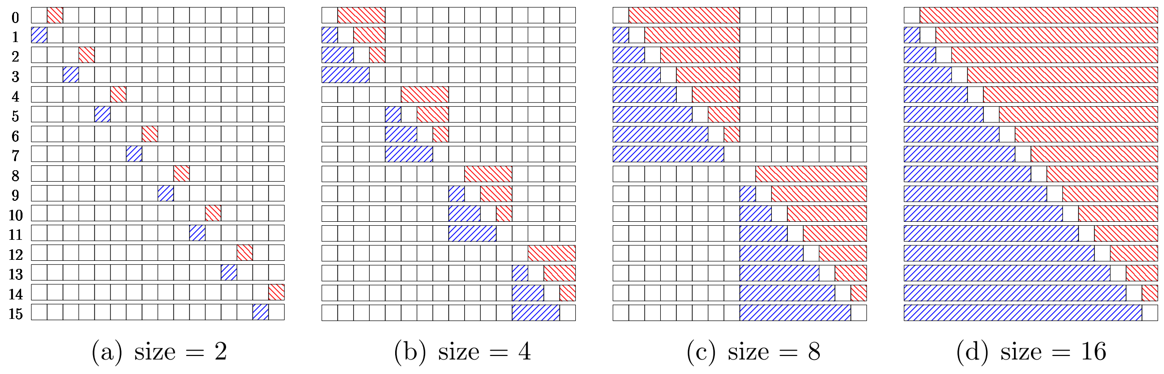


Figure 7.4: The complement clusters in the merging process of PFIND-complement. The subdomain size at each step is indicated in each sub-figure.

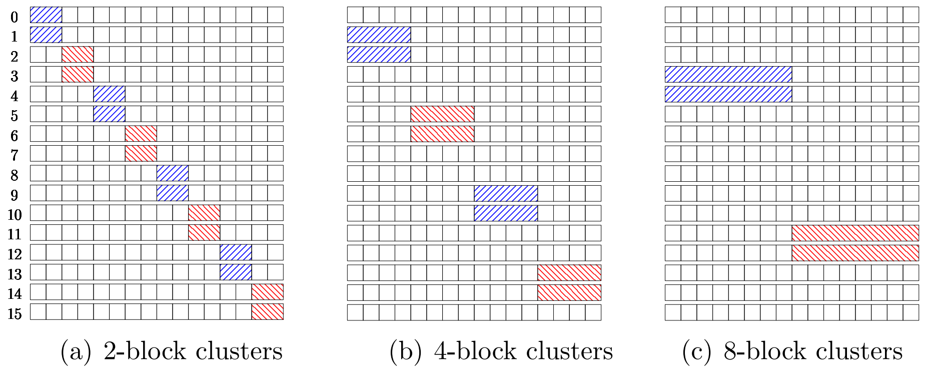


Figure 7.5: The subdomain clusters in the merging process of PFIND-complement.

processes, can be combined because in the process from Fig. 7.4(a)–Fig. 7.4(d), some processors are idle. For example, processors 1, 2, 5, 6, 9, 10, 13, and 14 are all idle when computing the clusters in Fig. 7.4(b); processors 3, 4, 11, and 12 are all idle when computing the clusters in Fig. 7.4(c), . As a result, we only need 4 steps in the above example:

1. individual blocks $\Rightarrow$ Fig. 7.5(a)
2. Fig. 7.4(a) and Fig. 7.5(a) $\Rightarrow$ Fig. 7.4(b) and Fig. 7.5(b)
3. Fig. 7.4(b) and Fig. 7.5(b) $\Rightarrow$ Fig. 7.4(c) and Fig. 7.5(c)
4. Fig. 7.4(c) and Fig. 7.5(c) $\Rightarrow$ Fig. 7.4(d).

Fig. 7.6 shows the communication pattern of PFIND-Complement. At each step, one processor in each subdomain will send its results (the merged clusters in Figs. 7.5(a)–7.5(c)) to the neighboring subdomain.

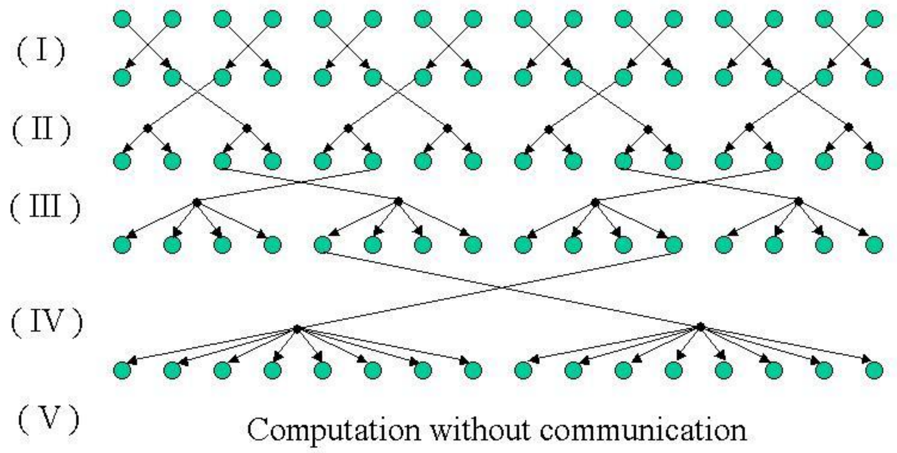


Figure 7.6: The communication pattern of PFIND-complement

Simple analysis shows that for a whole domain with n blocks, we need at most $\lceil \log_2 n \rceil$ steps. At each step, the computation cost for each processor is $\frac{7}{3}d^3$ floating-point multiplications (and the same order of floating-point additions), where d is the number of nodes we eliminate in each step. Since each processor only needs to keep the results in the current step, the storage cost is d^3 unit, where one unit is the storage cost for each floating-point number (e.g., 4 bytes for one single-precision floating-point real number and 16 bytes for one double-precision floating-point complex number). This cost is minimal because the cost remains the same throughout the computation process and the space for the current results at each step is required in any scheme.

At each step, every processor receives the results of size d^3 units from another processor in a different subdomain, but only one processor in each subdomain will send out its results (also of size d^3 units). Since the communication is between a single processor and multiple processors, an underlying multicast network topology can make it efficient, especially in late steps because fewer processors send out messages there.

7.4 Parallel schemes in 2D

The PFIND-Complement scheme can be easily extended to 2D problems. Instead of computing the complement of the target block with respect to the whole domain at the end of the whole computation process, we compute the complement of the target block with respect to subdomains at every step. Fig. 7.7 shows how the complement of target block 5 is computed in the original serial FIND-1way scheme (left) and in PFIND-Complement.

We can see that the right tree is shorter than the left one. Generally, the augmented tree for FIND-1way is twice as tall as that in PFIND-Complement. This means that PFIND-Complement saves the computation time approximately by half.

If we look at the trees in Fig. 7.7 carefully and compare these trees with the augmented trees for other target blocks (not shown), we can see that there is less overlap among different trees in PFIND-Complement. Such a lack of overlap leads to more overall computation cost at the top of the tree, but since we have more idle processors there, it does not lead to extra computation time. More detailed analysis will be discussed in our future work.

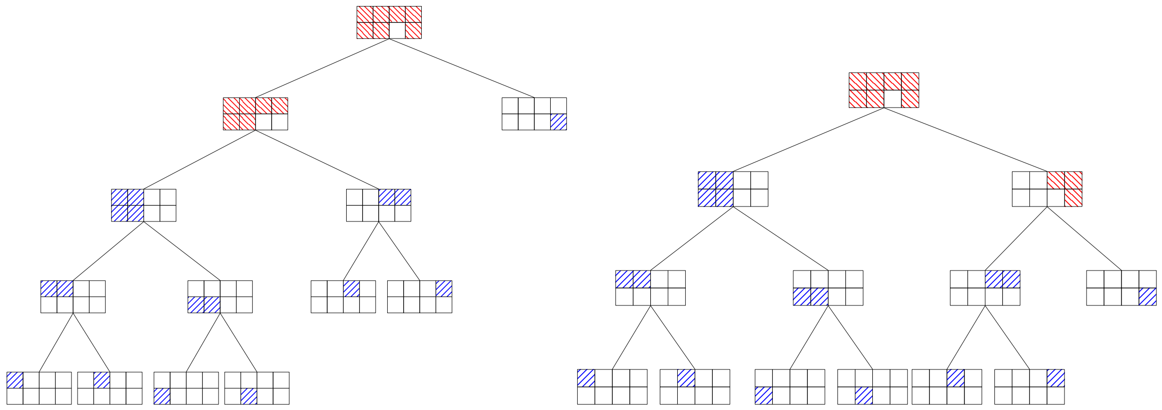


Figure 7.7: Augmented trees for target block 14 in FIND-1way and PFIND-Complement schemes

Chapter 8

Conclusions and Directions for Future Work

In this work, we have created a number of schemes for sparse matrix inverse related problems. These schemes are all based on the nested dissection method, which was originally designed for solving sparse linear systems. We made the method applicable to matrix inverse related problems, including computing $\mathbf{G}^r = \mathbf{A}^{-1}$, $\mathbf{G}^< = \mathbf{A}^{-1}\mathbf{\Sigma}\mathbf{A}^{-\dagger}$, and current density. The application leads to two schemes. One of them is based on Takahashi's observation and the other is based on our own observations. Both of them are faster than the state-of-the-art RGF method for similar problems by an order of magnitude. They are especially suitable for 2D problems of large size and our optimization makes them faster than RGF even for problems of quite small size. These schemes are all parallelization friendly. We proposed a few schemes for parallelization and implemented one of them.

Chapters 3, 4, and 5 discussed our basic FIND algorithm (FIND-1way) based on our three observations. This work was a collaboration with the Network for Computational Nanotechnology at Purdue for real device simulation. Chapter 6 discussed another FIND algorithm (FIND-2way). It is based on Takahashi's observation but we extended it to computing $\mathbf{G}^<$ and made it more parallelization friendly. Chapter 6 also discussed our implementation, analysis, and some practical issues in parallelization. This work was a collaboration with the Nano-Science Center at University of

Copenhagen. Chapter 7 proposed a few more advanced parallel schemes for future work.

All our implementations have proved to be excellent in running time, storage cost, and also stability for real problems. Theoretically, however, similar to other direct methods with various numbering schemes, we cannot at the same time employ typical pivoting schemes for preserving stability in the sparse LU factorization. We may take advantage of the limited freedom of numbering within the framework of nested dissection and employ partial pivoting when it is permitted, but whether it can be effective will depend on concrete problems. We leave it as our future work.

There are also other interesting serial and parallel schemes for our problems that we have not discussed yet. FIND-1way-SS (FIND-1way Single-Separator, cf. Page 92) is a serial algorithm similar to FIND-1way but more efficient and suitable for advanced parallel schemes. In FIND-1way we use the boundary nodes of two neighboring clusters as the *double-separator* between them, while in FIND-1way-SS we use a shared separator between neighboring clusters as in the nested dissection method. As a result, FIND-1way-SS has fewer columns to eliminate in each step and will significantly reduce the computation cost. This is achieved through the separation of the matrix multiplication and addition in (3.2). We will discuss more on this in our future work.

We also proposed another advanced parallel scheme (PFIND-Overlap) but have not discussed it. All these schemes can be generalized to sparse matrices arising from 3D problems and 2D problems with more complicated connectivity as well. Some of the parallel schemes can be efficiently applied to repeated linear systems. We will discuss them in our future work.

Appendices

Appendix A

Proofs for the Recursive Method Computing $\mathbf{G}^r$ and $\mathbf{G}^<$

Proposition A.1 *The following equations can be used to calculate* $Z_q := \mathbf{G}^q_{qq}$, $1 \leq q \leq n$*:*

$$Z_1 = \mathbf{A}_{11}^{-1}$$
$$Z_q = \left(\mathbf{A}_{qq} - \mathbf{A}_{q,q-1} Z_{q-1} \mathbf{A}_{q-1,q}\right)^{-1}$$

.

Proof: In general, for matrices W and X such that

$$\begin{pmatrix} W_1 & W_2 \\ W_3 & W_4 \end{pmatrix} \begin{pmatrix} X_1 & X_2 \\ X_3 & X_4 \end{pmatrix} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}$$

by Gaussian elimination, we have

$$\begin{pmatrix} W_1 & W_2 \\ \mathbf{0} & W_4 - W_3 W_1^{-1} W_2 \end{pmatrix} \begin{pmatrix} X_1 & X_2 \\ X_3 & X_4 \end{pmatrix} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \times & \mathbf{I} \end{pmatrix}$$

and then we have

$$X_4 = (W_4 - W_3 W_1^{-1} W_2)^{-1}.$$

For this problem, $X_4 = Z_q$, $W_4 = \mathbf{A}_{qq}$, $W_3 = \mathbf{A}_{q,1:q-1}$, and $W_1 = \mathbf{A}^{q-1}$, so we have

$$Z_q = (\mathbf{A}_{qq} - \mathbf{A}_{q,1:q-1}(\mathbf{A}^{q-1})^{-1}\mathbf{A}_{1:q-1,q})^{-1}.$$

By the block-tridiagonal structure of $\mathbf{A}$, we have $\mathbf{A}_{q,1:q-1} = (\mathbf{0} \;\; \mathbf{A}_{q,q-1})$ and $\mathbf{A}_{1:q-1,q} = \begin{pmatrix} \mathbf{0} \\ \mathbf{A}_{q-1,q} \end{pmatrix}$. Also, $\mathbf{G}^{q-1} = (\mathbf{A}^{q-1})^{-1}$ and $Z_{q-1} = \mathbf{G}^{q-1}_{q-1,q-1}$, so we have

$$Z_q = (\mathbf{A}_{qq} - \mathbf{A}_{q,q-1}Z_{q-1}\mathbf{A}_{q-1,q})^{-1}.$$

Starting from $Z_1 = \mathbf{A}_{11}^{-1}$, we can calculate all the Z_q, $1 \le q \le n$. □

Proposition A.2 *The following equations can be used to calculate* $\mathbf{G}_{qq}$, $1 \le q \le n$ *based on* $Z_q = \mathbf{G}^q_{qq}$*:*

$$\begin{aligned} \mathbf{G}_{nn} &= Z_n \\ \mathbf{G}_{qq} &= Z_q + Z_q\mathbf{A}_{q,q+1}\mathbf{G}_{q+1,q+1}\mathbf{A}_{q+1,q}Z_q. \end{aligned}$$

Proof: In general, for matrices W and X such that

$$\begin{pmatrix} W_1 & W_2 \\ W_3 & W_4 \end{pmatrix} \begin{pmatrix} X_1 & X_2 \\ X_3 & X_4 \end{pmatrix} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}$$

by Gaussian elimination and backward substitution, we have

$$X_1 = W_1^{-1} + W_1^{-1}W_2X_4W_3W_1^{-1}. \tag{A.1}$$

Let $W = \mathbf{A} = \begin{pmatrix} \mathbf{A}_{1:q,1:q} & \mathbf{A}_{1:q,q+1:n} \\ \mathbf{A}_{q+1:n,1:q} & \mathbf{A}_{q+1:n,q+1:n} \end{pmatrix}$ and $X = \mathbf{G} = \begin{pmatrix} \mathbf{G}_{1:q,1:q} & \mathbf{G}_{1:q,q+1:n} \\ \mathbf{G}_{q+1:n,1:q} & \mathbf{G}_{q+1:n,q+1:n} \end{pmatrix}$.

Since $W_1 = \mathbf{A}_{1:q,1:q} = \mathbf{A}^q$, we have $W_1^{-1} = \mathbf{G}^q = \begin{pmatrix} \times & \times \\ \times & Z_q \end{pmatrix}$.

Also, $W_2 = \mathbf{A}_{1:q,q+1:n} = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{q,q+1} & \mathbf{0} \end{pmatrix}$, $W_3 = \mathbf{A}_{q+1:n,1:q} = \begin{pmatrix} \mathbf{0} & \mathbf{A}_{q+1,q} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}$, and

$X_4 = \mathbf{G}_{q+1:n,q+1:n} = \begin{pmatrix} \mathbf{G}_{q+1,q+1} & \times \\ \times & \times \end{pmatrix}$, so we have $W_2X_4W_3 = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{q,q+1}\mathbf{G}_{q+1,q+1}\mathbf{A}_{q+1,q} \end{pmatrix}$ and $W_1^{-1}W_2X_4W_3W_1^{-1} = \begin{pmatrix} \times & \times \\ \times & Z_q\mathbf{A}_{q,q+1}\mathbf{G}_{q+1,q+1}\mathbf{A}_{q+1,q}Z_q \end{pmatrix}$. Consider the (q,q) block of (2.4), we have

$$\mathbf{G}_{qq} = Z_q + Z_q\mathbf{A}_{q,q+1}\mathbf{G}_{q+1,q+1}\mathbf{A}_{q+1,q}Z_q.$$

Starting from $\mathbf{G}_{nn} = \mathbf{G}^n_{nn}$ we can calculate all the diagonal blocks $\mathbf{G}_{qq}$, $1 \le q \le n$, using a backward recurrence. □

A.1 Forward recurrence

We denote

$$\mathbf{A}^q\mathbf{G}^{<q} \overset{def}{=} \mathbf{\Sigma}^q(\mathbf{G}^q)^*, \tag{A.2}$$

where as before, $\mathbf{\Sigma}^q = \mathbf{\Sigma}_{1:q,1:q}$. The following forward recurrence holds.

Proposition A.3 *The following equations can be used to calculate $Z_q^< = \mathbf{G}^{<q}_{qq}$, $1 \le q \le n$:*

$$Z_1^< = Z_1\mathbf{\Sigma}^1 Z_1^*$$
$$Z_q^< = Z_q\big(\mathbf{\Sigma}_{q,q} - \mathbf{\Sigma}_{q,q-1}Z_{q-1}^*\mathbf{A}_{q,q-1}^* - \mathbf{A}_{q,q-1}Z_{q-1}\mathbf{\Sigma}_{q-1,q} + \mathbf{A}_{q,q-1}Z_{q-1}^<\mathbf{A}_{q,q-1}^*\big)Z_q^*.$$

Proof: In general, consider W, X, and Y such that

$$\begin{pmatrix} W_1 & W_2 \\ W_3 & W_4 \end{pmatrix}\begin{pmatrix} X_1 & X_2 \\ X_3 & X_4 \end{pmatrix}\begin{pmatrix} W_1 & W_2 \\ W_3 & W_4 \end{pmatrix}^* = \begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix}.$$

By Gaussian elimination, we have

$$\begin{pmatrix} W_1 & W_2 \\ \mathbf{0} & W_4 - W_3W_1^{-1}W_2 \end{pmatrix}\begin{pmatrix} X_1 & X_2 \\ X_3 & X_4 \end{pmatrix}\begin{pmatrix} W_1^* & \mathbf{0} \\ W_2^* & W_4^* - W_2^*W_1^{-*}W_3^* \end{pmatrix}^*$$

$$= \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ -W_3W_1^{-1} & \mathbf{I} \end{pmatrix} \begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} \begin{pmatrix} \mathbf{I} & -W_1^{-*}W_3^* \\ \mathbf{0} & \mathbf{I} \end{pmatrix} = \begin{pmatrix} \tilde{Y_1} & \tilde{Y_2} \\ \tilde{Y_3} & \tilde{Y_4} \end{pmatrix},$$

where

$$\tilde{Y_4} = Y_4 - W_3W_1^{-1}Y_2 - Y_3W_1^{-*}W_3^* + W_3W_1^{-1}Y_1W_1^{-*}W_3^*,$$

so we have

$$X_4 = (W_4 - W_3W_1^{-1}W_2)^{-1}\tilde{Y_4}(W_4^* - W_2^*W_1^{-*}W_3^*)^{-1}.$$

Now, let $W = \mathbf{A}^q$, $X = \mathbf{G}^{<q}$, and $Y = \mathbf{\Sigma}^q$, and blockwise, let

$$W_1 = \mathbf{A}^{q-1}, W_2 = \mathbf{A}_{1:q-1,q}, W_3 = \mathbf{A}_{q,1:q-1}, W_4 = \mathbf{A}_{qq}$$

$$X_1 = \mathbf{G}^{<q}_{1:q-1,1:q-1}, X_2 = \mathbf{G}^{<q}_{1:q-1,q}, X_3 = \mathbf{G}^{<q}_{q,1:q-1}, X_4 = \mathbf{G}^{<q}_{q,q}$$

$$Y_1 = \mathbf{\Sigma}^{q-1}, Y_2 = \mathbf{\Sigma}_{1:q-1,q}, Y_3 = \mathbf{\Sigma}_{q,1:q-1}, Y_4 = \mathbf{\Sigma}_{q,q}.$$

Since $(W_4 - W_3W_1^{-1}W_2)^{-1} = \mathbf{G}^q_{qq} = Z_q$, $W_3 = (\mathbf{0} \quad \mathbf{A}_{q,q-1})$ and $W_1^{-1}\mathbf{\Sigma}^{q-1}W_1^{-*} = \mathbf{G}^{<q-1}$, we have

$$Z_q^< = Z_q\big(\mathbf{\Sigma}_{q,q} - \mathbf{\Sigma}_{q,q-1}Z_{q-1}^*\mathbf{A}_{q,q-1}^* - \mathbf{A}_{q,q-1}Z_{q-1}\mathbf{\Sigma}_{q-1,q} + \mathbf{A}_{q,q-1}Z_{q-1}^<\mathbf{A}_{q,q-1}^*\big)Z_q^*.$$

Starting from $Z_1^< = Z_1\mathbf{\Sigma}^1Z_1^*$, we have a recursive algorithm to calculate all the $Z_q^< = X_{qq}^q$, $1 \le q \le n$. □

As before, the last block $\mathbf{G}^{<n}_{nn}$ is equal to $\mathbf{G}^<_{nn}$. In the next section, we use a backward recurrence to calculate $\mathbf{G}^<_{qq}$, $1 \le q \le n$.

A.2 Backward recurrence

The recurrence is this time given by the following.

Proposition A.4 *The following equations can be used to calculate $G^<_{qq}$, $1 \le q \le n$:*

$$\begin{aligned}
\mathbf{G}^<_{nn} &= Z^<_n \\
\mathbf{G}^<_{qq} &= Z^<_q + Z_q \mathbf{A}_{q,q+1} \mathbf{G}_{q+1,q+1} \mathbf{A}_{q+1,q} Z^{<*}_g + Z^<_q \mathbf{A}^*_{q+1,q} \mathbf{G}^*_{q+1,q+1} \mathbf{A}^*_{q,q+1} Z^*_q \\
&+ Z_q[\mathbf{A}_{q,q+1} \mathbf{G}^<_{q+1,q+1} \mathbf{A}^*_{q,q+1} - \mathbf{A}_{q,q+1} \mathbf{G}_{q+1,q+1} \mathbf{\Sigma}_{q+1,q} - \mathbf{\Sigma}_{q,q+1} \mathbf{G}^*_{q+1,q+1} \mathbf{A}^*_{q,q+1}] Z^*_q.
\end{aligned}$$

Proof: Consider matrices W, X, and Y such that

$$\begin{pmatrix} W_1 & W_2 \\ W_3 & W_4 \end{pmatrix} \begin{pmatrix} X_1 & X_2 \\ X_3 & X_4 \end{pmatrix} \begin{pmatrix} W_1 & W_2 \\ W_3 & W_4 \end{pmatrix}^* = \begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix}^* .$$

Performing Gaussian elimination, we have

$$\begin{pmatrix} W_1 & W_2 \\ \mathbf{0} & U_4 \end{pmatrix} \begin{pmatrix} X_1 & X_2 \\ X_3 & X_4 \end{pmatrix} \begin{pmatrix} W_1^* & \mathbf{0} \\ W_2^* & U_4^* \end{pmatrix} = \begin{pmatrix} Y_1 & \tilde{Y}_2 \\ \tilde{Y}_3 & \tilde{Y}_4 \end{pmatrix}^*$$

$$\Rightarrow W_1 X_1 W_1^* = Y_1^* - W_2 X_3 W_1^* - W_1 X_2 W_2^* - W_2 X_4 W_2^*.$$

Similar to the back substitution in the previous section, we have

$$\begin{gathered}
U_4 X_3 W_1^* = \tilde{Y}_3 - U_4 X_4 W_2^* = Y_3 - W_3 W_1^{-1} Y_1 - U_4 X_4 W_2^* \\
\Rightarrow X_3 W_1^* = U_4^{-1} Y_3 - U_4^{-1} W_3 W_1^{-1} Y_1 - X_4 W_2^*
\end{gathered}$$

and

$$\begin{gathered}
W_1 X_2 U_4^* = \tilde{Y}_2 - W_2 X_4 U_4^* = Y_2 - Y_1 W_1^{-*} W_3^* - W_2 X_4 U_4^* \\
\Rightarrow W_1 X_2 = Y_2 U_4^{-*} - Y_1 W_1^{-*} W_3^* U_4^{-*} - W_2 X_4.
\end{gathered}$$

Now we have

$$\begin{aligned}
X_1 &= (W_1^{-1} Y_1 W_1^{-*}) + W_1^{-1}(W_2 U_4^{-1} W_3)(W_1^{-1} Y_1 W_1^{-*}) + (W_1^{-1} Y_1 W_1^{-*})(W_3^{-*} U_4^{-*} W_2^*) W_1^{-*} \\
&\quad + W_1^{-1}(W_2 X_4 W_2^* - W_2 U_4^{-1} Y_3 - Y_2 U_4^{-*} W_2^*) W_1^{-*}.
\end{aligned}$$

By definition, we have

$$X_1 = \begin{pmatrix} \times & \times \\ \times & \mathbf{G}^<_{qq} \end{pmatrix}, W_1^{-1} Y_1 W_1^{-*} = \begin{pmatrix} \times & \times \\ \times & Z^<_q \end{pmatrix}, W_1^{-1} = \begin{pmatrix} \times & \times \\ \times & Z_q \end{pmatrix}.$$

By the result of previous section, we have

$$W_2 U_4^{-1} W_3 = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{q,q+1}\mathbf{G}_{q+1,q+1}\mathbf{A}_{q+1,q} \end{pmatrix}, W_3^{-*} U_4^{-*} W_2^* = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}^*_{q+1,q}\mathbf{G}^*_{q+1,q+1}\mathbf{A}^*_{q,q+1} \end{pmatrix}$$

and

$$X_4 = \begin{pmatrix} \mathbf{G}^<_{q+1,q+1} & \times \\ \times & \times \end{pmatrix}, U_4^{-1} = \begin{pmatrix} \mathbf{G}_{q+1,q+1} & \times \\ \times & \times \end{pmatrix}, W_2 = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{q,q+1} & \mathbf{0} \end{pmatrix}$$

$$Y_3 = \begin{pmatrix} \mathbf{0} & \mathbf{\Sigma}_{q+1,q} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}, Y_2 = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{\Sigma}_{q,q+1} & \mathbf{0} \end{pmatrix}$$

and then

$$W_2 X_4 W_2^* = \begin{pmatrix} \times & \times \\ \times & \mathbf{A}_{q,q+1}\mathbf{G}^<_{q+1,q+1}\mathbf{A}^*_{q,q+1} \end{pmatrix}, W_2 U_4^{-1} Y_3 = \begin{pmatrix} \times & \times \\ \times & \mathbf{A}_{q,q+1}\mathbf{G}_{q+1,q+1}\mathbf{\Sigma}_{q+1,q} \end{pmatrix},$$

and

$$Y_2 U_4^{-*} W_2^* = \begin{pmatrix} \times & \times \\ \times & \mathbf{\Sigma}_{q,q+1}\mathbf{G}^*_{q+1,q+1}\mathbf{A}^*_{q,q+1} \end{pmatrix}.$$

Put them all together and consider the (q, q) block of equation (*):

$$\begin{aligned} \mathbf{G}^<_{qq} &= Z^<_q + Z_q\mathbf{A}_{q,q+1}\mathbf{G}_{q+1,q+1}\mathbf{A}_{q+1,q}Z^{<*}_g + Z^<_q\mathbf{A}^*_{q+1,q}\mathbf{G}^*_{q+1,q+1}\mathbf{A}^*_{q,q+1}Z^*_q \\ &\quad + Z_q[\mathbf{A}_{q,q+1}\mathbf{G}^<_{q+1,q+1}\mathbf{A}^*_{q,q+1} - \mathbf{A}_{q,q+1}\mathbf{G}_{q+1,q+1}\mathbf{\Sigma}_{q+1,q} - \mathbf{\Sigma}_{q,q+1}\mathbf{G}^*_{q+1,q+1}\mathbf{A}^*_{q,q+1}]Z^*_q. \end{aligned}$$

$\square$

Appendix B

Theoretical Supplement for FIND-1way Algorithms

In this appendix, we present the properties, theorems, corollaries, and their proofs that are used for computing $\mathbf{G}^r$ and $\mathbf{G}^<$. We first present those used for computing both $\mathbf{G}^r$ and $\mathbf{G}^<$, and then those used for computing only $\mathbf{G}^<$.

B.1 Proofs for both computing $\mathbf{G}^r$ and computing $\mathbf{G}^<$

In this section, we first list all the properties of the mesh node subsets with proofs if necessary, then the precise statement of Theorem B.1 with its proof, and lastly the three corollaries with proofs, which were used in Section 3. The first property is fairly simple but will be used again and again in proving theorems, corollaries, and other properties.

Property B.1 *By definition of* T_r^+*, one of the following relations must hold:* $\mathsf{C}_i \subset \mathsf{C}_j$*,* $\mathsf{C}_i \supset \mathsf{C}_j$*, or* $\mathsf{C}_i \cap \mathsf{C}_j = \emptyset$*.*

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

Property B.2 $\mathsf{I}_i = \cup_{\mathsf{C}_j \subseteq \mathsf{C}_i} \mathsf{S}_j$, *where* C_i *and* $\mathsf{C}_j \in \mathsf{T}_r^+$.

Proof: By the definition of S_i, we have $\mathsf{I}_i \subseteq (\mathsf{I}_i \backslash \cup_{\mathsf{C}_j \subset \mathsf{C}_i} \mathsf{S}_j) \cup (\cup_{\mathsf{C}_j \subset \mathsf{C}_i} \mathsf{S}_j) = \mathsf{S}_i \cup (\cup_{\mathsf{C}_j \subset \mathsf{C}_i} \mathsf{S}_j) = \cup_{\mathsf{C}_j \subseteq \mathsf{C}_i} \mathsf{S}_j$, so it remains to show $\cup_{\mathsf{C}_j \subseteq \mathsf{C}_i} \mathsf{S}_j \subseteq \mathsf{I}_i$. Since $\mathsf{S}_j \subseteq \mathsf{I}_j$ and then $\cup_{\mathsf{C}_j \subseteq \mathsf{C}_i} \mathsf{S}_j \subseteq \cup_{\mathsf{C}_j \subseteq \mathsf{C}_i} \mathsf{I}_j$, if suffices to show $\cup_{\mathsf{C}_j \subseteq \mathsf{C}_i} \mathsf{I}_j \subseteq \mathsf{I}_i$.

To show this, we show that $\mathsf{C}_j \subseteq \mathsf{C}_i$ implies $\mathsf{I}_j \subseteq \mathsf{I}_i$. For any C_i, $\mathsf{C}_j \in \mathsf{T}_r^+$ and $\mathsf{C}_j \subseteq \mathsf{C}_i$, if $\mathsf{I}_j \not\subseteq \mathsf{I}_i$, then $\mathsf{I}_j \backslash \mathsf{I}_i \neq \emptyset$ and $\mathsf{I}_j \subset \mathsf{C}_j \subseteq \mathsf{C}_i = \mathsf{I}_i \cup \mathsf{B}_i \Rightarrow \mathsf{I}_j \backslash \mathsf{I}_i \subseteq (\mathsf{I}_i \cup \mathsf{B}_i) \backslash \mathsf{I}_i = \mathsf{B}_i \Rightarrow \mathsf{I}_j \backslash \mathsf{I}_i = (\mathsf{I}_j \backslash \mathsf{I}_i) \cap \mathsf{B}_i \subset \mathsf{I}_j \cap \mathsf{B}_i \Rightarrow \mathsf{I}_j \cap \mathsf{B}_i \neq \emptyset$, which contradicts the definition of I_i and B_i. So we have $\mathsf{I}_j \subseteq \mathsf{I}_i$ 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 3.2.1 and illustrated in Fig. 3.7.

Property B.3 *If* C_i *and* C_j *are the two children of* C_k, *then* $\mathsf{S}_k \cup \mathsf{B}_k = \mathsf{B}_i \cup \mathsf{B}_j$ *and* $\mathsf{S}_k = (\mathsf{B}_i \cup \mathsf{B}_j) \backslash \mathsf{B}_k$.

Property B.4 *For any given augmented tree* T_r^+ *and all* $\mathsf{C}_i \in \mathsf{T}_r^+$, S_i, B_{-r}, *and* C_r *are all disjoint and* $\mathsf{M} = (\cup_{\mathsf{C}_i \in \mathsf{T}_r^+} \mathsf{S}_i) \cup \mathsf{B}_{-r} \cup \mathsf{C}_r$.

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

Since $\forall \mathsf{C}_i \in \mathsf{T}_r^+$ we have $\mathsf{C}_i \subseteq \mathsf{C}_{-r}$, and by Property B.3, we have $\mathsf{S}_i \subseteq \mathsf{I}_{-r}$. Since $\mathsf{I}_{-r} \cap \mathsf{B}_{-r} = \emptyset$, we have $\mathsf{S}_i \cap \mathsf{B}_{-r} = \emptyset$. Since $\mathsf{C}_{-r} \cap \mathsf{C}_r = \emptyset$, $\mathsf{S}_i \subset \mathsf{C}_{-r}$, and $\mathsf{B}_{-r} \subset \mathsf{C}_{-r}$, we have $\mathsf{S}_i \cap \mathsf{C}_r = \emptyset$ and $\mathsf{B}_{-r} \cap \mathsf{C}_r = \emptyset$. By Property B.3 again, we have $\cup_{\mathsf{C}_i \in \mathsf{T}_r^+} \mathsf{S}_i = \cup_{\mathsf{C}_i \subseteq \mathsf{C}_{-r}} \mathsf{S}_i = \mathsf{I}_{-r}$. So we have $(\cup_{\mathsf{C}_i \in \mathsf{T}_r^+} \mathsf{S}_i) \cup \mathsf{B}_{-r} \cup \mathsf{C}_r = (\mathsf{I}_{-r} \cup \mathsf{B}_{-r}) \cup \mathsf{C}_r = \mathsf{C}_{-r} \cup \mathsf{C}_r = \mathsf{M}$. □

Below we list properties of S_i for specific orderings.

Property B.5 *If* $\mathsf{S}_i < \mathsf{S}_j$, *then* $\mathsf{C}_j \not\subset \mathsf{C}_i$, *which implies either* $\mathsf{C}_i \subset \mathsf{C}_j$ *or* $\mathsf{C}_i \cap \mathsf{C}_j = \emptyset$.

This property is straightforward from the definition of $\mathsf{S}_i < \mathsf{S}_j$ and Property B.1.

The following two properties are related to the elimination process and will be used in the proof of Theorem B.1.

Property B.6 *For any k, u such that $\mathsf{C}_k, \mathsf{C}_u \in \mathsf{T}_r^+$, if $\mathsf{S}_k < \mathsf{S}_u$, then $\mathsf{S}_u \cap \mathsf{I}_k = \emptyset$.*

Proof: By Property B.5, we have $\mathsf{C}_u \not\subset \mathsf{C}_k$. So for all j such that $\mathsf{C}_j \subseteq \mathsf{C}_k$, we have $j \neq u$ and thus $\mathsf{S}_j \cap \mathsf{S}_u = \emptyset$ by Property B.3. By Property B.2, $\mathsf{I}_k = \cup_{\mathsf{C}_j \subseteq \mathsf{C}_k} \mathsf{S}_j$, so we have $\mathsf{I}_k \cap \mathsf{S}_u = \emptyset$. □

Property B.7 *If C_j is a child of C_k, then for any C_u such that $\mathsf{S}_j < \mathsf{S}_u < \mathsf{S}_k$, we have $\mathsf{C}_j \cap \mathsf{C}_u = \emptyset$ and thus $\mathsf{B}_j \cap \mathsf{B}_u = \emptyset$.*

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

Proof: By Property B.5, either $\mathsf{C}_j \subset \mathsf{C}_u$ or $\mathsf{C}_j \cap \mathsf{C}_u = \emptyset$. Since C_j is a child of C_k and $u \neq k$, we have $\mathsf{C}_j \subset \mathsf{C}_u \Rightarrow \mathsf{C}_k \subset \mathsf{C}_u \Rightarrow \mathsf{S}_k < \mathsf{S}_u$, which contradicts the given condition $\mathsf{S}_u < \mathsf{S}_k$. So $\mathsf{C}_j \not\subset \mathsf{C}_u$ and then $\mathsf{C}_j \cap \mathsf{C}_u = \emptyset$. □

Now we re-state Theorem B.1 more precisely with its proof (see page 35).

Theorem B.1 *If we perform Gaussian elimination as described in Section 3.2 on the original matrix $\mathbf{A}$ with ordering consistent with any given T_r^+, then*

1. $\mathbf{A}_g(\mathsf{S}_{\geq g}, \mathsf{S}_{<g}) = 0$;

2. $\forall h \geq g, \mathbf{A}_g(\mathsf{S}_h, \mathsf{S}_{>h} \backslash \mathsf{B}_h) = \mathbf{A}_g(\mathsf{S}_{>h} \backslash \mathsf{B}_h, \mathsf{S}_h) = 0$;

3. (a) $\mathbf{A}_{g+}(\mathsf{B}_g, \mathsf{S}_{>g} \backslash \mathsf{B}_g) = \mathbf{A}_g(\mathsf{B}_g, \mathsf{S}_{>g} \backslash \mathsf{B}_g)$;

   (b) $\mathbf{A}_{g+}(\mathsf{S}_{>g} \backslash \mathsf{B}_g, \mathsf{B}_g) = \mathbf{A}_g(\mathsf{S}_{>g} \backslash \mathsf{B}_g, \mathsf{B}_g)$;

   (c) $\mathbf{A}_{g+}(\mathsf{S}_{>g} \backslash \mathsf{B}_g, \mathsf{S}_{>g} \backslash \mathsf{B}_g) = \mathbf{A}_g(\mathsf{S}_{>g} \backslash \mathsf{B}_g, \mathsf{S}_{>g} \backslash \mathsf{B}_g)$;

4. $\mathbf{A}_{g+}(\mathsf{B}_g, \mathsf{B}_g) = \mathbf{A}_g(\mathsf{B}_g, \mathsf{B}_g) - \mathbf{A}_g(\mathsf{B}_g, \mathsf{S}_g)\mathbf{A}_g(\mathsf{S}_g, \mathsf{S}_g)^{-1}\mathbf{A}_g(\mathsf{S}_g, \mathsf{B}_g)$.

The matrices $\mathbf{A}_i$ and $\mathbf{A}_{i+}$ on page 35 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 $\mathsf{S}_{>h}\backslash\mathsf{B}_h$ are connected to each other. The property is shown in the matrix below:

$$\begin{array}{cccc} & \mathsf{S}_h & \mathsf{B}_h & \mathsf{S}_{>h}\backslash\mathsf{B}_h \\ \mathsf{S}_h & \times & \times & \mathbf{0} \\ \mathsf{B}_h & \times & \times & \times \\ \mathsf{S}_{>h}\backslash\mathsf{B}_h & \mathbf{0} & \times & \times \end{array}$$

2. If (2) holds for all $g = j$ such that $\mathsf{S}_j < \mathsf{S}_k$, then by Property B.1 we have either $\mathsf{C}_j \subset \mathsf{C}_k$ or $\mathsf{C}_j \cap \mathsf{C}_k = \emptyset$.

   - if $\mathsf{C}_j \subset \mathsf{C}_k$, consider u such that $\mathsf{S}_k < \mathsf{S}_u$. By Property B.6, we have $\mathsf{I}_k \cap \mathsf{S}_u = \emptyset$. Since $\mathsf{C}_k = \mathsf{I}_k \cup \mathsf{B}_k$, we have $(\mathsf{S}_u\backslash\mathsf{B}_k) \cap \mathsf{C}_k = \emptyset$. So we have $\mathsf{B}_j \cap (\mathsf{S}_u\backslash\mathsf{B}_k) \subseteq (\mathsf{S}_u\backslash\mathsf{B}_k) \cup \mathsf{C}_j \subseteq (\mathsf{S}_u\backslash\mathsf{B}_k) \cap \mathsf{C}_k = \emptyset \Rightarrow \mathsf{B}_j \cap (\mathsf{S}_{>k}\backslash\mathsf{B}_k) = \emptyset$.
   - if $\mathsf{C}_j \cap \mathsf{C}_k = \emptyset$, then $\mathsf{B}_j \subset \mathsf{C}_j$ and $\mathsf{S}_k \subset \mathsf{C}_k \Rightarrow \mathsf{B}_j \cap \mathsf{S}_k = \emptyset$.

   So in both cases, we have $(\mathsf{B}_j, \mathsf{B}_j) \cap (\mathsf{S}_k, \mathsf{S}_{>k}\backslash\mathsf{B}_k) = \emptyset$. Since for every $\mathsf{S}_j < \mathsf{S}_k$, (1), (2), (3), and (4) hold, i.e., eliminating the S_j columns only affects the $(\mathsf{B}_j, \mathsf{B}_j)$ entries, we have $\mathbf{A}_k(\mathsf{S}_k, \mathsf{S}_{>k}\backslash\mathsf{B}_k) = \mathbf{A}_k(\mathsf{S}_{>k}\backslash\mathsf{B}_k, \mathsf{S}_k) = \mathbf{0}$. Since the argument is valid for all $h \geq k$, we have $\forall h \geq k$, $\mathbf{A}_k(\mathsf{S}_h, \mathsf{S}_{>h}\backslash\mathsf{B}_h) = \mathbf{A}_k(\mathsf{S}_{>h}\backslash\mathsf{B}_h, \mathsf{S}_h) = \mathbf{0}$. So (2) holds for $g = k$ as well.

By strong mathematical induction, we have that (2) holds for all g such that $\mathsf{C}_g \in \mathsf{T}_r^+$. □

Below we restate Corollaries 3.1–3.3 and give their proofs.

Corollary B.1 *If C_i and C_j are the two children of C_k, then $\mathbf{A}_k(\mathsf{B}_i, \mathsf{B}_j) = \mathbf{A}(\mathsf{B}_i, \mathsf{B}_j)$ and $\mathbf{A}_k(\mathsf{B}_j, \mathsf{B}_i) = \mathbf{A}(\mathsf{B}_j, \mathsf{B}_i)$.*

Proof: Without loss of generality, let $\mathsf{S}_i < \mathsf{S}_j$. For any $\mathsf{S}_u < \mathsf{S}_k$, consider the following three cases: $\mathsf{S}_u < \mathsf{S}_j$, $\mathsf{S}_u = \mathsf{S}_j$, and $\mathsf{S}_j < \mathsf{S}_u < \mathsf{S}_k$.

If $\mathsf{S}_u < \mathsf{S}_j$, then by Property B.1, either $\mathsf{C}_u \subset \mathsf{C}_j$ or $\mathsf{C}_u \cap \mathsf{C}_j = \emptyset$.

- if $\mathsf{C}_u \subset \mathsf{C}_j$, then since $\mathsf{C}_i \cap \mathsf{C}_j = \emptyset$, we have $\mathsf{C}_u \cap \mathsf{C}_i = \emptyset \Rightarrow \mathsf{B}_u \cap \mathsf{B}_i = \emptyset$;
- if $\mathsf{C}_u \cap \mathsf{C}_j = \emptyset$, then since $\mathsf{B}_u \subset \mathsf{C}_u$ and $\mathsf{B}_j \subset \mathsf{C}_j$, we have $\mathsf{B}_u \cap \mathsf{B}_j = \emptyset$.

So we have $(\mathsf{B}_u, \mathsf{B}_u) \cap (\mathsf{B}_i, \mathsf{B}_j) = (\mathsf{B}_u, \mathsf{B}_u) \cap (\mathsf{B}_j, \mathsf{B}_i) = \emptyset$.

If $\mathsf{S}_u = \mathsf{S}_j$, then $\mathsf{B}_u = \mathsf{B}_j$ and $\mathsf{B}_i \cap \mathsf{B}_j = \emptyset$, so we also have $(\mathsf{B}_u, \mathsf{B}_u) \cap (\mathsf{B}_i, \mathsf{B}_j) = (\mathsf{B}_u, \mathsf{B}_u) \cap (\mathsf{B}_j, \mathsf{B}_i) = \emptyset$.

If $\mathsf{S}_j < \mathsf{S}_u < \mathsf{S}_k$, then by Property B.7, we have $\mathsf{B}_u \cap \mathsf{B}_j = \emptyset$. So we have $(\mathsf{B}_u, \mathsf{B}_u) \cap (\mathsf{B}_i, \mathsf{B}_j) = (\mathsf{B}_u, \mathsf{B}_u) \cap (\mathsf{B}_j, \mathsf{B}_i) = \emptyset$ as well.

So for every $\mathsf{S}_u < \mathsf{S}_k$, we have $(\mathsf{B}_u, \mathsf{B}_u) \cap (\mathsf{B}_i, \mathsf{B}_j) = (\mathsf{B}_u, \mathsf{B}_u) \cap (\mathsf{B}_j, \mathsf{B}_i) = \emptyset$. By Theorem B.1, eliminating S_u only changes $\mathbf{A}(\mathsf{B}_u, \mathsf{B}_u)$, so we have $\mathbf{A}_k(\mathsf{B}_i, \mathsf{B}_j) = \mathbf{A}(\mathsf{B}_i, \mathsf{B}_j)$ and $\mathbf{A}_k(\mathsf{B}_j, \mathsf{B}_i) = \mathbf{A}(\mathsf{B}_j, \mathsf{B}_i)$. □

Corollary B.2 *If C_i is a child of C_k, then $\mathbf{A}_k(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{A}_{i+}(\mathsf{B}_i, \mathsf{B}_i)$.*

Proof: Consider u such that $\mathsf{S}_i < \mathsf{S}_u < \mathsf{S}_k$. By Property B.7, we have $\mathsf{B}_u \cap \mathsf{B}_i = \emptyset$. By Theorem B.1, eliminating S_u columns will only affect $(\mathsf{B}_u, \mathsf{B}_u)$ entries, and so we have $\mathbf{A}_k(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{A}_{i+}(\mathsf{B}_i, \mathsf{B}_i)$. □

Corollary B.3 *If C_i is a leaf node in T_r^+, then $\mathbf{A}_i(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{A}(\mathsf{C}_i, \mathsf{C}_i)$.*

Proof: If $\mathsf{S}_u < \mathsf{S}_i$, by Property B.1 we have either $\mathsf{C}_u \subset \mathsf{C}_i$ or $\mathsf{C}_u \cap \mathsf{C}_i = \emptyset$. Since C_i is a leaf node, there is no u such that $\mathsf{C}_u \subset \mathsf{C}_i$. So we have $\mathsf{C}_u \cap \mathsf{C}_i = \emptyset \Rightarrow \mathsf{B}_u \cap \mathsf{C}_i = \emptyset$. By Theorem B.1, we have $\mathbf{A}_i(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{A}(\mathsf{C}_i, \mathsf{C}_i)$. □

B.2 Proofs for computing $\mathbf{G}^<$

This section is organized similarly to the previous section.

Theorem B.2 *If we perform Gaussian elimination as described in Section 3.2 and update $\mathbf{R}_g$ accordingly, then we have:*

1. $\forall h \geq g, \mathbf{R}_g(\mathsf{S}_h, \mathsf{S}_{>h} \backslash \mathsf{B}_h) = \mathbf{R}_g(\mathsf{S}_{>h} \backslash \mathsf{B}_h, \mathsf{S}_h) = 0$;
2. (a) $\mathbf{R}_{g+}(\mathsf{B}_g, \mathsf{S}_{>g} \backslash \mathsf{B}_g) = \mathbf{R}_g(\mathsf{B}_g, \mathsf{S}_{>g} \backslash \mathsf{B}_g)$;

  (b) $\mathbf{R}_{g+}(\mathsf{S}_{>g}\backslash\mathsf{B}_g, \mathsf{B}_g) = \mathbf{R}_g(\mathsf{S}_{>g}\backslash\mathsf{B}_g, \mathsf{B}_g)$;

  (c) $\mathbf{R}_{g+}(\mathsf{S}_{>g}\backslash\mathsf{B}_g, \mathsf{S}_{>g}\backslash\mathsf{B}_g) = \mathbf{R}_g(\mathsf{S}_{>g}\backslash\mathsf{B}_g, \mathsf{S}_{>g}\backslash\mathsf{B}_g)$;

3. $\mathbf{R}_{g+}(\mathsf{B}_g, \mathsf{B}_g) = \mathbf{R}_g(\mathsf{B}_g, \mathsf{B}_g) - \mathbf{R}_g(\mathsf{B}_g, \mathsf{S}_g)\mathbf{R}_g(\mathsf{S}_g, \mathsf{S}_g)^{-1}\mathbf{R}_g(\mathsf{S}_g, \mathsf{B}_g)$.

In short, applying the $\mathcal{L}_g^{-1}$ and $\mathcal{L}_g^{-\dagger}$ will only affect the $(\mathsf{B}_g, \mathsf{B}_g)$, $(\mathsf{S}_g, \mathsf{B}_g)$, and $(\mathsf{B}_g, \mathsf{S}_g)$ entries in $\mathbf{R}_g$, as shown in the previous matrices.

Proof: Since (1) implies (2) and (3) for each g, it suffices to prove (1). We will prove (1) by strong mathematical induction.

For $g = i_1$, (1) holds because an entry in the original matrix is nonzero iff the corresponding two nodes connect to each other and no nodes in S_h and $\mathsf{S}_{>h}\backslash\mathsf{B}_h$ are connected to each other.

If (1) holds for all $g = j$ such that $\mathsf{S}_j < \mathsf{S}_k$, then by Property B.1 we have either $\mathsf{C}_j \subset \mathsf{C}_k$ or $\mathsf{C}_j \cap \mathsf{C}_k = \emptyset$.

- If $\mathsf{C}_j \subset \mathsf{C}_k$, consider u such that $\mathsf{S}_k < \mathsf{S}_u$. By Property B.6, we have $\mathsf{I}_k \cap \mathsf{S}_u = \emptyset$. Since $\mathsf{C}_k = \mathsf{I}_k \cup \mathsf{B}_k$, we have $(\mathsf{S}_u\backslash\mathsf{B}_k) \cap \mathsf{C}_k = \emptyset$. So we have $\mathsf{B}_j \cap (\mathsf{S}_u\backslash\mathsf{B}_k) \subseteq (\mathsf{S}_u\backslash\mathsf{B}_k) \cap \mathsf{C}_j \subseteq (\mathsf{S}_u\backslash\mathsf{B}_k) \cap \mathsf{C}_k = \emptyset \Rightarrow \mathsf{B}_j \cap (\mathsf{S}_{>k}\backslash\mathsf{B}_k) = \emptyset$.
- If $\mathsf{C}_j \cap \mathsf{C}_k = \emptyset$, then $\mathsf{B}_j \subset \mathsf{C}_j$ and $\mathsf{S}_k \subset \mathsf{C}_k \Rightarrow \mathsf{B}_j \cap \mathsf{S}_k = \emptyset$.

So in both cases, we have $(\mathsf{B}_j, \mathsf{B}_j) \cap (\mathsf{S}_k, \mathsf{S}_{>k}\backslash\mathsf{B}_k) = \emptyset$. In addition, by Property B.4, for $j \neq k$ we have $\mathsf{S}_j \cap \mathsf{S}_k = \emptyset$. Since for every $\mathsf{S}_j < \mathsf{S}_k$, (1), (2), and (3) hold, we have $\mathbf{R}_k(\mathsf{S}_k, \mathsf{S}_{>k}\backslash\mathsf{B}_k) = \mathbf{R}_k(\mathsf{S}_{>k}\backslash\mathsf{B}_k, \mathsf{S}_k) = \mathbf{0}$. Since the argument is valid for all $h \geq k$, we have $\forall h \geq k$, $\mathbf{R}_k(\mathsf{S}_h, \mathsf{S}_{>h}\backslash\mathsf{B}_h) = \mathbf{R}_k(\mathsf{S}_{>h}\backslash\mathsf{B}_h, \mathsf{S}_h) = \mathbf{0}$. So (1) holds for $g = k$ as well.

By strong mathematical induction, we have that (1) holds for all g such that $\mathsf{C}_k \in \mathsf{T}_r^+$. $\square$

Corollary B.4 *If C_i and C_j are the two children of C_k , then $\mathbf{R}_k(\mathsf{B}_i, \mathsf{B}_j) = \mathbf{R}(\mathsf{B}_i, \mathsf{B}_j)$ and $\mathbf{R}_k(\mathsf{B}_j, \mathsf{B}_i) = \mathbf{R}(\mathsf{B}_j, \mathsf{B}_i)$.*

Proof: The key point of the proof is to show

$$(\mathsf{B}_u, \mathsf{B}_u) \cap (\mathsf{B}_i, \mathsf{B}_j) = (\mathsf{S}_u, \mathsf{B}_u) \cap (\mathsf{B}_i, \mathsf{B}_j) = (\mathsf{B}_u, \mathsf{S}_u) \cap (\mathsf{B}_i, \mathsf{B}_j) = \emptyset \tag{B.1}$$

and

$$(\mathsf{B}_u, \mathsf{B}_u) \cap (\mathsf{B}_j, \mathsf{B}_i) = (\mathsf{S}_u, \mathsf{B}_u) \cap (\mathsf{B}_j, \mathsf{B}_i) = (\mathsf{B}_u, \mathsf{S}_u) \cap (\mathsf{B}_j, \mathsf{B}_i) = \emptyset. \tag{B.2}$$

If $\mathsf{S}_u \leq \mathsf{S}_j$, then by Property B.1, either $\mathsf{C}_u \subseteq \mathsf{C}_j$ or $\mathsf{C}_u \cap \mathsf{C}_j = \emptyset$.

- If $\mathsf{C}_u \subseteq \mathsf{C}_j$, then since $\mathsf{C}_i \cap \mathsf{C}_j = \emptyset$, we have $\mathsf{C}_u \cap \mathsf{C}_i = \emptyset \Rightarrow \mathsf{B}_u \cap \mathsf{B}_i$ and $\mathsf{S}_u \cap \mathsf{B}_i = \emptyset$.
- If $\mathsf{C}_u \cap \mathsf{C}_j = \emptyset$, then $\mathsf{B}_u \cap \mathsf{B}_j = \emptyset$ and $\mathsf{S}_u \cap \mathsf{B}_j = \emptyset$.

So (1) and (2) hold.

If $\mathsf{S}_j < \mathsf{S}_u < \mathsf{S}_k$, then by Property B.7, we have $\mathsf{C}_u \cap \mathsf{C}_j = \emptyset \Rightarrow \mathsf{S}_u \cap \mathsf{B}_j = \mathsf{B}_u \cap \mathsf{B}_j = \emptyset$. So (1) and (2) hold as well.

So for every $\mathsf{S}_u < \mathsf{S}_k$, both (1) and (2) hold. By Theorem B.2, the step u of the updates only changes $\mathbf{R}_u(\mathsf{B}_u, \mathsf{B}_u), \mathbf{R}_u(\mathsf{B}_u, \mathsf{S}_u)$, and $\mathbf{R}_u(\mathsf{S}_u, \mathsf{B}_u)$. So we have $\mathbf{R}_k(\mathsf{B}_i, \mathsf{B}_j) = \mathbf{R}(\mathsf{B}_i, \mathsf{B}_j)$ and $\mathbf{R}_k(\mathsf{B}_j, \mathsf{B}_i) = \mathbf{R}(\mathsf{B}_j, \mathsf{B}_i)$. □

Corollary B.5 *If C_i is a child of C_k, then $\mathbf{R}_k(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{R}_{i+}(\mathsf{B}_i, \mathsf{B}_i)$.*

Proof: Consider u such that $\mathsf{S}_i < \mathsf{S}_u < \mathsf{S}_k$. By Property B.7, we have $\mathsf{C}_u \cap \mathsf{C}_i = \emptyset$ and thus (1) and (2) hold. By Theorem B.2, the step u of the updates only changes $\mathbf{R}_u(\mathsf{B}_u, \mathsf{B}_u), \mathbf{R}_u(\mathsf{B}_u, \mathsf{S}_u)$, and $\mathbf{R}_u(\mathsf{S}_u, \mathsf{B}_u)$. So we have $\mathbf{R}_k(\mathsf{B}_i, \mathsf{B}_j) = \mathbf{R}(\mathsf{B}_i, \mathsf{B}_j)$ and $\mathbf{R}_k(\mathsf{B}_j, \mathsf{B}_i) = \mathbf{R}(\mathsf{B}_j, \mathsf{B}_i)$. □

Corollary B.6 *If C_i is a leaf node in T_r^+, then $\mathbf{R}_i(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{R}(\mathsf{C}_i, \mathsf{C}_i)$.*

Proof: Consider $\mathsf{S}_u < \mathsf{S}_i$. By Property B.1, either $\mathsf{C}_u \subset \mathsf{C}_i$ or $\mathsf{C}_u \cap \mathsf{C}_i = \emptyset$. Since C_i is a leaf node, there is no u such that $\mathsf{C}_u \subset \mathsf{C}_i$. So we have $\mathsf{C}_u \cap \mathsf{C}_i = \emptyset$ and thus $\mathsf{B}_u \cap \mathsf{C}_i = \mathsf{S}_u \cap \mathsf{C}_i = \emptyset$. By Theorem B.2, we have $\mathbf{R}_i(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{R}(\mathsf{C}_i, \mathsf{C}_i)$. □

Theorem B.3 *For any r and s such that $\mathsf{C}_i \in \mathsf{T}_r^+$ and $\mathsf{C}_i \in \mathsf{T}_s^+$, we have*

$$\mathbf{R}_{r,i}(\mathsf{S}_i \cup \mathsf{B}_i, \mathsf{S}_i \cup \mathsf{B}_i) = \mathbf{R}_{s,i}(\mathsf{S}_i \cup \mathsf{B}_i, \mathsf{S}_i \cup \mathsf{B}_i).$$

Proof: If C_i is a leaf node, then by Corollary B.6, we have $\mathbf{R}_{r,i}(\mathsf{S}_i \cup \mathsf{B}_i, \mathsf{S}_i \cup \mathsf{B}_i) = \mathbf{R}_{r,i}(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{R}_r(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{R}_s(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{R}_{s,i}(\mathsf{C}_i, \mathsf{C}_i) = \mathbf{R}_{s,i}(\mathsf{S}_i \cup \mathsf{B}_i, \mathsf{S}_i \cup \mathsf{B}_i)$

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 B.2, we have $\mathbf{R}_{r,i+}(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{R}_{s,i+}(\mathsf{B}_i, \mathsf{B}_i)$ and $\mathbf{R}_{r,j+}(\mathsf{B}_j, \mathsf{B}_j) = \mathbf{R}_{s,j+}(\mathsf{B}_j, \mathsf{B}_j)$;
- by Corollary B.5, we have $\mathbf{R}_{r,k}(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{R}_{r,i+}(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{R}_{s,i+}(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{R}_{s,k}(\mathsf{B}_i, \mathsf{B}_i)$ and $\mathbf{R}_{r,k}(\mathsf{B}_j, \mathsf{B}_j) = \mathbf{R}_{r,j+}(\mathsf{B}_j, \mathsf{B}_j) = \mathbf{R}_{s,j+}(\mathsf{B}_j, \mathsf{B}_j) = \mathbf{R}_{s,k}(\mathsf{B}_j, \mathsf{B}_j)$;
- by Corollary B.4, we have $\mathbf{R}_{r,k}(\mathsf{B}_i, \mathsf{B}_j) = \mathbf{R}_r(\mathsf{B}_i, \mathsf{B}_j) = \mathbf{R}_s(\mathsf{B}_i, \mathsf{B}_j) = \mathbf{R}_{s,k}(\mathsf{B}_i, \mathsf{B}_j)$ and $\mathbf{R}_{r,k}(\mathsf{B}_j, \mathsf{B}_i) = \mathbf{R}_r(\mathsf{B}_j, \mathsf{B}_i) = \mathbf{R}_s(\mathsf{B}_j, \mathsf{B}_i) = \mathbf{R}_{s,k}(\mathsf{B}_j, \mathsf{B}_i)$.

Now we have $\mathbf{R}_{r,k}(\mathsf{B}_i \cup \mathsf{B}_j, \mathsf{B}_i \cup \mathsf{B}_j) = \mathbf{R}_{s,k}(\mathsf{B}_i \cup \mathsf{B}_j, \mathsf{B}_i \cup \mathsf{B}_j)$. By Property B.3, we have $\mathbf{R}_{r,k}(\mathsf{S}_k \cup \mathsf{B}_k, \mathsf{S}_k \cup \mathsf{B}_k) = \mathbf{R}_{s,k}(\mathsf{S}_k \cup \mathsf{B}_k, \mathsf{S}_k \cup \mathsf{B}_k)$. By induction, the theorem is proved. $\square$

Corollary B.7 *For any r and s such that $\mathsf{C}_i \in \mathsf{T}_r^+$ and $\mathsf{C}_i \in \mathsf{T}_s^+$, we have*

$$\mathbf{R}_{r,i+}(\mathsf{B}_i, \mathsf{B}_i) = \mathbf{R}_{s,i+}(\mathsf{B}_i, \mathsf{B}_i).$$

Proof: By Theorem B.2 and Theorem B.3. $\square$

Appendix C

Algorithms

This appendix presents the algorithms of block cyclic reduction and the hybrid method used in Chapter 6. The algorithm InverseBCR given in Alg. C.1 calculates the block tridiagonal portion of the inverse of $\mathbf{A}$, namely $\mathrm{Trid}_{\mathbf{A}}\{\mathbf{G}\}$, via a pure block cyclic reduction (BCR) approach. The algorithm performs a BCR reduction on $\mathbf{A}$ 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 $\mathbf{G}$ using this first block of the inverse $\mathbf{g}_{kk}$, and the stored LU factors in $\mathbf{L}$ and $\mathbf{U}$.

Algorithm C.1: InverseBCR($\mathbf{A}$)

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

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

This variable $\mathbf{i}^{\mathrm{BCR}}$ 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*, which store the value of the top and bottom row indices "owned" by one of the participating parallel processes.

Algorithm C.2: InverseHybrid($\mathbf{A}, \mathbf{i}^{\mathrm{BCR}}$)

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

C.1 BCR Reduction Functions

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

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 C.3: ReduceBCR($\mathbf{A}, \mathbf{L}, \mathbf{U}, \mathbf{i}^{\mathrm{BCR}}$)

```
1: $k \leftarrow$ length of $\mathbf{i}^{\mathrm{BCR}}$ {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:     $\mathbf{i}^{\mathrm{elim}} \leftarrow$ determine the active rows
5:     for $row = 1$ up to length of $\mathbf{i}^{\mathrm{elim}}$ do {eliminate odd active rows}
6:         $\mathbf{A}, \mathbf{L}, \mathbf{U} \leftarrow$ Reduce($\mathbf{A}, \mathbf{L}, \mathbf{U}, row, level, \mathbf{i}^{\mathrm{elim}}$)
7: return $\mathbf{A}, \mathbf{L}, \mathbf{U}$
```

The core operation of the reduction phase of BCR are the block row updates performed by the method Reduce given in Alg. C.4, where the full system $\mathbf{A}$ is supplied along with the LU block matrices $\mathbf{L}$ and $\mathbf{U}$. 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 toward. The method then looks at neighboring 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 C.4: Reduce($\mathbf{A}, \mathbf{L}, \mathbf{U}$, *row*, *level*, $\mathbf{i}^{\text{elim}}$)

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

C.2 BCR Production Functions

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

The algorithm works by traversing each level of the elimination tree (line 1), where an appropriate stride 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 C.5: ProduceBCR($\mathbf{A}, \mathbf{L}, \mathbf{U}, \mathbf{G}, \mathbf{i}^{\text{BCR}}$)

```
for $level = h$ down to 1 do
    $stride \leftarrow 2^{level-1}$
    $\mathbf{i}^{\text{prod}} \leftarrow$ determine the rows to be produced
    for $i = 1$ up to length of $\mathbf{i}^{\text{prod}}$ do
        $k_{\text{to}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i]]$
        if $i = 1$ then
            $k_{\text{from}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i] + stride]$
            $\mathbf{G} \leftarrow$ CornerProduce($\mathbf{A}, \mathbf{L}, \mathbf{U}, \mathbf{G}, k_{\text{from}}, k_{\text{to}}$)
        if $i \neq 1$ and $i =$ length of $\mathbf{i}^{\text{prod}}$ then
            if $\mathbf{i}^{\text{prod}}[\text{end}] \leq$ length of $\mathbf{i}^{\text{BCR}} - stride$ then
                $k_{\text{above}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i] - stride]$
                $k_{\text{below}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i] + stride]$
                $\mathbf{G} \leftarrow$ CenterProduce($\mathbf{A}, \mathbf{L}, \mathbf{U}, \mathbf{G}, k_{\text{above}}, k_{\text{to}}, k_{\text{below}}$)
            else
                $k_{\text{from}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i] - stride]$
                $\mathbf{G} \leftarrow$ CornerProduce($\mathbf{A}, \mathbf{L}, \mathbf{U}, \mathbf{G}, k_{\text{from}}, k_{\text{to}}$)
        if $i \neq 1$ and $i \neq$ length of $\mathbf{i}^{\text{prod}}$ then
            $k_{\text{above}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i] - stride]$
            $k_{\text{below}} \leftarrow \mathbf{i}^{\text{BCR}}[\mathbf{i}^{\text{prod}}[i] + stride]$
            $\mathbf{G} \leftarrow$ CenterProduce($\mathbf{A}, \mathbf{L}, \mathbf{U}, \mathbf{G}, k_{\text{above}}, k_{\text{to}}, k_{\text{below}}$)
return $\mathbf{G}$
```

The auxiliary methods CornerProduce and CenterProduce are given in Alg. C.6 and Alg. C.7.

Algorithm C.6: CornerProduce($\mathbf{A}, \mathbf{L}, \mathbf{U}, \mathbf{G}, k_{\text{from}}, k_{\text{to}}$)

```
$\mathbf{g}_{k_{\text{from}},k_{\text{to}}} \leftarrow -\mathbf{g}_{k_{\text{from}},k_{\text{from}}} \boldsymbol{\ell}_{k_{\text{from}},k_{\text{to}}}$
$\mathbf{g}_{k_{\text{to}},k_{\text{from}}} \leftarrow -\mathbf{u}_{k_{\text{to}},k_{\text{from}}} \mathbf{g}_{k_{\text{from}},k_{\text{from}}}$
$\mathbf{g}_{k_{\text{to}},k_{\text{to}}} \leftarrow \mathbf{a}^{-1}_{k_{\text{to}},k_{\text{to}}} - \mathbf{g}_{k_{\text{to}},k_{\text{from}}} \boldsymbol{\ell}_{k_{\text{from}},k_{\text{to}}}$
return $\mathbf{G}$
```

Algorithm C.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 $\mathbf{G}$
```

C.3 Hybrid Auxiliary Functions

Finally, this section 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 Alg. C.8, while the final production phase to generate the final block tridiagonal $\text{Trid}_{\mathbf{A}}\{\mathbf{G}\}$ is done by PRODUCESCHUR in Alg. C.9.

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 Alg. C.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 C.8: ReduceSchur($\mathbf{A}$)

```
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 $\mathbf{A}, \mathbf{L}, \mathbf{U}$
```

Algorithm C.9: ProduceSchur($\mathbf{A}, \mathbf{L}, \mathbf{U}, \mathbf{G}$)

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

Bibliography

M. P. Anantram, Shaikh S. Ahmed, Alexei Svizhenko, Derrick Kearney, and Gerhard Klimeck. NanoFET. doi: 10254/nanohub-r1090.5, 2007. 49

K. Bowden. A direct solution to the block tridiagonal matrix inversion problem. *International Journal of General Systems*, 15(3):185–98, 1989. 12

J. R. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*, 31:162–79, 1977. 74

J. R. Bunch and B. N. Parlett. Direct methods for solving symmetric indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 8:639–55, 1971. 74

S. Datta. *Electronic Transport in Mesoscopic Systems.* Cambridge University Press, 1997. 3, 5, 7

S. Datta. Nanoscale device modeling: the Green's function method. *Superlattices and Microstructures*, 28(4):253–278, 2000. 3, 4, 8

T. A. Davis. *Direct Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, 2006. 12

A. M. Erisman and W. F. Tinney. On computing certain elements of the inverse of a sparse matrix. *Numerical Mathematics*, 18(3):177–79, 1975. 11, 20, 21

D. K. Ferry and S. M. Goodnick. *Transport in Nanostructures.* Cambridge University Press, 1997. 5

Walter Gander and Gene Golub. Cyclic reduction - history and applications. In *Scientific Computing: Proceedings of the Workshop 10-12 March 1997*, pages 73–85, 1997. 104

A. George. Nested dissection of a regular finite-element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–63, 1973. 18, 23, 29, 30, 39, 91, 92, 93

G. H. Golub and C. F. Van Loan. *Matrix Computations, Third Edition*, chapter 4.2.1, page 141. Johns Hopkins University Press, 1996a. 73

G. H. Golub and C. F. Van Loan. *Matrix Computations, Third Edition*, chapter 4.4, page 161. Johns Hopkins University Press, 1996b. 74

G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 2nd edition edition, 1989. 106

L. A. Hageman and R. S. Varga. Block iterative methods for cyclically reduced matrix equations. *Numerische Mathematik*, 6:106–119, 1964. 83

S. Hasan, J. Wang, and M. Lundstrom. Device design and manufacturing issues for 10 nm-scale MOSFETs: a computational study. *Solid-State Electronics*, 48:867–875, 2004. 2

Don Heller. Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems. *SIAM Journal of Numerical Analysis*, 13(4):484–496, 1976. 104

R. Lake, G. Klimeck, R. C. Bowen, and D. Jovanovic. Single and multiband modeling of quantum electron transport through layered semiconductor devices. *Journal of Applied Physics*, 81:7845, 1997. 3

S. Li and E. Darve. Optimization of the FIND algorithm to compute the inverse of a sparse matrix. In *International Workshop on Computational Electronics, Beijing, China*, 27-29, May 2009. 94

S. Li, S. Ahmed, G. Klimeck, and E. Darve. Computing entries of the inverse of a sparse matrix using the FIND algorithm. *Journal of Computational Physics*, 227: 9408–9427, 2008. URL `http://dx.doi.org/10.1016/j.jcp.2008.06.033`. 91, 93

H. Niessner and K. Reichert. On computing the inverse of a sparse matrix. *International Journal for Numerical Methods in Engineering*, 19:1513–1526, 1983. 20

G. Peters and J. Wilkinson. *Linear Algebra, Handbook for Automatic Computation, Vol. II*, chapter The calculation of specified eigenvectors by inverse iteration, pages 418–439. Springer-Verlag, 1971. 10

G. Peters and J. Wilkinson. Inverse iteration, ill-conditioned equations and Newton's method. *SIAM Review*, 21:339–360, 1979. 10

D. E. Petersen. *Block Tridiagonal Matrices in Electronic Structure Calculations*. PhD thesis, Dept. of Computer Science, Copenhagen University, 2008. 106

R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8, 1982. 12

J Schröder. Zur Lösung von Potentialaufgaben mit Hilfe des Differenzenverfahrens. *Zeitschrift für Angewandte Mathematik und Mechanik*, 34:241–253, 1954. 83

G. Shahidi. SOI technology for the GHz era. *IBM Journal of Research and Development*, 46:121–131, 2002. 2

SIA. *International Technology Roadmap for Semiconductors 2001 Edition.* Semiconductor Industry Association (SIA), 2706 Montopolis Drive, Austin, Texas 78741, 2001. URL `http://www.itrs.net/Links/2001ITRS/Home.htm`. 3, 47

A. Svizhenko, M. P. Anantram, T. R. Govindan, and B. Biegel. Two-dimensional quantum mechanical modeling of nanotransistors. *Journal of Applied Physics*, 91 (4):2343–54, 2002. 4, 5, 7, 8, 9, 13, 14, 20, 21, 49, 60, 72, 83

K. Takahashi, J. Fagan, and M.-S. Chin. Formation of a sparse bus impedance matrix and its application to short circuit study. In *8th PICA Conf. Proc.*, pages 63–69, Minneapolis, Minn., June 4–6 1973. 10, 20, 83, 108

J. Varah. The calculation of the eigenvectors of a general complex matrix by inverse iteration. *Math. Comp.*, 22:785–791, 1968. 10

D. Vasileska and 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*, 52:227, 2005. 2

R. Venugopal, Z. Ren, S. Datta, M. S. Lundstrom, and D. Jovanovic. Simulating quantum transport in nanoscale transistors: Real versus mode-space approaches. *Journal of Applied Physics*, 92(7):3730–9, OCT 2002. 3, 4

T. J. Walls, V. A. Sverdlov, and K. K. Likharev. Nanoscale SOI MOSFETs: a comparison of two options. *Solid-State Electronics*, 48:857–865, 2004. 2

J. Welser, J. L. Hoyt, and J. F. Gibbons. NMOS and PMOS transistors fabricated in strained silicon/relaxed silicon-germanium structures. *IEDM Technical Digest*, page 1000, 1992. 2

J. Wilkinson. Inverse iteration in theory and practice. In *Symposia Matematica, Vol. X*, pages 361–379. Institutio Nationale di Alta Matematica Monograf, Bologna, Italy, 1972. 10

H. S. Wong. Beyond the conventional transistor. *IBM Journal of Research and Development*, 46:133–168, 2002. 1, 2