High performance selected inversion methods for sparse matrices

Direct and stochastic approaches to selected inversion

Doctoral Dissertation submitted to the

Faculty of Informatics of the Università della Svizzera italiana

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

presented by

Fabio Verbosio

under the supervision of

Olaf Schenk

February 2019

Dissertation Committee

Dissertation accepted on 25 February 2019

Research Advisor	PhD Program Director
Olaf Schenk	**Walter Binder**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Fabio Verbosio
Lugano, 25 February 2019

To my whole family.
In its broadest sense.

Le conoscenze matematiche sono proposizioni costruite dal nostro intelletto in modo da funzionare sempre come vere, o perché sono innate o perché la matematica è stata inventata prima delle altre scienze. E la biblioteca è stata costruita da una mente umana che pensa in modo matematico, perché senza matematica non fai labirinti.

Umberto Eco,
"Il nome della rosa"

Abstract

The explicit evaluation of selected entries of the inverse of a given sparse matrix is an important process in various application fields and is gaining visibility in recent years. While a standard inversion process would require the computation of the whole inverse who is, in general, a dense matrix, state-of-the-art solvers perform a selected inversion process instead. Such approach allows to extract specific entries of the inverse, e.g., the diagonal, avoiding the standard inversion steps, reducing therefore time and memory requirements. Despite the complexity reduction already achieved, the natural direction for the development of the selected inversion software is the parallelization and distribution of the computation, exploiting multinode implementations of the algorithms.

In this work we introduce parallel, high performance selected inversion algorithms suitable for both the computation and estimation of the diagonal of the inverse of large, sparse matrices. The first approach is built on top of a sparse factorization method and a distributed computation of the Schur-complement, and is specifically designed for the parallel treatment of large, dense matrices including a sparse block. The second is based on the stochastic estimation of the matrix diagonal using a stencil-based, matrix-free Krylov subspace iteration. We implement the two solvers and prove their excellent performance on Cray supercomputers, focusing on both the multinode scalability and the numerical accuracy.

Finally, we include the solvers into two distinct frameworks designed for the solution of selected inversion problems in real-life applications. First, we present a parallel, scalable framework for the log-likelihood maximization in genomic prediction problems including marker by environment effects. Then, we apply the matrix-free estimator to the treatment of large-scale three-dimensional nanoelectronic device simulations with open boundary conditions.

Acknowledgements

First of all, I want to thank my advisor, Prof. Dr. Olaf Schenk, for the support he gave me during these five years and for making this work possible. Thank you for helping me understand my limits and my strengths. A special *grazie* to Dr. Drosos Kourounis, my personal co-advisor, for every time we confronted and exchanged opinions and for the open-air work sessions in Villa Luganese. You really helped me evolving as a researcher, thanks!

Thanks to all the Ph.D. committee members for spending some time to analyze and review my work: I really appreciate your effort. In particular, thanks to Prof. Dr. Matthias Bollhöfer for allowing me to work with him and being his guest in Braunschweig.

Thanks to the *ragazze* from the Decanato: Jacinta, Nina, Elisa, Mili, and Nadia. I found in you very special people and great friends, I won't forget how much you helped me. Thanks also to all the Lab people—from the first to the fourth floor—who stood beside me during this long process: Sonia, Alberta and Mario, Davide and Davide, and all the colleagues who enriched me as a person, somehow.

Infine, grazie alla mia famiglia, perché sempre presente, anche a chilometri di distanza, sempre vicino a me e sempre pronta a sorreggermi e a comprendermi. Grazie infinite a Benedetta, per avermi accompagnato con pazienza e coraggio lungo un percorso che più di una volta mi ha visto in difficoltà. Grazie per avermi aiutato a trovare la strada giusta quando credevo di non farcela.

Contents

Figures

Tables

Chapter 1

Introduction

In numerical linear algebra, matrix inversion is one of the most computationally expensive operations. Given an invertible matrix of order n, the computation of the entries of the inverse is equivalent to the solution of n linear systems: once that an LU factorization of the matrix is given, it is possible to compute the columns of the inverse one by one, by solving the linear systems having the columns of the identity as right-hand sides. The LU factorization performed via Gaussian elimination costs $\mathcal{O}(\frac{2}{3}n^3)$ operations, while each of the n linear systems requires $\mathcal{O}(2n^2)$ operations for the forward and backward substitutions, giving a total of $\mathcal{O}(\frac{8}{3}n^3)$ operations. This technique is in general unpractical to apply, so the cost can be reduced by half by diagonalizing the matrix, applying a Gaussian elimination process on the columns, in order to obtain an upper triangular matrix, followed by the same process on the rows, in order to obtain the identity. The whole process then costs a total of $\mathcal{O}(\frac{4}{3}n^3)$ operations. It is clear how this computational burden for the direct evaluation of the whole inverse is still impossible to sustain for large values of n. Many authors have therefore developed alternative strategies, e.g., for the iterative computation of the inverse, such as the Newton method [Ben-Israel, 1965; Pan and Reif, 1989]. Under appropriate assumptions on the spectral properties of the matrix, it has been shown that the Newton methods for the computation of the p-th root of regular functions can be adapted for the computation of the inverse p-th root and specifically, for $p = 1$, the inverse. For more on this, refer to the work by Higham [1986], Söderström and Stewart [1974], Pan and Schreiber [1991], and Lass et al. [2018]. The iterative Newton-based methods have been also specialized for the treatment of structured matrices, such as circulant, Toeplitz and Vandermonde matrices [Pan et al., 2004]. The iterative nature of these algorithms make them also suitable for preconditioning in order to substitute other techniques such as the incomplete LU factorization, in case of a sparse input [Chow and Saad, 1998; Bollhöfer et al., 2019]. Focusing on sparse matrices, it is important to mention that all the algorithms introduced so far have a dense nature, i.e., they do not take into account the presence of a zero pattern for the entries of the matrix, and tend to fill the zero entries with nonzero values.

While dense matrices are commonly used for several applications and exploit the good performance of the optimized libraries and vectorization properties of modern compilers, many mathematical models are designed using sparse matrices. A sparse matrix, in general, is defined as a matrix presenting a consistent number of zero entries. We will call the position of the nonzero entries "nonzero pattern" or "sparsity pattern", equivalently. The usage of sparse matrices entail numerous computational advantages: first, the possibility of storing a compressed version of the matrix which guarantees lower memory requirements. Second, the design of ad hoc sparse algorithms which may reduce the computational complexity taking advantage of the presence of the null entries, allowing the algorithm to skip a number of floating point operations. Considering the numerical algorithm for the factorization of the matrix, the solution of linear systems, or its inversion, it is therefore very important to keep in mind the possibility that the sparsity pattern of the matrix might change while operating on it and to design sparsity-aware methods who keep the nonzero pattern of the matrix as intact as possible. As an example, several mathematical models used in different fields of study such as nanoelectronics [Luisier et al., 2006; Kuzmin et al., 2013; Li and Darve, 2012; Verbosio et al., 2018], uncertainty quantification [Bekas et al., 2009], and genetics [De Coninck et al., 2015, 2016; Verbosio et al., 2017], require the explicit evaluation of selected entries of the inverse of a sparse matrix, e.g., the diagonal elements. This kind of problems are called "selected inversion problems". It is straightforward to see that the whole inversion process would lead to a huge waste of resources, particularly when large-scale datasets are involved. Additionally, even though the input data is compressed, the previously mentioned iterative methods would progressively fill the matrix with new nonzero entries—even though very small in magnitude—unless some sort of sparsification technique is applied, reducing thus the accuracy. In order to fulfil the requirements of a selected inversion problem, different methods and solvers have been designed depending on the nature of the application.

We can divide the selected inversion solvers into two major classes: the direct methods and the stochastic methods. The direct methods are based on the factorization of the original matrix, performing a sparse Gaussian elimination. It is clear how, even from small examples, the elimination process is prone to add additional fill-in nonzero entries (think, e.g., of the case of an arrowhead matrix, where the diagonal, the first row and the first column are filled with nonzeros). In order to add as minimum fill-in entries as possible, the matrix is reordered as a preliminary action, applying for instance graph partitioning techniques; this is called “symbolic factorization” phase. Despite the existence of trivial cases such as the arrowhead matrix mentioned above, where swapping the first and last row and the first and last column would guarantee no fill-in, the minimization of the fill-in entries is still computationally challenging. Both this and the graph partitioning problem are in fact NP-complete [Yannakakis, 1981], requiring the formulation and development of many heuristic methods, such as the minimum degree, nested dissection, or multilevel algorithms [Amestoy et al., 1996; George, 1973; Karypis and Kumar, 1998]. Once that the matrix has been reordered, the sparse LU factorization is computed. From here, one can proceed with evaluation the actual inverse entries. Even though the factors are sparse, the inverse is in general dense [Duff et al., 1988], even when very few fill-in entries are added during the factorization phase. For this reason, many solvers apply the Takahashi’s formula [Takahashi et al., 1973], exploiting the sparsity of the matrix to compute selected entries of the inverse. This formula designs a recursive computation of the entries through forward/backward substitution—as in the dense algorithm—skipping useless computation for the null entries of the LU factors.

The methods in the second class are instead iterative. In particular, we analyze stochastic algorithms for the estimation for the diagonal of the inverse. Given a sparse matrix, Hutchinson [1990] developed an estimator for its trace based on a Monte Carlo method. In a totally analogous way, this estimator has been extended to compute the trace and the diagonal of the inverse of the matrix [Tang and Saad, 2012] and used to accelerate some applications, such as uncertainty quantification [Bekas et al., 2009, 2012] and quantum physics nanodevice simulations [Verbosio et al., 2018]. The diagonal of the inverse of A is computed as a vector of n elements, and it is evaluated as the sample mean of a number of random vectors. Such vectors are computed by sampling the action of A^{-1} on a number of random vectors appropriately generated and mapped onto the image of A^{-1} by an inversion operator, e.g., a Krylov subspace method or some other linear system solving strategy. The mapped vectors, then, represent a sampling of the behavior of A^{-1} and are used to compute the final estimator. This estimator presents the disadvantage of having a variance depending on the off-diagonal (unknown) entries of the inverse, however it presents no bias. Because of the stochasticity and the dependence on the above mentioned inversion operator the estimator is not widely used, and other selected inversion techniques are sometimes preferred. The very low memory requirements and the relatively easy implementation make it still attractive, and we are going to show an innovative implementation further in this document.

This work is divided into three parts. The first (Part I–Selected inversion algorithms) introduces the sparse factorization process, the selected inversion problem, and the algorithms for the evaluation of the diagonal of the inverse, focusing on the Takahashi's formula. This is followed by some probabilistic and stochastic background for the formulation and study of the Hutchinson's stochastic estimator. Then, we propose a parallel version of the Takahashi's approach, especially designed for large-scale sparse matrices, and based on the distributed Schur-complement computation. We conclude the first part with the outline of a parallel, stencil-based, matrix-free formulation of the Monte Carlo estimator.

In Part II–High-performance computing selected inversion, we overview some of the most diffused state-of-the-art solvers used for selected inversion, and review some of their applications. We complete the section with a set of node-level performance tests comparing two of the most used solvers, PARDISO [Kuzmin et al., 2013] and PSelInv [Jacquelin et al., 2016], analyzing their behavior on a dataset of sparse and dense matrices.

Finally, in Part III–Selected inversion applications, we present two applications of the algorithms created in Part I. We include the parallel Takahashi's method in a framework for the maximum-likelihood estimation of parameters in large-scale genomic prediction models; then, using the highly scalable, stencil-based, matrix-free stochastic estimator for the diagonal of the inverse, we evaluate the diagonal of the non-equilibrium Green's function matrices in nanoelectronic three-dimensional device simulations with open boundary conditions.

The main contributions of this work to the existing literature are listed here (the order does not follow the order of the chapters).

- The design of a parallel, scalable, multinode Takahashi's approach suitable for the selected inversion of large-scale dense matrices with a large sparse block (chapter 2), based on the distributed computation of the Schur-complement and the selected inversion of the sparse submatrix.

- The study of the parallel performance of the scalable Takahashi's framework through weak-scaling and strong-scaling tests performed on up to to 400 nodes on a Cray cluster (chapter 6), and its efficiency on large-scale genomic predictions datasets in the average-information maximum likelihood optimization structure (chapter 7).

- The creation of a parallel, scalable stochastic estimator for the diagonal of the inverse of large sparse matrices with stencil-based structure, designed on top of a matrix-free preconditioned conjugate gradient (chapter 3).
- The analysis of the estimator's space parallelization, its scaling performance on up to 1024 tasks on a Cray cluster, and its validation for the evaluation of the diagonal of the retarded Green's function in a nanotransistor simulation problem (chapter 8).
- A comparative study of the scaling achievements of the state-of-the-art selected inversion solvers PARDISO and PSelInv (PEXSI) and their node-level performance on a set of sparse/dense matrices (chapter 5).

Part I

Selected inversion algorithms

Chapter 2

Factorization based selected inversion

In this chapter, we are going to introduce the selected inversion process and analyze the first class of methods called “factorization based” or “direct” methods. Such algorithms are based on the LU factorization of the matrix, therefore it is useful to first recall the standard linear algebra behind it, i.e., the Gaussian elimination. In the first part of the chapter we are going to introduce the formalism necessary to describe the factorization of a general matrix and the definition of a selected inversion process, followed by the reformulation of the factorization algorithm for sparse matrices based on the so called “Takahashi’s formula”, a selected inversion recursive method formulated by Takahashi et al. [1973]. Finally, a parallel, highly scalable version of the Takahashi’s formula will be presented in section 2.7; this approach, based on the parallelized computation of the Schur-complement, will be shown to be effective on a widely used category of sparse matrices. The scaling of the parallel method and its performance on a real-life application will be shown in parts II and III, respectively.

2.1 LU factorization and inversion algorithm

As a first step towards the formulation of a direct selected inversion algorithm, we first recall the linear algebra behind the LU factorization of a matrix. A square, $n \times n$ matrix A over $\mathbb{K}$ ($\mathbb{K}$ indicates $\mathbb{R}$ or $\mathbb{C}$) is invertible if and only if it is row-equivalent (or column-equivalent) to an identity matrix of size n. This means that A can be reduced to an identity matrix, I, by multiplying it on the left by a sequence of elementary matrices. Defining r_i, r_j, and r_k to be the ith, jth, and kth rows of A, $1 \leq j, k \leq n$, $j \neq k$, I the identity matrix of size n and e_j,e_k its jth and kth columns, and fixing $0 \neq \alpha \in \mathbb{K}$, the allowed elementary operations and associated elementary matrices are three:

P_{jk}: rows r_j and r_k are swapped ($r_j \longleftrightarrow r_k$);

$S_j(\alpha)$: row r_j is scaled by a factor α ($r_j \leftarrow \alpha r_j$);

$T_{jk}(\alpha)$: row r_j is added a multiple of row r_k ($r_j \leftarrow r_j + \alpha r_k$).

It is easily verified that P_{jk} is a permutation of I where only the jth and kth rows are exchanged, $S_j(\alpha)$ is obtained by multiplying by α the jth diagonal element of I, i.e., $S_j(\alpha) = I + (\alpha - 1)e_j e_j{}^\top$, while $T_{jk}(\alpha)$ is computed from I, setting the (j,k)th entry to α, i.e., $T_{jk}(\alpha) = I + \alpha e_j e_k{}^\top$. From their definitions, it is straightforward to prove that all the elementary matrices are nonsingular and their inverse are still elementary matrices.

Theorem 1 (Inverse of elementary matrices). *All the elementary matrices are invertible and the inverse are still elementary.*

Proof. It is easy to verify that

$$\begin{aligned} P_{jk}{}^{-1} &= P_{jk}\,, \\ S_j(\alpha)^{-1} &= S_j(\alpha^{-1})\,, \\ T_{jk}(\alpha)^{-1} &= T_{jk}(-\alpha)\,. \end{aligned}$$

□

The process of pre-multiplying A by the elementary matrices just introduced describes the Gaussian elimination process, transforming A into an upper triangular matrix U. There exist, then, $p > 0$ elementary matrices $M_1, \ldots, M_p$ such that

$$M_p M_{p-1} \cdots M_2 M_1 \; A = U \tag{2.1}$$

From this, we can define

$$L = \left(\overleftarrow{\prod_{k=1,\ldots,p}} M_k \right)^{-1} = \prod_{k=1,\ldots,p} M_k{}^{-1} \tag{2.2}$$

to factorize the A into the product $A = LU$. Notice that if no permutation matrix is involved, L is still lower triangular; on the other hand, in the general case, the factorization can be achieved in the form $PA = LU$. For the sake of simplicity, we can assume no pivoting to be needed, resulting in the condition $P = I$.

Recalling now that the invertibility of A allows the Gaussian elimination to be pushed forward in order to transform matrix U into an identity matrix I, there exist $\ell > 0$ elementary matrices $M_{p+1}, \ldots, M_{p+\ell}$ such that

$$\left(M_{p+\ell} M_{p+\ell-1} \cdots M_{p+1}\right) \left(M_p M_{p-1} \cdots M_2 M_1\right) A = I \,. \tag{2.3}$$

Given the factorizion $A = LU$, the reduction to the identity through the Gaussian elimination described in (2.3) entails the same result obtained by the solution via forward and backward substitution of n linear systems of the form

$$LU\, x_k = e_k \tag{2.4}$$

where x_k are the columns of A^{-1}. Fixed a value for k, the system is solved in two stages: first applying a forward substitution rule to the lower triangular system $Ly_k = e_k$, and then applying a backward substitution process to the upper triangular one having y_k (just computed) as a right-hand side, $Ux_k = y_k$. In short,

$$\begin{cases} Ly_k = e_k \\ Ux_k = y_k \end{cases} . \tag{2.5}$$

2.2 Complexity of the inversion algorithm

In order to evaluate the complexity of the inversion algorithm, it is useful to split the inversion process into two parts: the first being the reduction process from A to the upper triangular form (2.1), and the second being the reduction from upper triangular to diagonal (the identity).

Theorem 2 (Complexity of the inversion algorithm)**.** *The complexity for the inversion of an $n \times n$ invertible matrix is $\mathcal{O}\left(\frac{4}{3}n^3\right)$ floating point operations.*

Proof. We can split the computation of the inverse into two parts: (2.1) and (2.3). Every step of the first part (reduction to upper triangular through Gaussian elimination) requires the elimination of all the entries below the diagonal of the original matrix and the scaling of the diagonal itself to obtain a unitriangular matrix U. This means that, for every $k = 1, \ldots, n$, there is the need to compute k additions and k multiplications by $k - 1$ numbers, yielding $2k(k-1)$ operations, and giving a total of $\mathcal{O}\left(\frac{2}{3}n^3\right)$ operations. The second part, analogously, consists in a backward process in order to reduce the triangular matrix to lower triangular, at the same computational cost. The sum of the two quantities gives a total of $\mathcal{O}\left(\frac{4}{3}n^3\right)$ operations. □

It is clear how such elevated computational complexity may lead to prohibitive computing times. The asymptotic behavior we outlined in this section, however, can be adapted to the cases where some of the floating point operations can be avoided in presence of null entries in A, first, and in its factors, later. When A is sparse, in fact, the LU factorization and the inversion process can be performed taking into account the respective sparsity patterns and using sparse algorithms. In the rest of this document, we will consider sparse matrices and discuss how both the time and spatial complexity can be reduced. Notice that we do not explore spatial complexity, yet. Since the algorithms already introduced do not take into account the sparsity, the memory required to store each of A, L, U, and A^{-1} is n^2 double precision numbers so far.

2.3 Sparse Gaussian elimination and selected inversion

In this section we finally give the definition of a selected inversion process and formulate a "sparsity-aware" version of the Gaussian elimination. Given a sparse, invertible, $n \times n$ matrix A with entries in $\mathbb{K}$, we can give the following definition.

Definition 1 (Selected inversion process)**.** *We define as selected inversion process a procedure aiming at the computation or estimation of the entries of the inverse of a given sparse matrix, avoiding the standard (dense) inversion process.*

In other words, we call a selected inversion process any procedure able to evaluate a subset of the entries of the inverse of a given matrix with a complexity that is lower than the one computed in Theorem 2. In details, considering the set of the indices $\mathscr{I} = \{(i,j) \text{ for } i,j = 1,2,\ldots,n\}$, we define the nonzero pattern of A as the set $\mathscr{N} = \{(i,j) \text{ s.t. } A_{ij} \neq 0\}$. A selected inversion process computes the elements of the set

$$\left\{ {A^{-1}}_{ij} \text{ for } (i,j) \in \mathscr{S} \subset \mathscr{I} \right\}, \tag{2.6}$$

where the pattern $\mathscr{S} \subset \mathscr{I}$ describes the position of the desired entries, usually $\mathscr{S} \subset \mathscr{N}$. As an example, in many applications one only needs to compute the diagonal of the inverse of A, hence the selected inversion pattern becomes

$$\mathscr{D} = \{(j,j),\ j = 1,2,\ldots,n\}.$$

In order to approach the sparse selected inversion algorithms, it is useful to consider the LDU factorization of A instead of the LU one. For an invertible matrix A, from (2.1), U is an upper unitriangular matrix, while L is lower triangular but not unitriangular, since the elements on the diagonal might different from 1. In order to make L also unitriangular, it is possible to set $D = \text{diag}(L_{11}, L_{22}, \dots, L_{nn})$ and replace the diagonal entries of L with 1's. Matrix A is then decomposed in the product

$$A = LDU\,. \tag{2.7}$$

We still suppose no pivoting is required. According to [Takahashi et al., 1973], given the LDU factorization of A, it is possible to rearrange the equation above in order to obtain two expressions for its inverse. We denote $X := A^{-1}$ for simplicity. Let us consider the formulas

$$\begin{cases} AX = I \\ XA = I \end{cases} \iff \begin{cases} (LDU)X = I \\ X(LDU) = I \end{cases} \iff \begin{cases} X = D^{-1}L^{-1} + (I-U)X \\ X = U^{-1}D^{-1} + X(I-L) \end{cases} \tag{2.8}$$

Let us consider now the rightmost part of (2.8) and focus on the first identity. We can see that the unknown, X, appears on boths sides of the expression. At a first glance, this may look like an unnecessary complication, however, looking at the expression in details, we see that this describes a recursion formula. On the right-hand side, in fact, $D^{-1}L^{-1}$ is a lower triangular matrix since D is diagonal and L is lower triangular, while the inverse X is multiplied by matrix $(I-U)$. Being U unitriangular, the diagonal of $(I-U)$ is null, hence the factor is strictly upper triangular. Analogously, $U^{-1}D^{-1}$ is upper triangular and $(I-L)$ is strictly lower triangular. From this, then, we can see that (2.8) can be used to calculate the entries of X recursively and the procedure

$$\begin{cases} X_{nn} = (D^{-1}L^{-1})_{nn} = (DL)_{nn}^{\;-1} \\ X_{ij} = \displaystyle\sum_{\substack{k>i \\ U_{ik}\neq 0}} (I-U)_{ik} X_{kj}\,, \quad i = n-1, \dots, 1, \quad j \geq i \\ X_{ij} = \displaystyle\sum_{\substack{k<i \\ L_{kj}\neq 0}} X_{ik} (I-L)_{kj}\,, \quad i = 2, 3, \dots, n, \quad j < i \end{cases} \tag{2.9}$$

fills both the upper triangular (second equation) and lower triangular part (third equation) of X. Notice that the terms $(U^{-1}D^{-1})_{ij}$ do not appear in the third equation because they are zero when $j < i$. Considering that if A is symmetric then its factorization becomes $A = LDL^{\top}$ and X is symmetric too, computing the upper triangular part of X is enough for calculating the whole inverse: considering the second line of the (2.9) and the fact that $X_{ij} = X_{ji}$, the third part becomes redundant.

Looking at (2.9) in details, we notice how some of the recursion steps can be skipped: the addends in $\sum_{k>i}(I-U)_{ik}X_{kj}$ for which $(I-U)_{kj}$ is zero are ignored, and analogously for $\sum_{k<i}X_{ik}(I-L)_{kj}$. The complexity of the algorithm depends strictly on the sparsity pattern of the factors $\mathcal{N}(L)+\mathcal{N}(U)$, therefore a reduction of the fill-in entries appearing during the LDU factorization through the application of reordering strategies on the columns of A is crucial. The sparse recursion formula represents a great improvement with respect to the dense approach, providing a convenient approach to selected inversion for both memory and computing time. Additionally, thinking ahead at the implementation, one natural direction to follow for optimizing the performance is the formulation of a block version, exploiting the computational performance of the dense linear algebra libraries.

2.4 The block Takahashi's inversion algorithm

We consider now a block decomposition of an invertible $n \times n$ matrix C. Here, the integer subindices no longer indicate single entries, but blocks of entries. From the studies on saddle point problems in [Benzi et al., 2005], assuming that C is partitioned as

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A & B \\ E & D \end{pmatrix}, \tag{2.10}$$

for compatible sizes of the subblocks C_{ij}, we can decompose C as the product

$$\begin{aligned} C = \begin{pmatrix} A & B \\ E & D \end{pmatrix} &= \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix}\begin{pmatrix} U_{11} & U_{12} \\ & S \end{pmatrix} \\ &= \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12}+S \end{pmatrix}. \end{aligned} \tag{2.11}$$

Assuming that the factorization $A = L_{11}U_{11}$ is known, we can compute the unknown factors as $U_{12} = (L_{11})^{-1}B$ and $L_{21} = E(U_{11})^{-1}$. Analogously, they can be expressed as the solutions of the linear systems $L_{11}X = B$ and $XU_{11} = E$, respectively. Finally, considering that S is the Schur-complement of A in C, S can be computed as

$$S = D - \left(EU_{11}{}^{-1}\right)\left(L_{11}{}^{-1}B\right) = D - EA^{-1}B\,. \tag{2.12}$$

This expression is intended to be well defined, hence A must be invertible, and the product $A^{-1}B$ is not calculated explicitly, but the corresponding linear system is solved instead. The Schur-complement-based decomposition (2.11) just presented provides an interesting expression for the inverse of C that can be computed blockwise starting from the inverse of the Schur-complement itself. It is reasonable to expect the inverse to be in the form

$$C^{-1} = \begin{pmatrix} C^{-1}{}_{11} & C^{-1}{}_{12} \\ C^{-1}{}_{21} & S^{-1} \end{pmatrix},$$

from which we can extrapolate the following:

$$\begin{array}{lll} \text{(a)} & L_{11}U_{11}\,C^{-1}{}_{11} + L_{11}U_{12}\,C^{-1}{}_{21} & = I \\ \text{(b)} & L_{11}U_{11}\,C^{-1}{}_{12} + L_{11}U_{12}\,S^{-1} & = \mathrm{O} \\ \text{(c)} & L_{21}U_{11}\,C^{-1}{}_{11} + (L_{21}U_{12} + S)\,C^{-1}{}_{21} & = \mathrm{O} \\ \text{(d)} & L_{21}U_{11}\,C^{-1}{}_{12} + (L_{21}U_{12} + S)\,S^{-1} & = I \end{array} \tag{2.13}$$

The identity matrices I and the null matrices O have the appropriate size. Equation (d) gives an expression for $C^{-1}{}_{12} = -U_{11}{}^{-1}U_{12}\,S^{-1}$, while $C^{-1}{}_{21}$ can be computed from equation $L_{11}{}^{-1}(\mathrm{b}) - L_{21}{}^{-1}(\mathrm{c})$, and use it for comptuing $C^{-1}{}_{11}$. The final expression for C^{-1} is then

$$C^{-1} = \begin{pmatrix} A^{-1} + X\,S^{-1}\,Y & -X\,S^{-1} \\ -S^{-1}\,Y & S^{-1} \end{pmatrix}, \tag{2.14}$$

where X and Y are such that $U_{11}X = U_{12}$ and $Y L_{11} = L_{21}$.

The reason to prefer the block formula (2.14) in place of the (2.9) is the possibility to use level-3 BLAS (basic linear algebra subprograms) libraries [Lawson et al., 1979], providing better performance than the elementwise operations. Notice that the block decomposition (2.10) is rather common, and can be obtained also from any sparse matrix by moving to its end the dense rows and columns that C may present or by applying a reordering to its rows and columns such that a domain decomposition-like structure structure is obtained:

$$C = \left(\begin{array}{cccc|c} A_1 & & & & B_1 \\ & A_2 & & & B_2 \\ & & \ddots & & \vdots \\ & & & A_p & B_p \\ \hline C_1 & C_2 & \cdots & C_p & D \end{array}\right). \tag{2.15}$$

Here, the A_j's are $n_j \times n_j$, D is $n_D \times n_D$, and $\sum_j n_j + n_D = n$; renaming

$$A = \begin{pmatrix} A_1 & & \\ & \ddots & \\ & & A_p \end{pmatrix}, \quad B = \begin{pmatrix} B_1 \\ \vdots \\ B_p \end{pmatrix}, \quad \text{and } E = \begin{pmatrix} C_1 \\ \vdots \\ C_p \end{pmatrix}^{\top},$$

we find again the structure depicted in (2.10). Such reordering can be found for any matrix, as shown, e.g., in [Karypis and Kumar, 1998], and using, for instance, the software METIS/ParMETIS and KaHIP [Karypis and Kumar, 2009; Meyerhenke et al., 2015]. We are discussing additional details about this and the symbolic factorization phase in Part II of this document.

2.5 Exploiting the sparsity of the blocks

Among the applications where the inverse of a matrix must be computed, an expression for the whole inverse is usually not necessary, but only a limited number of entries must be computed, e.g., the diagonal. These applications are heterogeneous in nature, and span from portfolio management [Tang and Saad, 2012] to nanoelectronic device simulation [Kuzmin et al., 2013] and genomic prediction [Verbosio et al., 2017]. Additionally, in all of the mathematical models considered, the original matrix, say C, is sparse or, at least, presents a large sparse block A. For these reasons, we have developed an algorithm aiming at computing the diagonal of C^{-1} exploting both the Schur-complement decomposition and the sparsity of A. First of all, in order to compute the diagonal of the inverse, it is sufficient to compute the entries of the diagonal blocks in (2.14), i.e.,

$$\text{blockDiag}\left(C^{-1}\right) = \begin{pmatrix} A^{-1} + X\,S^{-1}\,Y & \\ & S^{-1} \end{pmatrix}. \tag{2.16}$$

Here, the sparsity of block ${C^{-1}}_{11} = \left(A^{-1} + X\,S^{-1}\,Y\right)$ should be preserved, therefore requiring A^{-1} to be computed as a selected inverse. On the other hand, since the Schur-complement, by definition, is dense, S^{-1} must be stored as a dense matrix unless the product $A^{-1}B$ in (2.12) is calculated explicitly evaluating a selected inverse of A and sparsifying both the computation of S and its inverse, introducing an approximation error.

A sketch of the procedure for evaluating the (2.16) is described in Algorithm 1. The workflow starts from the computation of the Schur-complement (and its inverse). For performance, the linear systems designed for computing X and Y (lines 2 and 3) should be solved using a sparse direct solver able to reuse the LU factorization of A for the solution with many righ-hand sides. In order to save both memory and computing time, an LU factorization of A is computed and stored before addressing the two linear systems, then the stored factors are used to solve both. As an example, PARDISO [Kuzmin et al., 2013; Petra, Schenk and Anitescu, 2014; Petra, Schenk, Lubin and Gärtner, 2014] fits these requirements. On the other hand, since there is no need to reuse the original values of D, this block can be overwritten with the Schur-complement S, first, and its inverse, later, in order to save memory (lines 4 and following). Recalling the requirement to keep the sparsity of block ${C^{-1}}_{11}$, function SELINV in line 7 evaluates a selected inverse of A, computing at least the entries belonging to $\mathscr{N}(A)$. Finally, the selected entries of A^{-1} are updated according to (2.16) (line 9). Here, $X_{(i,\cdot)}$ and $Y_{(\cdot,j)}$ indicate the ith row of X and the jth column of Y, respectively. Notice that the operator diag : $[\mathbb{K}^{m\times m} \ni M \mapsto \mathrm{diag}(M) \in \mathbb{K}^m], m \in \mathbb{N}$, used in line 11 extracts the diagonal of a given matrix as a vector.

Algorithm 1 Sequential Takahashi's method for computing the diagonal of the inverse.

```
1:  procedure BLOCKSELECTEDINVERSION($A$, $B$, $E$, $D$)
2:      Solve for $X$ the system $AX = B$    $(\iff X = A^{-1}B)$
3:      Solve for $Y$ the system $YA = E$    $(\iff Y = EA^{-1})$
4:      $S \leftarrow D - EX$
5:      Compute INV($S$)                        ▷ Compute the inverse of $S$
6:      ${C^{-1}}_{22} \leftarrow S^{-1}$
7:      $A_{inv} \leftarrow$ SELINV($A$)              ▷ Compute the selected inverse of $A$
8:      for $(i,j) \in \mathscr{N}(A)$ do
9:          $\left({C^{-1}}_{11}\right)_{ij} \leftarrow (A_{inv})_{ij} + X_{(i,\cdot)} S^{-1} Y_{(\cdot,j)}$
10:     end for
11:     $d \leftarrow \left(\mathrm{diag}({C^{-1}}_{11}), \mathrm{diag}({C^{-1}}_{22})\right)^\top$    ▷ $d$ contains the diagonal of $C^{-1}$
12:     return $d$
13: end procedure
```

2.6 The selected inversion algorithm

In Algorithm 1, the most expensive computational kernels are the selected inversion of A and the computation of the inverse Schur-complement. These two operations, however, present a set of shared sub-tasks that can be computed simultaneously. First, computing the Schur-complement of A in C and, later, the evaluation of the entries of the inverse, require the solution of two linear systems (lines 2 and 3), both requiring the factorization of A. Second, the selected inversion process in line 7 is as well based on the factorization of A. The structure of the algorithm suggests then to reuse the factorization already computed, reducing the computational time dramatically. This feature is already implemented in different solvers like PARDISO, keeping a copy of the LU fac-

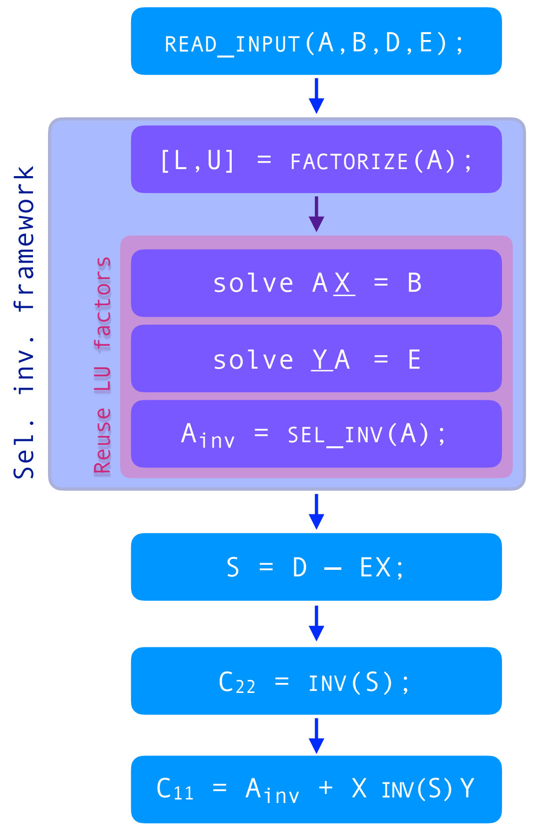

Figure 2.1. Sequential Takahashi's inversion workflow.

tors of A, so that they are computed once and for all, until manually released by the user. The factorization hence is used to first solve the two linear systems, then to compute the selected inverse of A, and finally to evaluate the Schur-complement. We illustrate the complete workflow in Figure 2.1.

Notice that the Schur-complement is a dense block, hence the inverse S^{-1} should not be computed as a selected inverse. The inversion process requires then to use linear algebra routines optimized for dense operations, i.e., LAPACK libraries [Anderson et al., 1999]. In several cases, block D has a size considerably smaller than A, allowing an agile treatment of the Schur-complement without essentially altering the complexity. In a more general setup, however, this is not true and it is very important to design a strategy to tackle the Schur-complement calculation at a reduced computational cost. Looking closer at (2.12), it is possible to decompose the computation of S in subblocks, independent of one another: each entry of S is computed considering different rows and columns of B and E separately. In particular, as depicted in Figure 2.2, provided that A is shared among all the computing entities, each of them can compute a single entry $(S)_{ij}$ by considering only $E_{(i,\cdot)}$ and $A^{-1}B_{(\cdot,j)}$. This property suggests to implement a parallel computation of the Schur-complement, which is the base of the parallel Takahashi's selected inversion we will present in the following chapter.

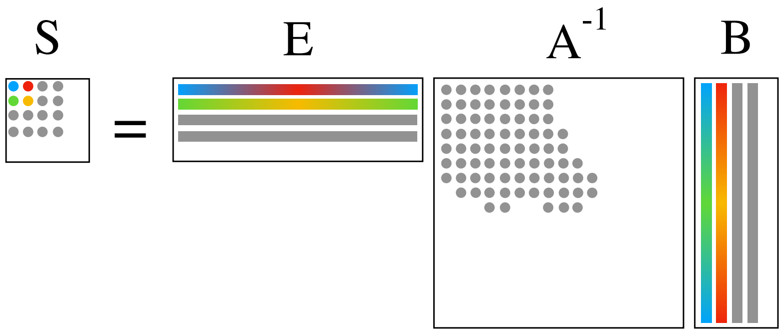

Figure 2.2. Schematic representation of the Schur-complement computation. Element S_{ij} is the scalar product of $E_{(i,\cdot)}$ and $A^{-1}B_{(\cdot,j)}$.

2.7 The parallel Takahashi's method

For application-related reasons described in [De Coninck et al., 2014], we also add the condition that C is symmetric, i.e., both A and D are symmetric and $E = B^{\top}$. The parallel Takahashi method is described in Algorithm 2, and combines the Takahashi's formula (2.9) with the Schur-complement decomposition (2.11) allowing the treatment of large-scale matrices through the distribution of its blocks among a set of processes. The computation of the Schur-complement and the update of the selected inverse are performed in parallel, and the results are gathered on a root process. The algorithm performance is boosted by the usage of optimized linear algebra and communication libraries and the MPI (message passing interface) paradigm [Snir et al., 1998].

2.7.1 Matrix representation and compression

Considering a matrix C, decomposed in blocks as in (2.10), the first core aspect that the parallel Takahashi's algorithm must address is the distribution of such blocks. The distribution is made according to the standard defined by the BLACS (basic linear algebra communication subprograms) libraries, which are the communicaion layer of ScaLAPACK (scalable linear algebra package) and

Algorithm 2 — Parallel Takahashi selected inversion

```
 1: Allocate a $p \times p$ MPI process grid, $\mathcal{G}$
 2: Divide $D$ into $p^2$ blocks, $D_{ij}$      ▷ $1 \le i, j \le p$, of size $\frac{n_D}{p} \times \frac{n_D}{p}$
 3: Divide $B$ into $p$ blocks, $B_i$      ▷ $1 \le i \le p$, of size $n_A \times \frac{n_D}{p}$
 4: parfor $(i, j) \in \mathcal{G}$ do (in parallel)
 5:     SEND$(i, B_i)$      ▷ Send the blocks of $B$ to the respective rows of $\mathcal{G}$
 6:     SEND$(i, j, D_{ij})$  ▷ Send the blocks of $D$ to the respective processes of $\mathcal{G}$
 7:     $D_{ij} \leftarrow D_{ij} - B_i^{\top} A^{-1} B_j$      ▷ Compute the Schur-complement
 8: end parfor
 9: $D \leftarrow D^{-1}$      ▷ on each process, using BLACS
10: $A_{\text{inv}} \leftarrow$ SELINV(A)      ▷ on the root process, using PARDISO
11: parfor $(i, j) \in \mathcal{G}$ do (in parallel)
12:     $Y_j \leftarrow$ SOLVE$(A, B_j)$      ▷ Solve for $Y_j$ the system $AY_j = B_j$
13:     if $i \neq j$ then
14:         $Y_i \leftarrow$ SOLVE$(A^{\top}, B_i)$      ▷ Solve for $Y_i$ the system $A^{\top} Y_i = B_i$
15:     end if
16: end parfor
17: $A_{\text{inv}} \leftarrow A_{\text{inv}} + Y_i^{\top} D Y_j$
18: $d \leftarrow \left(\text{diag}(A_{\text{inv}}), \text{diag}(D)\right)^{\top}$
```

PBLAS (parallel basic linear algebra subprograms) libraries [Blackford et al., 1997; Choi et al., 1996]. For blocks D and B, which are in general dense for application-related reasons, a cyclic distribution scheme is adopted; we analyze the details of the matrix distribution in the following section. Block A, instead, is entirely allocated on the local memory of each process: storing A on every local memory is in general possible since the sparsity of A allows it to be represented in CSR (compressed sparse row) format with a considerable amount of memory saved with respect to the dense version, requiring the allocation of n^2 numbers. Instead of storing all the n^2 entries of the matrix, three vectors are used: one for the nonzeros, one for the column indices of the nonzeros, and one for the indices pointing at the first nonzero of each row. We are presenting more implementation details about the matrix distribution and storage in section 6.1

2.7.2 The parallel Schur-complement

Let us consider p^2 MPI processes arranged in a virtual 2D square grid of size $p \times p$, such that each of them can be identified by the couple (i, j) for some $0 \le i, j \le p-1$. This setup improves the efficiency of operations and communications on a particular distribution scheme we will analyze at a later time. The goal of the parallel Takahashi algorithm is to make the processes operate independently on matrices of size much smaller than C, and at the end of the computation gather the results on the master process. In order to proceed accordingly to the parallel Schur-complement computation presented in Figure 2.2, block D will be decomposed both row-wise and column-wise into as-square-as-possible patches, while B will be decomposed column-wise into vertical strips. The details about the memory distribution are presented in Section 6.1.

Let us now analyze the details of the parallel algorithm (Algorithm 2). The block subdivision of matrices D and B (steps 2 and 3) is crucial. For the sake of simplicity, we assume at the moment that both n_A and n_D are multiples of p, however we will analyze a general case in section 6.1. The loop in lines 4 to 8 is the core of the parallel implementation: each process, identified by its coordinates in the grid, (i, j), can compute its own piece of the Schur-complement D_{ij} independently from the other processes. Recall that, as an exception to the usual notation, D_{ij} does not indicate a single entry of D, but a block of entries. Analogously, by B_j and $B_i^\top$ in line 7 we indicate the blocks composed by the columns of B and $B^\top$ and, in the same fashion, blocks Y_j and $Y_i^\top$ are the stripes composing the solution of the system $AY = B$ and $A^\top Y = B$. Notice that Y_j and $Y_i^\top$ are computed by p processes in parallel (line 11); the reason why only p processes, and not p^2, are involved in this computation lies in the fact that the distribution of B is done according to a 1D cyclic distribution (and not 2D). Finally, in the second to last line the entries of A^{-1} are updated according to the rule defined in the sequential case (Algorithm 1, line 9). Since A^{-1} is computed as a selected inverse, only the nonzero entries are actually updated, e.g., the diagonal.

In chapter 7 we are going to present the performance of the parallel Takahashi's approach in the computation of the maximum-likelihood estimators in a genomic prediction framework.

Chapter 3

Stochastic estimation for the diagonal of the inverse

In this chapter we discuss a technique to estimate the diagonal of the inverse using a stochastic estimator. Such technique can be used to compute the selected entries of the inverse up to some tolerance greater than machine precision. Applying this method can be useful to calculate the first decimal digits of the diagonal entries in an agile way, avoiding any fill-in and using optimized Krylov-subspace methods. We first recall how to design the estimator and its properties, then we describe an iterative algorithm that estimates the diagonal of the inverse by generating random samples. An application of the algorithm follows in chapter 8.

3.1 Unbiased estimators for diagonal and trace

In order to introduce the estimator, it is useful to recall some notations and definitions from the probability theory.

Definition 2 (Probability and statistics notations). *We introduce the following simplified definitions and notations that are going to be used throughout the whole document.*

(i) We say that a variable x is distributed as another variable y (and we indicate it as $x \sim y$) if the domains of x and y are the same and if $\mathbb{P}(x = \omega) = \mathbb{P}(y = \omega)$ for any valid choice of ω in the domain.

(ii) We say that a variable x follows the probability distribution $\mathscr{P}$ ($x \sim \mathscr{P}$) if the probability distribution of x coincides with $\mathscr{P}$.

(iii) *We say that $p > 1$ variables $x_1, \dots, x_p$ are independent and identically distributed (and we indicate it with "i.i.d.") if $x_i \sim x_j$ and x_i is independent from x_j ($x_i \perp\!\!\!\perp x_j$) for all $1 \le i, j \le p$ and $i \neq j$.*

3.1.1 Estimator for the trace of a given matrix

Let now A be an $n \times n$ (sparse) matrix and e_k for $k = 1, \dots, n$ the columns of the identity matrix of the same order, I. It is straightforward to see that it is possible to extract the vector containing the diagonal elements of A and, consequently, the trace of A, according to the formulas

$$\begin{aligned} \operatorname{diag}(A) &= \sum_{k=1}^{n} \left(e_k^{\top} A e_k\right) e_k \quad \text{and} \\ \operatorname{tr}(A) &= \sum_{k=1}^{n} e_k^{\top} A e_k \,. \end{aligned} \tag{3.1}$$

In 1990, Hutchinson developed an estimator for the trace of A of the form $\tau = u^{\top} A u$, where the random vector u has length n, zero mean and fixed variance.

Theorem 3 (Estimator for the trace [Hutchinson, 1990]). *Given a square, $n \times n$ matrix A and $\{v_i\}_{i=1}^{n}$ i.i.d., $u = (v_1, \dots, v_n)$, such that $\mathbb{E}(v_i) = 0$ and $\mathbb{V}(v_i) = \sigma^2$ for any i, let us consider the scalar $\tau = u^{\top} A u$. Then, such scalar is an estimator for* $\operatorname{tr}(A)$ *and*

$$\mathbb{E}(\tau) = \sigma^2 \operatorname{tr}(A) \,.$$

Proof. By rewriting the estimator, we can see that

$$\begin{aligned} \tau = \sum_{i,j=1}^{n} u_i a_{ij} u_j &= \sum_{i,j=1}^{n} a_{ij} u_j u_i \\ &= \sum_{i=1}^{n} a_{ii} u_i^2 + \sum_{i<j} a_{ij} u_i u_j + \sum_{i>j} a_{ij} u_i u_j \\ &= \sum_{i=1}^{n} a_{ii} u_i^2 + \frac{1}{2} \sum_{i \neq j} \left(a_{ij} u_i u_j + a_{ji} u_j u_i\right) \\ &= \sum_{i=1}^{n} a_{ii} u_i^2 + \frac{1}{2} \sum_{i \neq j} (a_{ij} + a_{ji}) u_i u_j \,. \end{aligned} \tag{3.2}$$

Hence the expected value is

$$\begin{aligned}
\mathbb{E}(\tau) &= \mathbb{E}\left(\sum_{i=1}^{n} a_{ii}\, u_i^2 + \frac{1}{2}\sum_{i\neq j}(a_{ij}+a_{ji})\, u_i u_j\right) \\
&= \mathbb{E}\left(\sum_{i=1}^{n} a_{ii}\, u_i^2 + \sum_{1\le i<j\le n}(a_{ij}+a_{ji})\, u_i u_j\right) \\
&= \sum_{i=1}^{n} a_{ii}\, \underbrace{\mathbb{E}(u_i^2)}_{=\mathbb{V}(u_i)} + \sum_{1\le i<j\le n}(a_{ij}+a_{ji})\, \mathbb{E}(u_i u_j) \\
&= \sum_{i=1}^{n} a_{ii}\, \underbrace{\mathbb{V}(u_i)}_{=\sigma^2\,\forall i} + \sum_{1\le i<j\le n}(a_{ij}+a_{ji})\, \underbrace{\mathbb{E}(u_i u_j)}_{=0 \text{ since } i\neq j} \\
&= \sigma^2\, \mathrm{tr}(A)\,.
\end{aligned} \tag{3.3}$$

□

From here, it is clear how a clever choice for the random samples could cancel the bias. It is easily proven that the Hutchinson's estimator for the trace of A, τ_H, defined as follows is unbiased and has minimum variance:

$$\begin{aligned}
\tau_H &= u^{\top} A u \qquad \text{such that} \\
&\quad u^{\top} = (v_1,\dots,v_n), \quad \{v_i\}_i \perp\!\!\!\perp \quad \text{and} \\
&\quad \mathbb{P}(v_i=+1) = \mathbb{P}(v_i=-1) = \frac{1}{2} \quad \forall i\,.
\end{aligned} \tag{3.4}$$

Theorem 4 (Unbiased estimator for the trace). *The Hutchinson's estimator* τ_H *defined in* (3.4) *is an unbiased estimator for* tr(A).

Proof. This follows immediately from Theorem 3 and the fact that for the Hutchinson's estimator $\mathbb{V}(v_i) = \mathbb{E}({v_i}^2) = 1$ for all i. □

Let us now focus on the variance of the biased estimator τ. We first observe that it can be seen as the quadratic form associated to matrix A applied to vector u. Calling

$$\mathcal{Q}_A : [\mathbb{R}^n \ni x \mapsto x^{\top} A x \in \mathbb{R}]\,,$$

we can write $\tau = \mathcal{Q}_A(u)$. The following result provides an expression for the variance of a random quadratic form associated with A.

Theorem 5 (Variance of the trace estimator)**.** *Given a square, $n \times n$ matrix $A = (a_{ij})_{ij}$ and $u \sim (v_1, \dots, v_n)$, $\{u_i\}_i$ i.i.d., $\mu_k := \mathbb{E}({v_1}^k) < \infty$ for $k = 1, 2, 3, 4$, and $\mathcal{Q}_A$ the quadratic form associated to A, define $\tau = \mathcal{Q}_A(u) = u^\top A u$. Then,*

$$\mathbb{V}(\tau) = (\mu_4 - 3\mu_2^2)\sum_i a_{ii}^2 + (\mu_2^2 - 1)\,\mathrm{tr}(A)^2 + 2\mu_2^2\,\mathrm{tr}(A^2)\,. \tag{3.5}$$

Proof. We first observe that the quadratic form associated to matrix A is equivalent to the quadratic form associated to $\frac{1}{2}(A + A^\top)$, so we can assume A to be symmetric. Now, in order to compute $\mathbb{V}(\tau)$, it is necessary to compute $\mathbb{E}(\tau^2)$ first:

$$\mathbb{E}(\tau^2) = \mathbb{E}\Big(\sum_{i,j,k,l=1}^{n} a_{ij}a_{kl}u_iu_ju_ku_l\Big) = \sum_{i,j,k,l=1}^{n} a_{ij}a_{kl}\mathbb{E}[u_iu_ju_ku_l]\,. \tag{3.6}$$

Given the independence of the v_i's and the zero mean, we can compute the expected value in formula (3.6) quite easily as

$$\mathbb{E}[u_iu_ju_ku_l] = \begin{cases} \mu_4 & \text{if} \quad i = j = k = l\,, \\ \mu_2^2 & \text{if}\, i = j \neq k = l \,\vee\, i = k \neq j = l \,\vee\, i = l \neq k = j\,, \\ 0 & \text{otherwise} \end{cases}. \tag{3.7}$$

From this, we rewrite (3.6) as

$$\begin{aligned} \mathbb{E}(\tau^2) &= \mu_4 \sum_i a_{ii}^2 + \mu_2^2 \sum_{i \neq k} a_{ii}a_{kk} + \mu_2^2 \sum_{i \neq j} a_{ij}a_{ji} + \mu_2^2 \sum_{l \neq k} a_{lk}a_{kl} \\ &= \mu_4 \sum_i a_{ii}^2 + \mu_2^2 \Big[\sum_{i \neq k} a_{ii}a_{kk} + 2\sum_{i \neq j} a_{ij}a_{ji}\Big]. \end{aligned} \tag{3.8}$$

Now, it is clear how the first term in the last square bracket can be rewritten as

$$\begin{aligned} \sum_{i \neq k} a_{ii}a_{kk} &= \sum_{i,k} a_{ii}a_{kk} - \sum_i a_{ii}a_{ii} \\ &= \mathrm{tr}(A)^2 - \sum_i a_{ii}^2\,, \end{aligned}$$

while the second term is

$$\begin{aligned} \sum_{i \neq j} a_{ij}a_{ji} &= \sum_{i,j} a_{ij}a_{ji} - \sum_i a_{ii}a_{ii} \\ &= \sum_i \sum_j a_{ij}a_{ji} - \sum_i a_{ii}^2 \\ &= \sum_i (B^2)_{ii} - \sum_i a_{ii}^2 \\ &= \mathrm{tr}(A^2) - \sum_i a_{ii}^2\,. \end{aligned}$$

Finally, replacing these two quantities in (3.8) and subtracting $\mathbb{E}^2(\tau) = \mathrm{tr}(A)^2$, we obtain (3.5). $\square$

From the previous theorem, considering now the υ_i's such that $\mu_2 = \sigma^2$ and $\mu_4 = \sigma^4$, we can rewrite (3.5) as

$$\mathbb{V}(\tau) = (\sigma^4 - 1)\,\mathrm{tr}(A)^2 + 2\sigma^4\Big(\mathrm{tr}(A^2) - \sum_i a_{ii}^2\Big). \tag{3.9}$$

Theorem 6 (Variance of the Hutchinson's estimator). *The Hutchinson's estimator's variance is*

$$\mathbb{V}(\tau_H) = 2\,\|A - \mathrm{Diag}(A)\|_F^2 .$$

where Diag(A) *is the diagonal matrix having the diagonal of A as entries.*

Proof. When the $\{\upsilon_i\}_i$ are chosen according to the definition (3.4), we have $\sigma^2 = 1$ and from (3.9) the variance becomes

$$\mathbb{V}(\tau_H) = 2\ \mathrm{tr}(A^2) - 2\sum_i a_{ii}^2 = 2\,\|A - \mathrm{Diag}(A)\|_F^2 , \tag{3.10}$$

minimizing (3.9). $\square$

Given the fact that it is unbiased and has minimum variance, the estimator can be evaluated as the sample mean of a sufficiently large set of samples designed according to the definition of u. Considering s i.i.d. vectors $v_k \sim u$ for $k = 1, \ldots, s$,

$$\mathrm{tr}(A) \approx \frac{1}{s}\sum_{k=1}^{s} {v_k}^\top A v_k . \tag{3.11}$$

This property is described in [Bekas et al., 2009; Tang and Saad, 2012]. For the arbitrary choice of A, (3.11) applies to any matrix, even to its inverse A^{-1}; hence, in the following sections, we are going to use it for the computation of both the trace and the diagonal of an inverse matrix. We will also discuss a strategy to reduce the computational cost of the inversion process, choosing an appropriate evaluation strategy for the samples.

3.1.2 Unbiased estimator for the trace of the inverse

The evaluation of the entries of the diagonal (3.1) can be specialized for the computation of the diagonal of the inverse $\mathrm{diag}(A^{-1}) = (\delta_1, \ldots, \delta_N)$. If A^{-1} is known, one simply needs to apply the (3.1), giving

$$\delta_k = \sum_{k=1}^{n} {e_k}^\top A^{-1} e_k .$$

This formula, however, would obviously be as expensive as the dense inversion process. We observe that the computation of the whole inverse can be avoided by solving a linear system, since ${e_k}^\top A^{-1} e_k$ can be computed equivalently as ${e_k}^\top y_k$, and $y_k = A^{-1} e_k \iff Ay_k = e_k$. The strategy to be adopted becomes then

$$\delta_k = \sum_{k=1}^{n} {e_k}^\top y_k \quad \text{where} \quad Ay_k = e_k \,.$$

Here, the linear systems $Ay_k = e_k$ are computed in several ways. A straightforward strategy would be calculating the solution of the n linear systems using the LU factorization of A, but this would give a computational complexity equivalent to the dense inversion. In order to design a selected inversion approach based on this idea, it becomes therefore necessary to reduce the computational burden of the problem; in order to do so, we consider two lines of action:

1. The replacement of the direct method for the computation of the y_k's with an iterative method, e.g., a Krylov-subspace iteration, so that both the execution time and the memory requirements are reduced. The computing time can be sensibly cut down by choosing an appropriate tolerance, and trading some of the accuracy for a quicker solution. On the other hand, independently from the tolerance, the memory remains modest, since the original matrix is never modified and no fill-in entries are generated. While the choice of the tolerance for the method is in principle a complex issue and may depend on the structure of the linear system, it could also possible to precondition the system through, for instance, an incomplete LU factorization [Bollhöfer et al., 2019] in order to reduce the number of iterations.

2. The reduction of the number of linear systems to be solved from n to a sensibly smaller number, say, $s \ll n$, together with formulating a process to extract "enough" information from the estimator. The latter is a delicate point and is strictly related to the design of the stochastic estimator τ_H.

Considering the first idea, one could approximate the diagonal of the inverse of A as

$$\begin{aligned} \operatorname{diag}(A^{-1}) &\approx \sum_{k=1}^{n} \left({e_k}^\top y_k\right) e_k \,, \\ \text{where} \quad y_k &\overset{\text{Krylov}}{\approx} A^{-1} e_k \end{aligned} \tag{3.12}$$

and the methods used to compute the y_k's can be heterogeneous, e.g. conjugate gradient [Golub and Van Loan, 2013, Chapter 11] for symmetric positive definite systems or others (MINRES [Paige and Saunders, 1975], GMRES [Saad, 2003]) for more general matrices. In addition, we can combine (3.12) with the second idea and reduce the number of systems to be solved. Considering method (3.11) to estimate the diagonal, in fact, one may reduce the number of right-hand-sides to some value $s \ll n$, such that the trace of A^{-1} can be estimated correctly, provided that the random samples are generated accordingly. The final expression for the estimator for the trace becomes then

$$\begin{aligned} \mathrm{tr}(A^{-1}) &\approx \frac{1}{s}\sum_{k=1}^{s} {v_k}^\top (A^{-1}v_k) \\ &= \frac{1}{s}\sum_{k=1}^{s} {v_k}^\top x_k\,, \quad \text{where}\, Ax_k \overset{\text{Krylov}}{\approx} v_k\,. \end{aligned} \tag{3.13}$$

3.2 Unbiased estimator for the diagonal of the inverse

From here, we extrapolate a formula for the diagonal of the inverse $\mathrm{diag}(A^{-1})$ by replacing the inner products ${v_k}^\top x_k$ with Hadamard (elementwise) operations

$$\begin{aligned} &\odot : \big((\alpha_1,\dots,\alpha_n),(\beta_1,\dots,\beta_n)\big) \mapsto \big(\alpha_1\beta_1,\dots,\alpha_n\beta_n\big) \text{ and} \\ &\oslash : \big((\alpha_1,\dots,\alpha_n),(\beta_1,\dots,\beta_n)\big) \mapsto \big(\frac{\alpha_1}{\beta_1},\dots,\frac{\alpha_n}{\beta_n}\big), \quad \alpha_k, 0 \neq \beta_k \in \mathbb{K}, \ \forall k. \end{aligned}$$

As discussed in [Bekas et al., 2009, equation (3)] and [Bekas et al., 2007, section 2.1], an estimator $\widehat{d_A}$ for the diagonal of A is computed as

$$\widehat{d_A} = \left(\sum_{k=1}^{s} v_k \odot Av_k\right) \oslash \left(\sum_{k=1}^{s} v_k \odot v_k\right). \tag{3.14}$$

Combining this with the estimator for the trace (3.13), we can replace the vectors Av_k's with the x_k's previously defined and design an estimator for the diagonal of the inverse, $\widehat{d}$, in the form

$$\widehat{d} = \left(\sum_{k=1}^{s} v_k \odot x_k\right) \oslash \left(\sum_{k=1}^{s} v_k \odot v_k\right). \tag{3.15}$$

Looking closer at the quantity in the square brackets, one can see that it can be interpreted as an iterative process. Starting from a null vector as initial guess ($k = 0$), at every iteration for $k > 1$, the previous estimation for $\mathrm{diag}(A^{-1})$ is

Algorithm 3 . Stochastic estimator for the diagonal of the inverse of an invertible matrix.

```
Input: matrix $A$, number of samples $s$, accuracy $\epsilon$
Output: approximate diagonal of the inverse $d$
1: $q \leftarrow (0,\dots,0)$
2: $r \leftarrow (0,\dots,0)$
3: for $k = 1,\dots,s$ do
4:     $v_k \leftarrow$RAND_VECTOR(n, 0, 1)  ▷ generate random vector $v_k$ (mean 0 and variance 1)
5:     $x_k \leftarrow$KRYLOV_SUBSPACE($A$, $v_k$, $\epsilon$)  ▷ solve the linear system $Ax_k = v_k$ up to accuracy $\epsilon$
6:     $q \leftarrow q + v_k \odot x_k$
7:     $r \leftarrow r + v_k \odot v_k$
8: end for
9: $d \leftarrow q \oslash r$
```

summed with the contribution given by the vector x_k, i.e., the solution of the linear system $Ax_k = v_k$ for a new random sample v_k. The contribution added is the elementwise product of the solution and the random sample, weighted according to the entries of the sample itself. Starting from two initial values $q^{(0)} = (0,\dots,0)$ and $r^{(0)} = (0,\dots,0)$, then, we can write

$$\begin{cases} q^{(k)} = q^{(k-1)} + v_k \odot x_k\,, & 1 \le k \le s \\ r^{(k)} = r^{(k-1)} + v_k \odot v_k\,, & 1 \le k \le s \\ \quad \widehat{d} = q^{(s)} \oslash r^{(s)} \end{cases} . \tag{3.16}$$

The essential process is described in Algorithm 3. First, two null n-vectors, q and r, are allocated. Then, at each iteration, a random vector with zero mean and unitary variance, v_k, is generated, and the linear system $Ax_k = v_k$ is solved using an iterative Krylov-subspace method (lines 4 and 5). The iteration is completed (line 6) updating each entry of q by adding to its value the corresponding entry of the solution of the linear system weighted by the corresponding entry of v_k: $q^i = q^i + x_k^i v_k^i$, $i = 1,\dots,n$. Notice that the update of r (line 7) simply consists in adding to each entry of r the square of the corresponding entry of v_k: $r^i = r^i + (v_k^i)^2$, $i = 1,\dots,n$. We obviously assume that all the operations are computable, hence that none of the entries of v_k are ever zero during the whole process. Given the independence of the samples, in case of a null entry, we can simply discard the sample and proceed with the generation of a new one.

In case that the Hutchinson's estimator is used, the v_k's are generated according to definition (3.4). The expressions for computing q and r then is simplified and becomes

$$\begin{cases} q^i = q^i + \text{sign}(v_k^i)\, x_k^i \\ r^i = r^i + 1 \end{cases},$$

giving $r = (s, s, \dots, s)$ at the end of the s iterations. This means that each entry of d, computed as $d^i = q^i/s$, will contain the sample mean of the weighted values of x_k, where the weights are either $+1$ or -1 with probability $\frac{1}{2}$.

In the following sections we describe a matrix-free version of the algorithm for the computation of the diagonal of a sparse matrix that can be described as a stencil. The Krylov subspace method will be formulated keeping the matrix implicit, reducing the memory requirements and allowing the parallelization of the iterative method.

3.3 Stencil-based formulation

In this section, we extend the stochastic estimator introduced before and apply it to the problem of estimating the diagonal of the inverse of a sparse matrix. We design a framework for a stencil-based, matrix-free approach, where we reformulate Algorithm 3 keeping the original matrix implicit, and computing the solutions of the computational kernel, i.e., the Krylov-subspace iterations, using stencil operations (for more details about stencils, refer to, e.g., [Saad, 2003, Section 2.2 and Section 10.3]).

3.3.1 Stencil-based matrix and vector operations

Assuming that the structure of the matrix whose diagonal must be estimated can be described as a stencil, we can translate the stochastic estimation algorithm introduced in the previous sections. The pivotal assumption behind such reformulation is the usage of a Krylov-subspace iteration method to solve the linear systems in (3.12). Krylov-subspace methods are in fact based solely on four types of operations that are vector sum, scalar-vector product, (vector-vector) dot product, and matrix-vector multiplication. The first three are elementary operations that can be performed directly on any data structure, while the last, matrix-vector multiplications, can be effortlessly translated into elementary stencil operations keeping the matrix actually implicit and never build it nor store it. This key property for the matrix-vector operations can be ex-

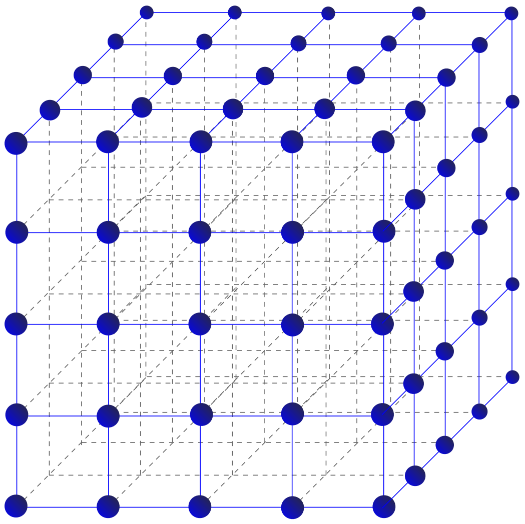

Figure 3.1. Discrete 3-dimensional grid with $5 \times 5 \times 5$ points.

ploited only if the nonzeros of the matrix describe a stencil-structured matrix.

Definition 3 (Stencil-structured matrix). *We say that a matrix is stencil-structured if there exist a d-dimensional stencil ($d > 1$) and a function called "reshape function", such that (i) the reshape function is invertible and maps any vector to a d-dimensional discrete domain, e.g., a regular grid, and (ii) any matrix-vector product can be expressed by applying the stencil to the vector reshaped as a discrete domain through the reshape function, and finally reshaping the resulting domain back in vector form.*

We consider a three-dimensional, $n \times n \times n$ regular discrete domain Ω with $N = n^3$ nodes (Figure 3.1), an $N \times N$ sparse matrix A, and a vector $b \in \mathbb{K}^N$. We indicate the nodes of Ω with p_{ijk} for $1 \leq i, j, k \leq n$, and with α_{ij} and β_j for $1 \leq i, j \leq N$ the entries of A and b, respectively. The structure of Ω can be intended to be the domain of a discrete scalar function f of three variables by describing the image of Ω as a grid of $n \times n \times n$ values

$$\mathscr{F} = \{f_{ijk} = f(p_{ijk}) \in \mathbb{K},\ 1 \leq i, j, k \leq n\}.$$

We first observe that the nodes in Ω can be labeled using a sequential index in

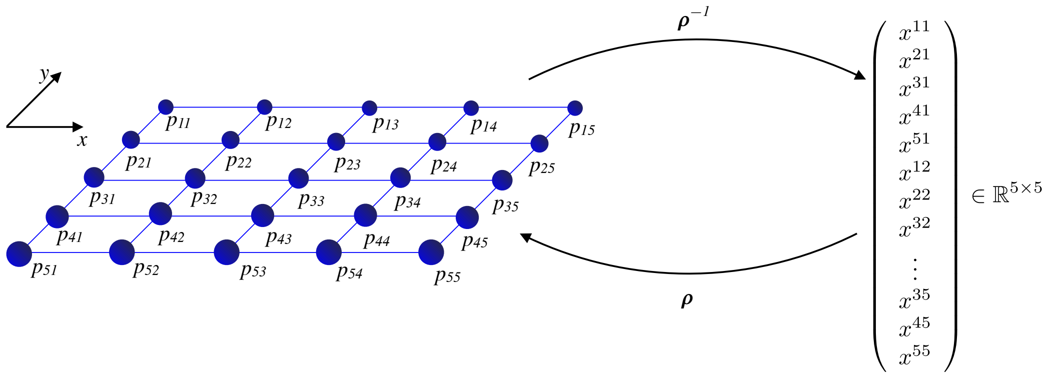
y
x
p_{11} p_{12} p_{13} p_{14} p_{15}
p_{21} p_{22} p_{23} p_{24} p_{25}
p_{31} p_{32} p_{33} p_{34} p_{35}
p_{41} p_{42} p_{43} p_{44} p_{45}
p_{51} p_{52} p_{53} p_{54} p_{55}
ρ^{-1}
ρ
x^{11} x^{21} x^{31} x^{41} x^{51} x^{12} x^{22} x^{32} ⋮ x^{35} x^{45} x^{55}
$\in \mathbb{R}^{5\times 5}$

Figure 3.2. An example of the reshape operator (3.18), describing a bijection between the points of an $n \times n$ grid and the vectors of $\mathbb{R}^{n^2}$.

a column-wise fashion, i.e., we can map it to an $N \times 1$ vector

$$\Omega \mapsto \underbrace{\begin{pmatrix} x^1 \\ \vdots \\ x^N \end{pmatrix}}_{\in \mathbb{K}^N} \text{ s.t. } \begin{cases} p_{111} \mapsto x^1 & p_{121} \mapsto x^{n+1} & & p_{1nn} \mapsto x^{N-n+1} \\ p_{211} \mapsto x^2 & p_{221} \mapsto x^{n+2} & & p_{2nn} \mapsto x^{N-n+2} \\ \vdots & \vdots & \cdots & \vdots \\ p_{n11} \mapsto x^n & p_{n21} \mapsto x^{2n} & & p_{nnn} \mapsto x^N \end{cases} . \qquad (3.17)$$

Equivalently, it is possible to relabel the elements belonging to any other grid $\mathscr{G}$ as a vector $g = (g^1, \ldots, g^N)^\top$ using the same technique. We define a reshape operator as the inverse of (3.17), mapping the vectors of $\mathbb{K}^N$ onto any grid

$$\begin{aligned} &\rho : \mathbb{K}^N \longrightarrow \mathscr{G}, \; s.t. \\ &\quad \text{for any } v = (\mu_1, \ldots, \mu_N)^\top \in \mathbb{K}^N \text{ and } 1 \le k \le N, \\ &\quad \rho(\mu_k) = g^p \quad \text{for some } p. \end{aligned} \qquad (3.18)$$

Notice that operator (3.17) is trivially invertible (we imply an obvious bijection between any grid $\mathscr{G}$ and the domain Ω). See a small two-dimensional example in Figure 3.2. When matrix A is stencil-structured, it is possible to describe its nonzero pattern via a stencil function,

$$f_A : \mathscr{G} \longrightarrow \mathscr{F}, \qquad (3.19)$$

where both $\mathscr{G}$ and $\mathscr{F}$ are grids of scalars in $\mathbb{K}$. The nonzeros in the jth row of A describe the coefficients of the nodes involved in computing the value of the stencil at node g^j $(1 \le j \le N)$, i.e., if the kth entry in such row a_{jk} is nonzero,

Algorithm 4 . Stencil-based, matrix-free stochastic estimator for the diagonal of the inverse.

Input: stencil function f_A, number of samples s, accuracy ϵ
Output: approximate diagonal of the inverse d

```
 1: $\mathscr{Q}$ ←ZERO_GRID(n,n)
 2: $\mathscr{R}$ ←ZERO_GRID(n,n)
 3: for $k = 1, \dots, s$ do
 4:     $\mathscr{V}_k$ ←RAND_GRID_PLUS_MINUS($n$, $n$)    ▷ generate random $n \times n$ grid $\mathscr{V}_k$
 5:     $\mathscr{X}_k$ ←KRYLOV_STENCIL($f_A$, $\mathscr{V}_k$, $\epsilon$)    ▷ solve the stencil equation $f_A(\mathscr{X}_k) = \mathscr{V}_k$ up to $\epsilon$
 6:     $\mathscr{Q}$ ← $\mathscr{Q}$+ HADAMARD_PROD($\mathscr{X}_k$, $\mathscr{V}_k$)    ▷ extend vector operation $q \leftarrow q + x_k \odot v_k$
 7:     $\mathscr{R}$ ← $\mathscr{R}$+ HADAMARD_PROD($\mathscr{V}_k$, $\mathscr{V}_k$)    ▷ extend vector operation $r \leftarrow r + v_k \odot v_k$
 8: end for
 9: $\mathscr{D}$ ← HADAMARD_RATIO($\mathscr{Q}$, $\mathscr{R}$)    ▷ extend vector operation $d \leftarrow q \oslash r$
10: $d$ ←RESHAPE($\mathscr{D}$, $n$, 1)
```

then the value at node g^k is needed in order to apply the stencil at g^j and its coefficient in the stencil formula is a_{jk}. The product of A with a vector v, then, is computed as

$$y = Av \quad \Longleftrightarrow \quad y = \left(\rho^{-1} \circ f_A \circ \rho\right)(v). \tag{3.20}$$

Let us fix $n > 0$, $N = n^3$, and $1 \leq i \leq N$. For any vector $x \in \mathbb{K}^N$ the i-th entry of the product $y = Ax$ is obviously $y_i = A_{(i,\cdot)}x$. Using the reshape operator as suggested in (3.20), calling $\mathscr{X} = \rho(x)$ the reshaped vector, one can compute y_i by simply applying the stencil described by f_A to the point identified by (j, k, l) in $\mathscr{X}$ (for some $1 \leq j, k, l \leq n$) and mapped to x_i by ρ^{-1}, i.e.,

$$y_i = \left[f_A\big(\rho(x)\big)\right]_{jkl}.$$

Given the stencil function and the reshape operator, the design of stencil-based Krylov-subspace method solves the linear system $f_A(\mathscr{X}) = \mathscr{B}$ keeping both the right-hand side and the solution shaped as grids, once that matrix-vector products are translated into stencil operations. From here, it is possible to reformulate Algorithm 3 in a stencil form, where the linear systems are solved by replacing the standard Krylov-subspace method with its stencil formulation.

Algorithm 5 . Stencil-based, matrix-free Conjugate Gradient method for a generic stencil function f_A and a regular discrete domain $\mathscr{B}$.

```
1: function CG_STENCIL($f_A$, $\mathscr{B}$, tol, maxiter)
2:     $\mathscr{X} \leftarrow$ ZERO_GRID(n,n)                          ▷ initialize solution
3:     $\mathscr{M} \leftarrow$ FILL_GRID($\delta$, $n$, $n$)        ▷ fill grid $\mathscr{M}$ with diagonal entry of $A$, $\delta$
4:     $\mathscr{R} \leftarrow \mathscr{B} - f_A(\mathscr{X})$                          ▷ compute residual
5:     $\rho \leftarrow 0$
6:     for k = 1, ..., maxiter do
7:         $\mathscr{S} \leftarrow f_A(\mathscr{P})$
8:         $\alpha \leftarrow \rho /$ DOT($\mathscr{S}$,$\mathscr{P}$)
9:                                 ▷ DOT sums all the entries of HADAMARD_PROD(·,·)
10:        $\mathscr{X} \leftarrow \mathscr{X} + \alpha\mathscr{P}$
11:        $\mathscr{R} \leftarrow \mathscr{R} - \alpha\mathscr{S}$
12:        $\mathscr{U} \leftarrow$ HADAMARD_RATIO($\mathscr{S}$, $\mathscr{M}$)
13:                                        ▷ apply diagonal preconditioner $\mathscr{M}$
14:        $\rho_{\text{new}} \leftarrow$ DOT($\mathscr{R}$,$\mathscr{U}$)
15:        $\beta \leftarrow \rho_{\text{new}}/\rho$
16:        $\mathscr{P} \leftarrow \mathscr{U} + \beta\mathscr{P}$
17:        if DOT($\mathscr{R}$,$\mathscr{R}$)/DOT($\mathscr{B}$,$\mathscr{B}$) < tol*tol then
18:                                              ▷ use DOT to compute $\|\cdot\|^2$
19:            break
20:        end if
21:    end for
22:    if $k$ < maxit then
23:        return $\mathscr{X}$
24:    else
25:        Error, convergence not achieved.
26:    end if
27: end function
```

A pseudocode is described in Algorithm 4. Supposing that in line 5 the conjugate gradient is used [Saad, 2003, chapter 6], inside this algorithm, at each iteration, there is the need to compute two matrix-vector products that must be replaced with an implicit matrix formula. As we mentioned before, the remaining elementary operations to be computed in the Conjugate gradient can be performed as elementwise operations either on the grids or on the vectors, independently. For instance, the dot product of two vectors shaped as grids $\mathscr{V}$ and $\mathscr{W}$, indicated as DOT($\mathscr{V}$,$\mathscr{W}$), is computed as the sum of the entries of the Hadamard product

$\rho^{-1}(\mathscr{V}) \odot \rho^{-1}(\mathscr{W})$. The point-wise product and ratio, instead, are indicated as HADAMARD_PROD and HADAMARD_RATIO, respectively. An illustrative version of the Conjugate Gradient in stencil form is presented in Algorithm 5, where a simple diagonal preconditioner (Jacobi preconditioner) is included. Such preconditioner is preferable since it does not change the nonzero pattern of the matrix, preserving its stencil-structure. Additionally, applying the diagonal preconditioner involves only the scaling of the stencil coefficients, which translates into very simple changes in the stencil. Function KRYLOV_SUBSPACE in Algorithm 4, line 5 can identify also other methods, such as biconjugate gradient or GMRES [Saad, 2003], to be selected in case of non-positive definite matrices or non-symmetric ones.

In Part III of this document, we are going to present an application for the matrix-free, stencil-based stochastic estimator just designed, and create a framework for nanoelectronic device simulation, where the design of nanodevices requires the computation of the diagonal of the inverse of large, sparse, stencil-structured, symmetric, positive definite matrices.

Part II

High-performance computing selected inversion frameworks

Chapter 4

Selected inversion background

In this chapter we give an overview to some of the existing selected inversion methods and describe briefly their main characteristics and application capabilities. The field of sparse direct solvers is heterogeneous, however most of the techniques involve graph partitioning strategies, supernode-based factorization, and elimination tree parallelism. The following sections offer an overview of PARDISO, PSelInv, MUMPS, FIND, and some of their applications.

4.1 PARDISO—Parallel direct solver

PARDISO is a high-performance, memory efficient, robust, and OpenMP-scalable solver for sparse linear systems and selected inversion [Schenk et al., 2001; Schenk and Gärtner, 2004; Schenk and Gärtner, 2002]. The approach implemented by PARDISO to solve a sparse linear system is based on, first, the computation of fill-in reducing permutation for the coefficient matrix and, second, the factorization of the permuted matrix. The reordering strategy applied is either the minimum degree algorithm [Amestoy et al., 1996] or nested dissection [Karypis and Kumar, 1998, 2009]. In order to exploit the computational efficiency of the Level-3 BLAS libraries [Lawson et al., 1979], PARDISO and many other modern solvers are based on the supernode technology. Supernodes are groups of adjacent columns of a matrix sharing the same structure in the L factor and presenting a dense triangular block right below the diagonal; any exchange of lines belonging to the same supernode does not affect the fill-in, making the supernode mechanism widely used for designing pivoting strategies. The supernode dependencies are described through a graph and an elimination spanning tree is built in order to allow the parallelization of the factorization phase and, consequently, the forward/backward substitution. The elimination tree is created through a depth-first search: starting from a root supernode $\textcircled{n}$, at every step the algorithm selects the node among the neighbors of $\textcircled{n}$ who are still unvisited and having the biggest index. When all the neighbors of a node $\textcircled{k}$ are visited or no neighbor of $\textcircled{k}$ has larger index, then $\textcircled{k}$ is a leaf, and the algorithm tracks back the last node having unvisited neighbors. For more on this, refer to, e.g., [Bollhöfer and Schenk, 2006]. In case the computation of the LU factors requires pivoting, either a standard diagonal pivoting or a 2×2 Bunch-Kaufman pivoting [Bunch and Kaufman, 1975] is used. The latter, available only for symmetric matrices, is based on the comparison between the magnitude of the diagonal entries of the matrix and the off-diagonal ones after each step of the Gaussian elimination. Depending on the result of the comparison, either a single entry or a 2×2 block is picked as a pivot.

PARDISO implements different levels of parallelism. The first is related to trees and is generally exploited by all the sparse direct solvers: since the elimination tree is processed from the leaves to the root (bottom-up) during the factorization phase—and vice versa during the forward/backward substitution—, any node can be processed as soon as its children have completed the elimination operations. This allows a partially parallel computation for the processes as it is outlined in Figure 4.1 for a small example. The second type of parallelism exploit the presence of multiple processes to distribute the factorization phase, assigning the supernodes to the different processes by applying a 2D supernode-process mapping. The third and last type, finally, is called “pipelining parallelism” and is applied in combination with a supernode splitting strategy designed to increase load-balance and the solver’s pipeline efficiency. The splitting mechanism can be applied when a consistent number of processes is available and consists in dividing large supernodes into smaller blocks called “panels” which can be mapped onto the processes through a one-dimensional distribution scheme. The size of the panels, of course, is kept large enough to exploit the BLAS performance.

The pipelining parallelism can be implemented in a bottom-up traversal of the tree starting by allocating a queue containing all the leaves of the tree, each of them identifying one task to be completed, and reserving a group of processes. Every process asks to be assigned to one of the remaining tasks in the queue until all the panels are factorized, i.e., until the queue is empty. The process removes the assigned panel from the queue and allocates a new, separate queue of all the descendants of the panel’s supernode. As long as there are descendants in the new queue, the process performs the factorization steps with a right-looking strategy, then removes the descendant from the queue. This phase requires communication between the processes and must be synchronized, since

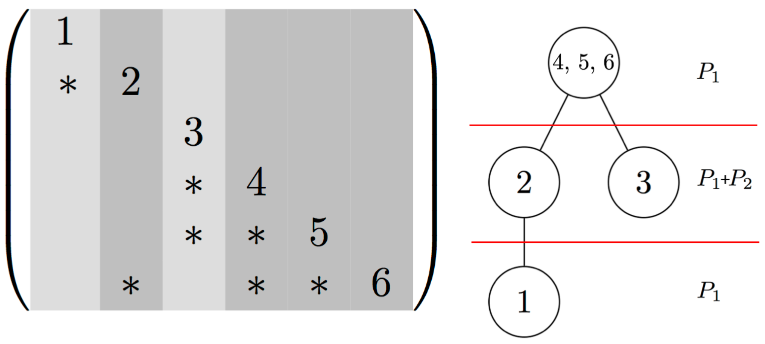

Figure 4.1. A small example of tree parallelism considering four supernodes (made by columns $\{1\}$, $\{2\}$, $\{3\}$, and $\{4,5,6\}$) and two processes (P_1 and P_2). Supernode ① is eliminated by process P_1, while ③ can be treated by P_2; as soon as the computation on ① is completed, P_1 can proceed to work on ② in parallel.

the descendants of the panel may be assigned to different processes. After this phase is completed, the internal factorization is computed—independently from other processes—and the queue of tasks is updated with the panels that are going to be involved in the next step. Notice that, as soon as it completes the factorization of a supernode, the process ends the right-looking phase by informing immediately all the supernodes having a ancestor/descendant relationship with the completed supernode: this allows other processes to start the factorization on those supernodes right away.

PARDISO applications

PARDISO was designed for the parallel scalable solution of large sparse systems arising in semiconductor device and process simulations [Schenk et al., 1999]. More recently, the Schur-complement feature of the solver has been widely used in stochastic optimization [Petra, Schenk, Lubin and Gärtner, 2014; Petra, Schenk and Anitescu, 2014]. The goal of the work is to design a scalable framework for the solution of stochastic optimization problems coming from the optimization of the USA power grids. The solution of such problems aims at finding the optimal operation of the facilities producing energy at the lowest cost and guaranteeing a reliable service. The inclusion of renewable energy as the wind, together with the impossibility to obtain complete and very precise weather forecasts, introduces increased uncertainty in the model. This results in larger optimization problems with a huge number of variables. The presence of both dense and sparse features in the linear systems involved, the ill-conditioning of the interior-point matrices (excluding the usage of iterative methods and forcing the adoption of a direct one), and the saddle-point structure of the matrices suggest a Schur-complement-based solution of the systems. On a related topic, Kourounis et al. addressed the solution of nonlinear optimal power flow problems through interior point methods. The modelling of distributed storage devices carries important computational challenges from the introduction of large scale renewables. The authors designed an efficient Schur-complement-based approach exploiting the PARDISO properties, designed ad hoc for the problem structure, several order of magnitude faster than the competitors, and saving a significant amount of memory [Kourounis et al., 2018].

A second important PARDISO application is the non-equilibrium Green's Function (NEGF) formalism for nanoelectronic devices [Kuzmin et al., 2013]. Recent advances in nanoelectronig device design require an effort to overcome the difficulites in simulatin nanotransistors. From the first transistor created in the late 1950s to the most recent ones, the size of the electrical components reduced by a factor 10^{10}. The challenges of such a size reduction are enormous and affect the whole industry, causing the producers to investigate single-atom transistors. As a drawback, the ab-initio—i.e., without any supporting experimental data—simulation of such tiny components would require enormous computational challenges arising from the involvement of the density functional theory framework. One of the most effective techniques is the NEGF, and is used to study the electronic and thermal properties of nanoscale transistors. In such framework, the computation of the so-called "retarded Green's function" is a selected inversion problem, where the diagonal of the inverse of a large, sparse, structurally symmetric matrix is required. PARDISO has been successfully used, showing excellent performance in the solution of realistically sized examples. On a side note, in this work (chapter 8), we are presenting a matrix-free stencil-based framework for the stochastic estimation of the retarded Green's function; PARDISO will be used as a baseline for the evaluation of the framework's accuracy and time performance.

Finally, we mention an application of the parallel Takahashi's approach (section 2.7) in genetics. In chapter 7 we introduce a scalable framework for the treatment of sparse/dense datasets in genomic prediction, aiming at the stochastic estimation of parameters for plants breeding. This genetic framework uses both the Schur-complement computation and the selected inversion features of PARDISO. The stochastic estimation process, in fact, requires the evaluation of the diagonal of the inverse of a dense matrix presenting a large, sparse block. The block structure of the data suggests the application of a block-based Gaussian elimination technique, where the selected entries of the sparse block—computed by PARDISO—are used in the evaluation of the parallel version of the Takahashi's formula (section 2.7). The results proposed in [Verbosio et al., 2017] and [De Coninck et al., 2016] show the good performance in the stochastic estimation framework with considerable memory and time savings.

4.2 PSelInv—Parallel selected inversion

PSelInv [Jacquelin et al., 2016] is a parallel framework for the selected inversion of sparse matrices, and is the parallel implementation of SelInv, an algorithm for the extraction of selected entries of the inverse who is in turn built on top of an efficient left-looking supernodal LU factorization [Lin et al., 2011].

PSelInv is based on the block-formula for the LU factorization (2.11), and can be interfaced with any supernodal factorization provided by different software, such as SuperLU and SuperLU_dist [Demmel et al., 1999; Li and Demmel, 2003]. Analogously to PARDISO, an elimination tree is created and it is traversed bottom-up to complete the factorization of the single supernodes. PSelInv, however, does not include any supernode-splitting technique, but aims at the distribution of the LU factors and entries of the inverse to a number of processes through MPI directives. The MPI processes are arranged in a virtual rectangular grid, and the factors are distributed according to a 2D block-cyclic pattern (see, for more details, Figure 6.2 and [Blackford et al., 1997]). The tree parallelism described in section 4.1 is exploited by PSelInv, however, the 2D cyclic distribution of both the factors and the inverse entries may prevent independent tasks to be executed completely in parallel. For this reason, the authors aim at the reordering of the computational tasks and the overlapping of computation and communication. The communication scheme is relatively simple: a process owning a diagonal block of a given supernode of the L factor must broadcast said block to all the processes in the same column of the virtual process grid who own a block of the same supernode. This is necessary to complete the factorization phase of the left-looking strategy. For the computation of the entries of the inverse, instead, the communication pattern is slightly more convoluted. Each block of the L factor must be shared with all the processes within the same grid column of processes who own a block of the inverse belonging to the same supernode. Since there is no guarantee that the sender and the receivers belong to the same communication group, this strategy is implemented using a series of point-to-point MPI send/receive calls. From the node-level point of view, we will show in the following chapter that this communication strategy causes significant overhead and latency.

PSelInv applications

Since both SelInv and PSelInv were designed for the treatment of electronic structure problems, they have been tested [Jacquelin et al., 2018] on matrices arising from (discontinuous Galerkin) density functional theory software such as [Lin et al., 2012] and [Soler et al., 2002]. The performance of the implementation of PSelInv—available in the PEXSI solver [Lin et al., 2009, 2013]—will be tested against PARDISO on selected inversion problems, in the following chapter.

4.3 MUMPS—Multifrontal massive parallel solver

Amestoy et al. have developed a multifrontal massively parallel solver (MUMPS), based on the simple assumtption that, given the LU factorization of any matrix, in order to compute the (j,k)th entry of the inverse, one simply has to extract the jth entry from the solution of the linear system (2.5) [Amestoy et al., 2001]. Exploiting the sparsity of the right-hand side, i.e., the kth column of the identity, MUMPS applies a "pruning" strategy on the elimination tree [Amestoy et al., 2015, section 2.2]. Such strategy is based on the fact that, in order to compute the (j,k)th entry of the inverse, the only nodes of the tree to be traversed are the ones on the paths from node $\textcircled{j}$ to the root and from $\textcircled{k}$ to the root. By doing so, the nodes which do not belong to these paths are not visited ("pruned" from the tree). As discussed before in this document, many applications need to compute a number of entries of the inverse in the order of the size of the matrix, e.g., the diagonal, or even bigger. If this is the case, computing the entries of the inverse for so many right-hand sides would require a prohibitive amount of storage, hence the evaluation of the entries must proceed in blocks. The entries are computed in blocks of variable size, e.g., 16 at a time, although this mechanisms may result in numerous accesses to some parts of the factors required for the calculation in multiple blocks, depending on the position of the required entries. In order to reduce this operation's cost, the authors developed an algorithm for splitting the right-hand side and an out-of-core strategy. The latter is combined with a combinatorial policy for the reordering [Amestoy et al., 2012]: once that the list of requested entries of the inverse is partitioned into blocks of a given block-size, each of them containing a smaller number of elements, such blocks are reordered to ensure that the entries are kept contiguous once that the blocks are merged as the software proceeds in managing the right-hand sides (therefore keeping the BLAS routines efficient). After the reordering is completed, for each of the blocks, the elimination tree is pruned accordingly. Then, for each of the nodes in the pruned tree, the block of right-hand sides to be included in the computation is determined according to the position of the entries in the block. Finally, the non-leaf nodes are merged to reach the predefined block-size, until the dimension of root node reaches the block-size itself. Obviously, the last block of right-hand sides may have a fewer number of entries. In [Amestoy et al., 2012], the authors show that partitioning the requested entries of the inverse in blocks in order to minimize the traversal of the tree is equivalent to the tree-partitioning problem, and show that this is an NP-complete problem [Theorem 3.2]. They propose few heuristics and build a model for the extraction of the diagonal entries on top of that.

Both the grouping strategy on the entry blocks and the sparsity-exploiting mechanism on the right-hand sides make MUMPS suitable not only for the selected inversion, but also for other applications where the simultaneous computation on multiple right-hand sides is requested. The solver is, in fact, widely used as a solver for sparse linear systems, however the large computational effort required by the blocking, postordering, and merging process may cause some lack of efficiency when the evaluation of a large number of entries of the inverse, e.g., the diagonal entries, is requested.

4.4 FIND—Fast inverse using nested dissection

Finally, we overview briefly the FIND (fast inverse using nested dissection) algorithm, specifically designed for the computation of the diagonal of the inverse [Li et al., 2008]. The key idea of the algorithm is the computation of many LU factorizations of a given $n \times n$ matrix A, where each factorization is designed to evaluate one specific entry of the inverse's diagonal. Once that one sparse LU factorization is given, in fact, recalling the Takahashi's formula [Takahashi et al., 1973]—and adapting (2.9)—, one can compute the bottom-right entry of the inverse of A as $(A^{-1})_{nn} = 1/U_{nn}$. The sparse LU factorization is of course computed after a symbolic factorization step, i.e., after A has been reordered to reduce the fill-in. The reordering permutation is crucial in the design of the algorithm: once that the entry in position (n, n) has been computed, A is permuted again in order to move to the bottom-right position all the nodes of the corresponding graph associated to the diagonal entries, one after the other. By doing so, after every reordering/factorization, the formula for $(A^{-1})_{nn}$ can be simply applied. At a first glance, this procedure would require the computation of n reordering/factorization couples, however the authors suggest a strategy to reduce the computational cost based on ad hoc permutations of A, order of the factorizations, and storage of intermediate parts of the different LU factors. The idea is to recursively subdivide the mesh associated with A (where each of the nodes corresponds to a row/column of the matrix and an edge from $\textcircled{i}$ to $\textcircled{j}$ represents a nonzero entry A_{ij}) into clusters, and the clusters are arranged into a binary tree—where every cluster is a node and its sub-clusters are its children. After that, an elimination process resembling the Gaussian elimination is performed. The process is divided into two steps: in the first, the inner nodes of each cluster are eliminated, while the second step consists in removing all the nodes who are external to any of the current leaf clusters. At every level of refinement every cluster is split into two sub-clusters, and the clusterization obtained at lower levels is reused for efficiency. The approach is inspired by the nested dissection algorithm [George, 1973], but it includes some differences. Once that a leaf cluster is defined, matrix A is reordered in a way that the entries of the leaf cluster form a bottom-right block. Then, the columns corresponding to the complement of the leaf cluster (with respect to the whole graph) are eliminated in order to obtain the entries of A^{-1} belonging to the leaf cluster. After a certain level of refinement is reached, the complementary of the leaves are quite large clusters, hence they overlap. In order to save computation, a so called “complement cluster tree” is created, where the parent of two leaves correspond to their intersection, while a parent would be the union of its children in the regular cluster tree. Both the trees are traversed to perform the elimination with the support of other tree data structures [Li and Darve, 2012, Section 6].

The main goal of FIND is again the computation of the non-equilibrium Green’s functions for two-dimensional nanotransistors, in particular the retarded Green’s function. It has been tested against the standard recursive Green’s function method [Sahasrabudhe et al., 2014], proving good performance and accuracy. Interestingly, the authors also designed an extension for the FIND algorithm [Li and Darve, 2012], in order to compute also the lesser Green’s function, needed for computing the charge and current densities of the nanodevices, and some of the off-diagonal entries. The method is based on the fact that the evaluation of the lesser Green’s function can be extrapolated by the LU factorization of the retarded Green’s function and the assumption that the latter and the self-energy matrix—essentially describing the boundary conditions of the problem—share a similar nonzero structure.

4.5 General observations

In this chapter we provided an essential overview of the most diffused algorithms and packages for the direct evaluation of the entries of the inverse of sparse matrices. In particular, we selected the algorithms suitable for the computation of the diagonal of the inverse. We can see that many of them (PSelInv, FIND, PARDISO) have been designed and/or successfully used in nanoelectronic structure problems, such as nanotransistor ab-initio simulations, a topic of paramount importance for the development of modern electronic devices. Related to this topic, we mention that other ad hoc solvers have been designed, such as the extension to the hierarchical Schur-complement method [Hetmaniuk et al., 2013]. Exploting an efficient $LDL^\top$ factorization of the NEGF matrix and a specific order of the operations, the method aims at keeping the fill-in as low as possible, and has been applied to 3D nanoscale devices [Zhao et al., 2016]. From a more general point of view, instead, we mention UMFPACK (based on the unsymmetric multifrontal method) [Davis, 2004], which combines a column pre-ordering strategy to limit the fill-in with a right-looking factorization. The package analyzes the matrix and automatically selects the most appropriate pre-ordering/pivoting strategy, however this is loosely related to the selected inversion topics treated in this document, so we are not investigating it any further.

Since computing the minimum fill-in obtainable for a matrix is a NP-complete problem [Yannakakis, 1981], it is clear that the design of heuristics for the reordering strategies are the key element of the state-of-the-art algorithms and determine in large part their efficiency. For this reason, many different approaches based on graph partitioning/relabelling have been designed through the years, as en example, the 1969 (reverse) Cuthill-McKee algorithm [Cuthill and McKee, 1969; Liu and Sherman, 1976]. Currently used methods, however, rely on multilevel and recursive methods such as the above-mentioned nested dissection and minimum degree, and possibly refinement techniques such as the novel parallel p-Laplacian refinement [Simpson et al., 2018]. Moreover, the parallelization techniques and reduction of the communication volume play a central role. The introduction of fill-in entries, in fact, complicates the communication between processes and supernodes during the traversal of the elimination trees. This aspect must be taken into account unless communication-avoiding stategies are included, e.g., imitating the communication-optimal LU factorization (CALU), designed for dense matrices [Grigori et al., 2011]. This method aims at reducing latency for dense LU routines implementing communication avoiding pivoting techniques for the factorization of blocks of columns. Considering that the final goal is to exploit the efficiency of the BLAS/LAPACK routines and the vectorization of modern compilers, the design of splitting techniques (such as the PARDISO panel subdivision) and the parallelization of the refinement methods may also help in the design of agile solvers.

In light of the discussion just concluded, in the following chapter we are analyzing the node-level performance of PARDISO and PSelInv. We will present the scaling performance of implementation of the two algorithms using multiple threads/processes (C++ code supported by MPI/OpenMP paradigm), and test their behavior on sparse and sparse/dense matrices. The choice of the datasets will be made keeping in mind the final goal, which is the implementation of the parallel Takahashi's formula. Analyzing the scaling results, we will justify the choice of including PARDISO in our sparse/dense framework in order to exploit its parallelization of the Schur-complement computation and the selected inversion.

Chapter 5

Comparative experiments for factorization-based selected inversion

The goal of this chapter is to analyze the performance of PARDISO against state-of-the-art direct solvers. The reason lies in the fact that the computation of the parallel Takahashi's selected inverse requires a sparse solver for the main computational kernels, from the selected inversion of the sparse blocks, to the Schur-complement calculation. Additionally, we show how the treatment of sparse-dense matrices, i.e., sparse matrices coupled with dense rows and columns, can increase the complexity of the state-of-the-art solvers, justifying the need of a sparse-dense framework for the usage of the parallel Takahashi's method. We aim at the usage of such framework in real life applications, and we show one in chapter 7, where we treat sparse-dense data in the field of genetics. As a final objective, we prove the efficiency of PARDISO as a sparse selected inversion solver and justify its usage in the sparse-dense framework designed in section 2.7. Among the modern software, we choose to compare the solvers PARDISO (section 4.1) and PSelInv (section 4.2); the first available in the PARDISO libraries, version 5.0.0, the second included in the PEXSI solver, version 0.9.0 [Lin et al., 2009, 2013]. The experiments aim at the evaluation of the node-level performance of the two software, and measure the time needed to compute the diagonal of the inverse of several matrices by PARDISO, using 1, 8, and 16 OpenMP threads, and by PEXSI, with 1, 8, and 16 MPI processes. The tests have been performed on a machine equipped with four octa-core Intel Xeon CPU E5-2650 @2.00 GHz and 128 GB of total memory.

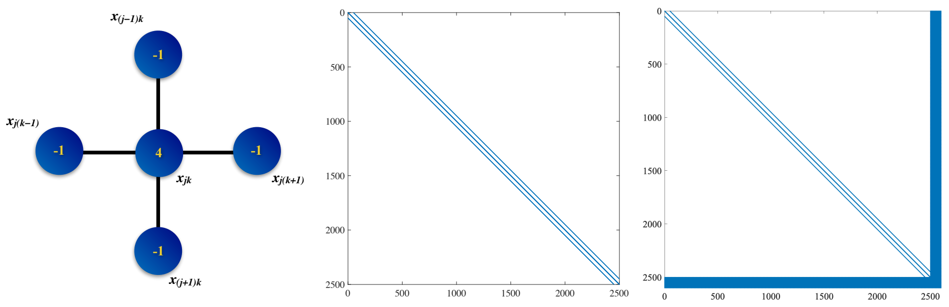

Figure 5.1. Weighted two-dimensional stencil for the finite differences discretization of the Laplace operator (left). Laplacian matrix generated by the 5-points 2D stencil on the left on a 50×50 grid (center). Laplacian matrix generated by the 2D stencil, coupled with dense rows and columns for the case $N = 50^2 + 100$ (right).

5.1 Setup of the node-level comparative tests

The test cases we decided to use to compare the node-level performance of the two direct solvers are made made of two different groups of matrices. The first consists of stencil matrices generated by the finite differences discretization of the Laplace operator, on both 2D and 3D regular grids, using a 7-points stencil analogous to the one depicted in Figure 5.1 (left and center). The second group is created considering finite difference Laplacian matrices over smaller 2D/3D grids, coupled with a fixed number (i.e., 100) of randomly generated dense rows and columns, according to the structure presented in Figure 5.1 (right). We consider three $n \times n$ 2D grids for $n = 500, 1000, 2000$, and three different $n \times n \times n$ 3D grids for for $n = 50, 80, 100$. The size of the matrices involved, N, is either n^2 or n^3 for the matrices in the first set, while it is $n^2 + 100$ or $n^3 + 100$ for the sparse/dense ones. The sparse dataset requires smaller computational effort than the sparse/dense one because of the low bandwidth, which is $n - 1$ for the 2D discretization and $n^2 - 1$ for the 3D one, while it becomes $N - 1$ for both the discretizations in the sparse/dense dataset. Both the solvers reduce the bandwidth during the symbolic factorization phase by applying a reordering to the rows and columns. PARDISO assign this phase to METIS, while PSelInv uses SuperLU_dist.; both analyze the sparsity pattern and group the different columns into supernodes, gathering together columns with the same sparsity pattern (chapter 4). In this chapter we show also how the two solvers use different parallelization techniques who may lead to considerably different communication patterns between processes, supernode splitting techniques, and, ultimately, execution time. The lower bandwidth of the sparse dataset, coupled with its stencil-structured nature, makes the reordering strategies more effective: the underlying mesh handled by the graph-partitioning software, included in the symbolic factorization packages, is particularly easy to treat and guarantees a good result for the minimization of the edge-cut and the load balance. On the other hand, we underline that the inclusion of the dense blocks cause the complexity of the inversion algorithm to grow the new nonzero structure complicates the graph partitioning problem. Additionally, we have to consider that the nonzero density for the sparse/dense matrices is considerably higher than in the sparse ones and this aspect creates higher fill-in in the factorization phase, when combined with the challenging graph partitioning solution. While it is well known that discretizing the Laplace operator leads to a total of $\mathcal{O}(N)$ nonzeros, precisely $n(5n-4)$ for the 2D case and $n^2(7n-6)$ for the 3D case, the number of nonzeros in the sparse/dense dataset is increased of $h(2N+h)$ when adding h dense rows and columns. This can raise dramatically the number of nonzeros, by a factor of approximately $1+\frac{2}{5}h$ for the 2D case and $1+\frac{2}{7}h$ for the 3D case. In our dataset, where $100(2N+100)$ nonzeros are added, we have that the sparse/dense factor presents around 41 times the nonzeros for the 2D discretization and 29 times for the 3D discretization. The final numbers are recapped in Table 5.1.

Table 5.1. Dataset for the node-level performance analysis. The sparse and sparse/dense datasets. The last column indicates the nonzero density of the matrix.

Dataset	Discretization	n	N	nnz	density (%)
sparse	2D	500	250 000	1.2480e+06	2.00e–03
		1000	1 000 000	4.9660e+06	5.00e–05
		2000	4 000 000	1.9992e+07	1.25e–05
	3D	50	125 000	8.6000e+05	5.50e–03
		80	512 000	3.5456e+06	1.35e–03
		100	1 000 000	6.9400e+06	6.94e–04
sparse/dense	2D	500	250 100	5.1258e+07	8.19e–02
		1000	1 000 100	2.0501e+08	2.05e–02
		2000	4 000 100	8.2000e+08	5.13e–03
	3D	50	125 100	2.5870e+07	1.65e–01
		80	512 100	1.0596e+08	4.04e–02
		100	1 000 100	2.0695e+08	2.07e–02

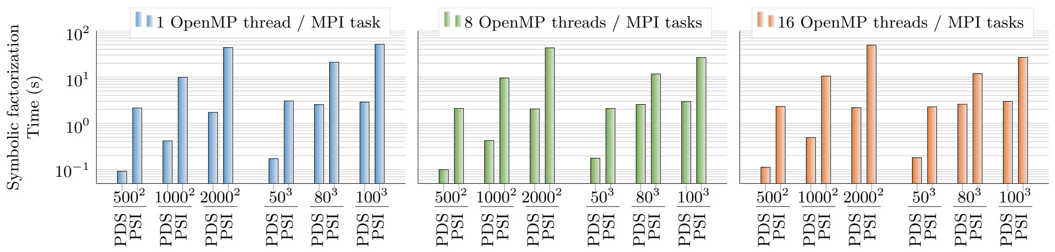

Figure 5.2. Symbolic factorization of the sparse matrices–comparing PARDISO (PDS) and PEXSI (PSI) execution time on 1, 8, and 16 threads/-tasks.

In the following section we show how both the consistent increase of the nonzeros and the complexity of the new nonzero structures affect the performance of both PARDISO and PSelInv. We will prove however that PARDISO not only succeeds in treating all the problems in the datasets, but is also able to do it in considerably less time. For this reason we pick it as the best direct solver capable of computing the diagonal of the inverse on the sparse/dense datasets and we will include it in the parallel Takahashi's framework, to be used for the treatment of genomic prediction datasets in chapter 7.

5.2 Node-level performance analysis

As part of the node-level performance experiments, we show not only how PARDISO can be preferred for the selected inversion of sparse matrices, but also how it is still the best choice for sparse/dense data. We report the time consumed by each of the direct solvers, showing that the PARDISO competitor cannot deal with the presence of dense rows and columns in the sparse structure, even if they are included in a small number. Schenk [2000] showed that that the computational complexity for the direct solution of a sparse linear system of equations arising from a regular finite element 3D grid is $\mathcal{O}(N^2)$ for the numerical factorization, and $\mathcal{O}(N^{4/3})$ for the triangular solution phase. On the other hand, PARDISO can still manage such cases successfully, making it our software of choice in the treatment of sparse/dense selected inversion datasets, even though we notice a lack of scalability.

The results relative to the pure sparse matrices are presented in Figures 5.2–5.4; from a first look we see that PARDISO outperforms PSelInv in all the stages of the selected inversion process. Let us start from the symbolic factorization, i.e., Figure 5.2. The time measured for PEXSI includes the reordering of the

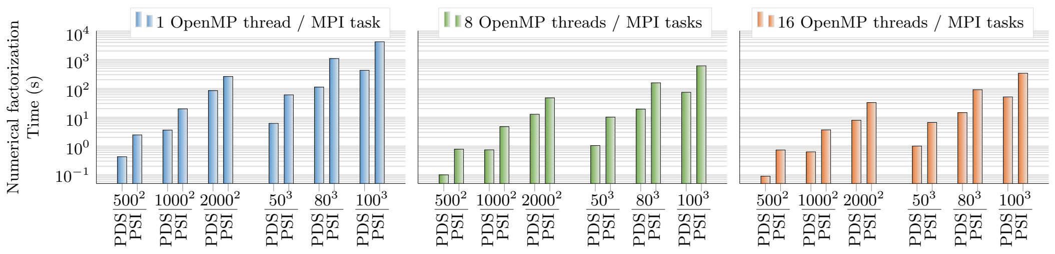

Figure 5.3. Numerical factorization of the sparse matrices–comparing PARDISO (PDS) and PEXSI (PSI) execution time on 1, 8, and 16 threads/-tasks.

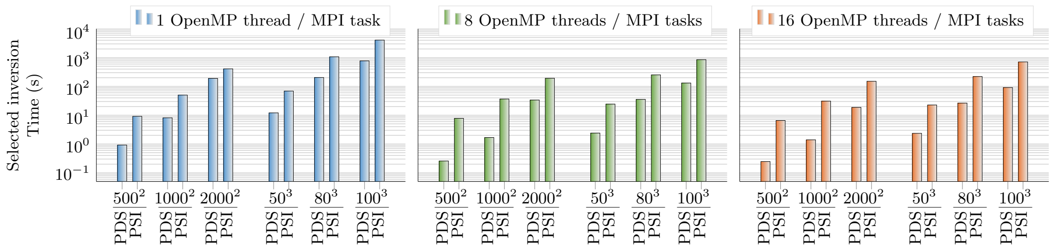

Figure 5.4. Selected inversion of the sparse matrices (after the factorization is completed)–comparing PARDISO (PDS) and PEXSI (PSI) execution time on 1, 8, and 16 threads/tasks.

matrix, the conversion of the internal structures into matrices, and the design of the communication pattern (see section 4.2 for further details). We can see that PARDISO is roughly 20 times faster than PSelInv on the 2D matrices, while the speedup is on average 10× on the 3D ones. For the numerical factorization (Figure 5.3), instead, the speedup is slightly lower, on average around a factor of 7. The same happens in general when performing the third phase (Figure 5.4) but the speedup factor in this case varies more, from 2× for the single-threaded $N = 2000^2$ case (left plot), to 30× on the $N = 500^2$ running on 8 threads (central plot). Additionally, the selected inversion phase can be performed by PARDISO up to three orders of magnitude faster than PSelInv in the sparse-dense case ($N = 1000^2 + 100$).

As predicted, we notice that the execution time for the sparse-dense dataset is higher than the pure sparse dataset for both PARDISO and PSelInv (Figures 5.5–5.7). PEXSI, however, suffers from heavier performance degrading than the first—mostly in the selected inversion phase. Also, PEXSI's execution on the 2000 × 2000 grid terminates prematurely since the 128 GB of memory

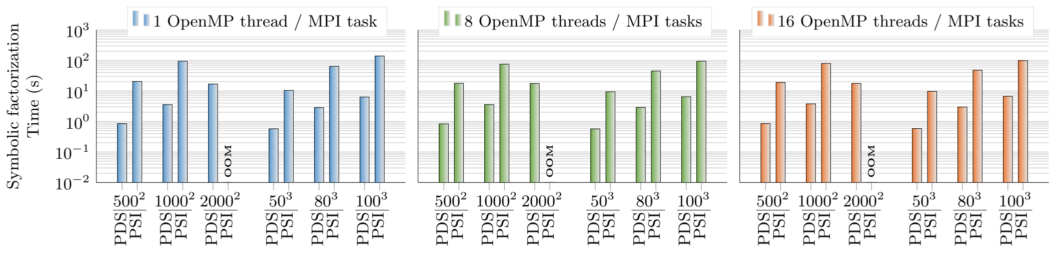

Figure 5.5. Symbolic factorization of the sparse-dense matrices–comparing PARDISO (PDS) and PEXSI (PSI) execution time on 1, 8, and 16 threads/-tasks.

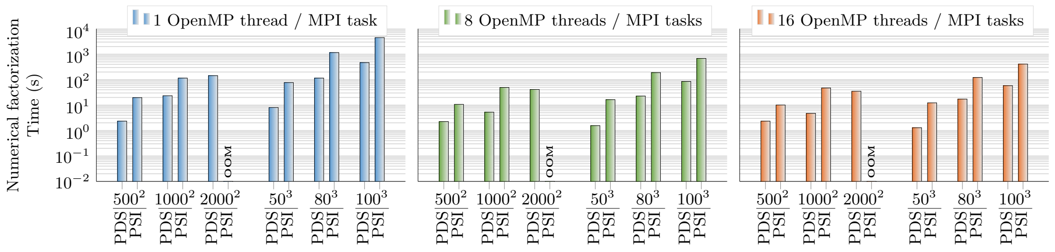

Figure 5.6. Numerical factorization of the sparse-dense matrices–comparing PARDISO (PDS) and PEXSI (PSI) execution time on 1, 8, and 16 threads/-tasks.

are not enough for the solver to perform the factorization. This translates into an out-of-memory error, hence this testcase is marked with the label "OOM" in the plots. Recalling Table 5.1, for this testcase, the allocated matrix would consists of 4,000,100 rows and a total of roughly 0.82 billion nonzeros, resulting in prohibitive fill-in. The structure of the sparse/dense matrices is particularly troubling for both the solvers, however the reordering strategy of PARDISO and its internal Cholesky factorization allow it to store only the upper triangular part of the matrix (reducing of a factor ~ 2 the nonzeros) and resulting in a total of only 0.57 billion nonzeros in the Cholesky factor. Looking in details the single phases, during the symbolic factorization (Figure 5.5), PARDISO gains a factor 19 speedup on average (on the testcases that can be actually solved by PEXSI, excluding therefore the 2000×2000 grid), while this becomes more consistent for the selected inversion phase (Figure 5.7). Here, the presence of a consistently higher number of nonzeros in the PEXSI's factors than the PARDISO's ones translates into huge time consumption for the creation of the communication pattern between supernodes, and also to higher execution time for the

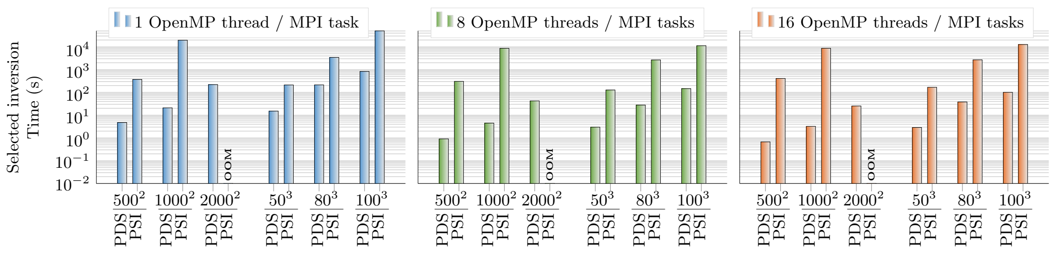

Figure 5.7. Selected inversion of the sparse-dense matrices (after the factorization is completed)–comparing PARDISO (PDS) and PEXSI (PSI) execution time on 1, 8, and 16 threads/tasks.

numerical evaluation of the entries. PARDISO is therefore up to three orders of magnitude faster on the 2D grids (2680× for the 1000 × 1000 grid) and up to two orders of magnitude faster (125× speedup for $N = 100^3 + 100$) for the 3D testcases.

In light of the analysis just completed, we can appreciate the good performance and memory savings that PARDISO ensures, and justify the decision to include PARDISO in our sparse-dense framework implementing the parallel Takahashi's method.

Chapter 6

Scalability of the parallel Takahashi's method

In this chapter, we describe the performance of the C++ implementation of the parallel Takahashi's algorithm for sparse, real, symmetric matrices. We present weak and strong scaling tests for the computation of the diagonal of the inverse, which is a selected inversion problem. The scaling tests are performed at the Swiss National Supercomputing Center (CSCS), on up to 400 nodes of the cluster Piz Dora, equipped with two 12-core Intel Haswell CPUs (Intel Xeon E5-2690 v3) per node. The C++ code is compiled on the Piz Dora using the Cray C++ compiler, and is supported by the Intel MKL ScaLAPACK and BLACS libraries together with MPI. According to the definition of Algorithm 2, section 2.7, we test its performance on a set of random matrices that can be decomposed into blocks as

$$C = \begin{pmatrix} A & B \\ B^\top & D \end{pmatrix},$$

considering A as a sparse block and B and D as dense.

6.1 Parallel Takahashi's method's implementation details

Recalling the structure of the algorithm, the workflow is based on the distribution of the dense blocks B and D among a 2D $p \times p$ ($p \geq 2$) virtual grid of processes, and the parallel computation/inversion of the Schur complement of block A in C. The implementation of Algorithm 2 requires some effort in both the design of the memory distribution pattern and the processes' arrangement.

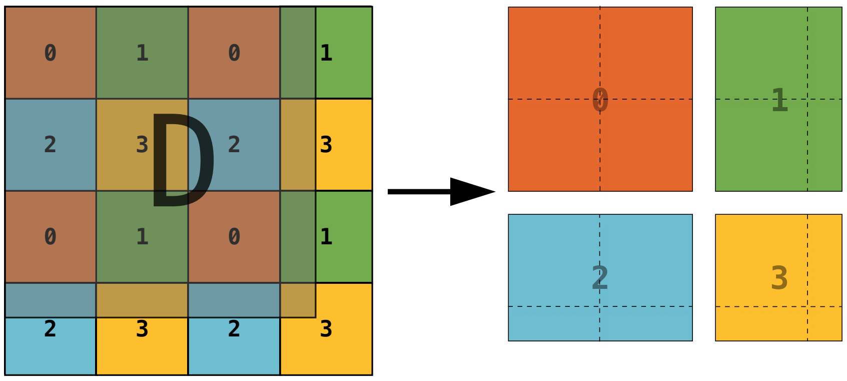

Figure 6.1. Example of simplified implementation for the ScaLAPACK 2D cyclic distribution. Left: matrix D is represented in gray and each process, identified by a number in $\{0,1,2,3\}$, owns a block of it. The colored blocks represent the maximum memory allocated by each process. Right: the actual subblocks of D saved on the local memory of each process. The scheme shows the part of D saved by each process.

Let us suppose that block D has size n_D. According to the ScaLAPACK standard [Blackford et al., 1997], D must be sliced into submatrices of a fixed size and assigned to the processes in the grid according to a 2D circular pattern. Let us fix s as the maximum size of the submatrices, which means that each subblock will contain at most $s \times s$ entries of D. It follows that in order to partition D into such subblocks, it must be sliced into $b^2 = \left\lceil \frac{n_D}{s} \right\rceil^2$ patches. The ceiling function $\lceil \cdot \rceil$ is used because and additional patch must be added in case n_D is not a multiple of s. Considering now a single process belonging to the grid, (i,j) for some $0 \le i,j \le p-1$, it will be assigned to store a number of rows and columns equal to

$$n_i^r = \left\lceil \frac{b-i}{p} \right\rceil \text{ and } n_j^c = \left\lceil \frac{b-j}{p} \right\rceil, \tag{6.1}$$

respectively, meaning that the process will need to allocate enough memory to store block D_{ij} of size $(s\,n_i^r) \times (s\,n_j^c)$. Notice that this rule may lead to an overestimation of the required memory that can be computed as $\omega = (bs)^2 - n_D^2 = (bs - n_D)(bs + n_D)$ numbers allocated that are not actually necessary for the computation. In the worst case, given a value for n_D, the choice of s is such that $n_D = sk+1$ for some $k \ge 1$; since $(bs - n_D) = (s(k+1)-(sk+1)) = (s-1)$, the memory overestimation becomes

$$\omega = (s-1)(bs + n_D).$$

An analogous mechanism is used for the storage of B. Each process (i,j) will store $s\,n_j^c$ columns of B and $s\,n_i^r$ rows of $B^\top$, adding up to $(n_A)\times(s\,n_j^c)$ and $(s\,n_i^r)\times(n_A)$ elements, respectively. The processes on the diagonal of the grid ($i = j$), instead, do not need to store both the columns of B and the rows of $B^\top$, because this would result in duplicating the necessary memory, since the two blocks coincide. The distribution pattern for a 2×2 process grid is portrayed in Figure 6.1.

As an example, let us consider $n_D = 1000$, a block size $s = 128$, and $p = 3$ (nine processes arranged in a 3×3 grid). Given these parameters, we have $b = \lceil \frac{1000}{128} \rceil = 8$, $n_0^\star = n_1^\star = 3$, and $n_2^\star = 2$. The memory allocated by each process (i,j) is summarized in the following scheme.

$_i\backslash^j$	0	1	2
0	384×384 $n^r = n^c = 3$	384×384 $n^r = n^c = 3$	384×256 $n^r = 3, n^c = 2$
1	384×384 $n^r = n^c = 3$	384×384 $n^r = n^c = 3$	384×256 $n^r = 3, n^c = 2$
2	256×384 $n^r = 2, n^c = 3$	256×384 $n^r = 2, n^c = 3$	256×256 $n^r = n^c = 2$

As mentioned before, the processes belonging to the last column and row of the grid (i.e., $i = 2$ or $j = 2$) allocate an amount of memory exceeding the actual memory sufficient to store the part of D assigned to them. This overestimation is handled by ScaLAPACK during the computation and communication phases. In a similar way, matrix B is divided into "vertical stripes" and distributed among the columns of the grid according to a 1D cyclic distribution, hence allocating $n_B\times 384$ elements on $(0,0)$ and $(0,1)$, and $n_B\times 256$ on $(0,2)$. See a comparison between the 1D and the 2D cyclic distribution in Figure 6.2.

From Algorithm 2 (page 20), we can see that the p^2 blocks D_{ij} forming the Schur-complement are computed in place to avoid memory duplication: each of them belongs to the local memory of a process (i,j) and is overwritten with the corresponding part of the Schur-complement of A in C, independently from the other processes. In order to achieve this parallelism, each process (i,j) assembles a smaller matrix, C_{ij}, composed coupling A, the slice B_j, the stripe $B_i^\top$, and D_{ij}, as in Figure 6.3. The process computes then the Schur-complement of A in C_{ij}, i.e.,

$$D_{ij} \longleftarrow D_{ij} - B_i^\top \underbrace{A^{-1}B_j}_{Y_j} = D_{ij} - B_i^\top Y_j\,. \tag{6.2}$$

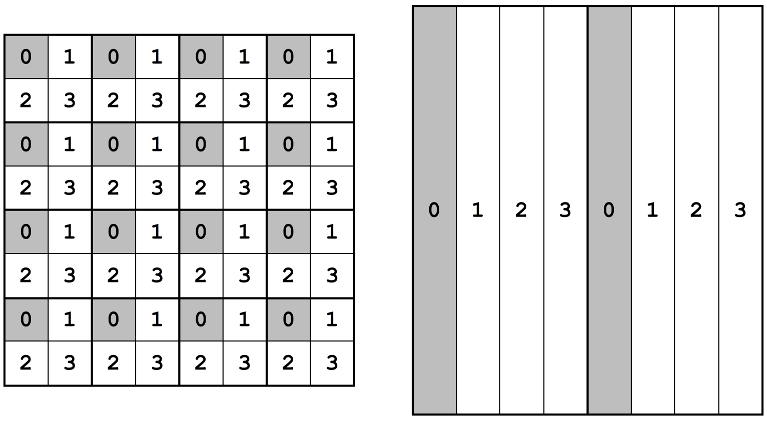

Figure 6.2. Example of two-dimensional (2D) (left) and one-dimensional (1D) (right) block-cyclic distribution among a 2×2 grid of processes labeled $\{0, 1, 2, 3\}$. The 2D distribution is recommended when the BLACS and ScaLAPACK libraries are used, in order to optimize the communication between the processes.

Block A must be stored in the memory of every process, even for large-scale datasets. For memory and computation optimization, A is stored in compressed sparse row format (CSR). The sparse matrix is encoded using three vectors: one for the nonzeros, `nz`, one for the column indices of the nonzeros, `cols`, and one for the indices pointing at the first nonzero of each row, `prows`. Supposing to have ζ nonzeros in A, `nz` is a vector of length ζ where the nonzeros are listed row-wise, `cols` has the same length and indicates the column-index of the nonzeros, while `prows` has length $n + 1$ and indicates where, in the list of the nonzeros, the first nonzero of each row is placed. The first entry of `prows` equals 0 (or 1 in a 1-based index environment such as FORTRAN or MATLAB), while the last equals ζ (or $\zeta + 1$ in a 1-based environment). Using this format, the nonzeros of the `k`-th row of the matrix (for `-1<k<n` in a 0-based environment or `0<k<n+1` in a 1-based environment) are stored in `nz`, in positions `prows[k],prows[k]+1,...,prows[k+1]-1`, and have column index `cols[prows[k]], cols[prows[k]+1], ..., cols[prows[k+ 1]-1]`, respectively.

Property 1 (Compressed Sparse Row (CSR) format storage). *Given an $n \times n$ sparse matrix A with ζ nonzeros, the CSR format reduces the memory requirements for storing A from n^2 elements to $\mathcal{O}(\zeta)$, precisely $2\zeta + n + 1$ (assuming that both the data types used for the matrix entries and for the nonzero indices occupy the same amount of memory). Additionally, if A is symmetric, then only the upper triangular*

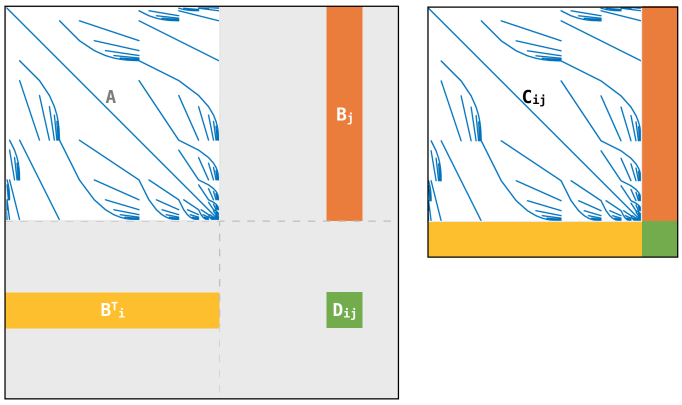

Figure 6.3. The colored part represents the data belonging to process (i, j). The contribution to the Schur-complement D_{ij} is computed by the process after assembling a new matrix C_{ij} (on the right), made of the parts stored in its own memory, which is smaller in size than the original matrix C (gray).

part can be stored, i.e., assuming that A has δ nonzeros on the diagonal, the cost is reduced to $\zeta - \delta + n + 1$.

Sparse matrix routines are designed to perform matrix-vector and matrix-matrix operations exploiting these structures, optimizing the performance not only of low level operations, but also reordering, factorization, solution of linear systems, and selected inversion. This system is analogous to the CSC (compressed sparse column), where the nonzeros are stored in a column-wise order and the roles of `cols` and `prows` are exchanged.

Going back to (6.2), we see that A must be treated by every process to compute the local Schur-complement, however the inverse is never computed explicitly, exactly as in the sequential case (Algorithm 1, line 2): the product $A^{-1}B_j$ is again calculated as the solution of the system $AY_j = B_j$, locally by the process (i, j). Note that the solution of the system is identical for the processes belonging to the same column of the process grid because of the 1D cyclic distribution of B. For this reason, matrix Y is stored in a pattern identical to that used for B, i.e., on the processes belonging to the first row of the grid. The only exception is the root process $(0, 0)$, where a selected inverse of A is actually computed in order to provide the base for the Takahashi's update of the diagonal entries, according to Algorithm 2, line 10. Here the factorization phase is not repeated during the computation of the Schur-complement component D_{00}, taking advantage of the factors stored in memory during the selected inversion. For performance, in the light of the node-level comparison tests run in Part II, we choose to use the solver PARDISO [Kuzmin et al., 2013; Petra, Schenk and Anitescu, 2014; Petra, Schenk, Lubin and Gärtner, 2014] for this phase; in order to save memory and computing time, PARDISO computes and stores the LU factorization of A before addressing the linear systems and use the stored factors to solve both the systems. Additionaly, the factors are reused when the selected inverse of A is computed explicitly on $\mathcal{N}(A)$. As a matter of fact, recalling the Takahashi's formula, PARDISO stores internally also the entries on $\mathcal{N}(L + U)$. As a last step, in order to complete calculating the entries of the diagonal of the inverse, the inverse of the whole Schur-complement D is calculated, as required by the bottom-right block of decomposition (2.14). Using the ScaLAPACK/BLACS routines such as send, receive, factorize and invert, it is possible to compute a distributed inverse of the block. The inverse overwrites D and follows the same distribution pattern (Algorithm 2, line 9). Finally, before the selected elements of A^{-1} are updated, the matrix-vector product $D^{-1}Y_j$ and the dot product between $Y_i^\top$ and $D^{-1}Y_j$ are calculated by PBLAS, since both factors are distributed across the process grid. The final result is stored in the master process.

This step concludes the algorithm. In order to simplify the computation, once that the nonzeros of A^{-1} are computed, they are stored in a vector, and the values are updated computing and communicating the matrix-vector and dot products on the fly for all the necessary matrix indices. In the following section we analyze the strong scaling performance of the algorithm's main components, including this last one.

6.2 Strong scaling tests

With the strong scaling experiments we want to demonstrate the performance of the parallel Takahashi's method on a high number of processes, showing how the reduction of the local workload on the single processes results in lower the execution time. The dataset is made of symmetric, positive definite matrices from the SuiteSparse Matrix Collection (former "University of Florida Sparse Matrix Collection") [Davis and Hu, 2011]. This matrices are chosen to be large and sparse block A in the block decomposition, and are listed in Table 6.1. We couple block A with a dense rectangular matrix B and a dense symmetric, positive definite block D of a fixed dimension. Both B and D are randomly generated, so there is no guarantee about the positive definiteness of the local matri-

Table 6.1. Matrices used for the strong scaling tests from the University of Florida Sparse Matrix Collection.

Name	**Order**	**Nonzeros**	**Density**	**Application**
pwtk	217,918	5,926,171	0.13e-01 %	Structural problem
af_shell3	504,855	9,046,865	0.36e-01 %	Structural problem
parabolic_fem	525,825	2,100,225	0.74e-05 %	Comp. fluid dynamics

ces C_{ij}. Being aware of that, the Cholesky factorization $LDL^\top$ of the C_{ij}'s might fail, so we instruct PARDISO to treat each block as a symmetric indefinite matrix when computing the factorizations. This worsens the overall performance, although uniformly among the processes, so it does not affect the scaling. In order to use the whole memory available at each node, we allocate a single MPI task on each node of the cluster. The performance plots are shown in Figures 6.4–6.6. There are three different quantities reported, indicating the time required for:

(a) the parallel computation of the complete Schur-complement of A in C, i.e., $S = D - B^\top A^{-1} B$, measured as the elapsed time between the start and the end of the computation of the S_{ij}'s;

(b) the collection of the entries on the diagonal of $C^{-1}{}_{22}$ and the update of the entries of A^{-1} as described in Algoritm 2, line 17;

(c) the whole process, excluding the matrix loading from file, which is approximately equal to the sum of items (a) and (b).

The strong scaling tests are designed to use an increasing number of nodes, from 16 to 400, and a fixed number of OpenMP threads, 16, used by PARDISO. We include only the square numbers of tasks, so that they can be arranged in square grids as 4×4, 5×5, and so on until 20×20. We underline that the time is not reported for less than 16 nodes, i.e., 1, 4, and 9, because of memory constraints: the memory required to store the C_{ij}'s exceeds the total memory available on each node. For this reason, the speedup reported in the plots (gray bars) is intended to be computed with respect to the minimum number of nodes suitable, giving unitary speedup for 16 nodes. Notice that the speedup is reported for the total execution time only, and is simply computed as the ratio between the total time for 16 nodes and the total time for the other configurations. The computation time is dominated by the final stages of the computation, i.e., the

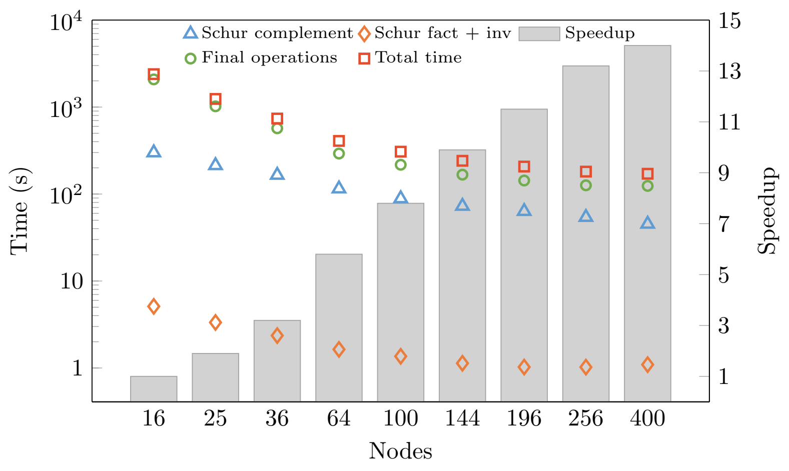

Figure 6.4. Strong scaling tests on Piz Dora as a function of the number of nodes involved on matrix af_shell3. Grey bars (right axes) show speedup obtained.

collection and update of the entries of the diagonal (green circles). This phase is communication-bounded, since it requires the usage of parallel, dense linear algebra libraries, BLACS, involving MPI communication routines. The information belonging to the single processes is collected by the master process to assemble $C^{-1}{}_{11}$ and $C^{-1}{}_{22}$—these operations correspond to the loop beginning at line 11 in Algorithm 2.

Analyzing the speedup, one can see that from 16 to 400 nodes (25×), the total execution time is reduce by a factor spanning from around 14× to 16×. We observe that a factor 25 in the number of nodes means that the size of the process grid changes from 4×4 to 20×20, i.e., the number of columns in the process grid handling the 1D distribution of the blocks B_i's and Y_i's increase from 4 to 20 (only 5×). Finally, looking at the speedup for the "pwtk" case (Figure 6.6) we notice a small loss of performance when passing from 196 to 256 ranks. We observed that the reordering computed by PARDISO in the factorization phases using 256 nodes is slightly worse than the one computed when using 196 nodes, cue to an unexpected inconsistent behavior in the reordering software (METIS). We also notice that the communication phase (green circles) presents a little bump in the execution time, revealing a possible system-related issue in the communication between nodes. Except the small problem just presented, we consider the scalability satisfying.

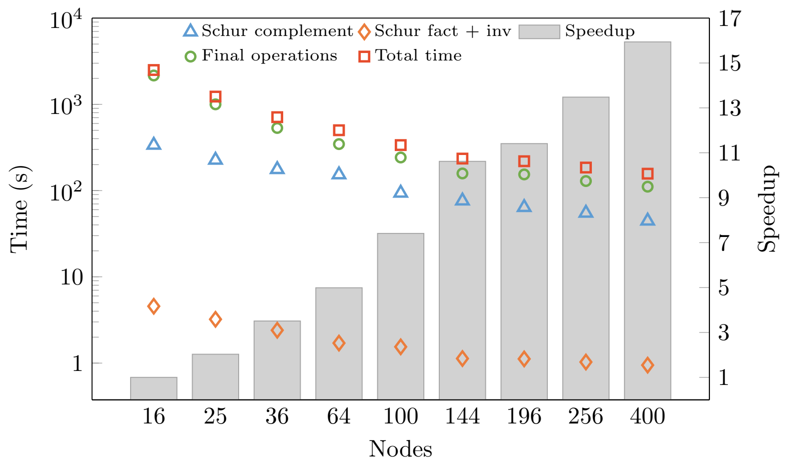

Figure 6.5. Strong scaling tests on Piz Dora as a function of the number of nodes involved on matrix parabolic_fem. Grey bars (right axes) show speedup obtained.

6.3 Weak scaling tests

For the weak scaling tests we designed a dataset where block A consists of 3D Laplacian matrix, i.e., a 7-points finite difference discretization of the Poisson equation on a 3D regular grid. Blocks B and D are dense and randomly generated. We fix $n = 100$ and choose an $n \times n \times n$ regular grid, giving A have size $n_A = n^3 = 10^6$ and $n^2(7n-6) \approx$ 7e+06 nonzeros (see section 5.1) and we couple

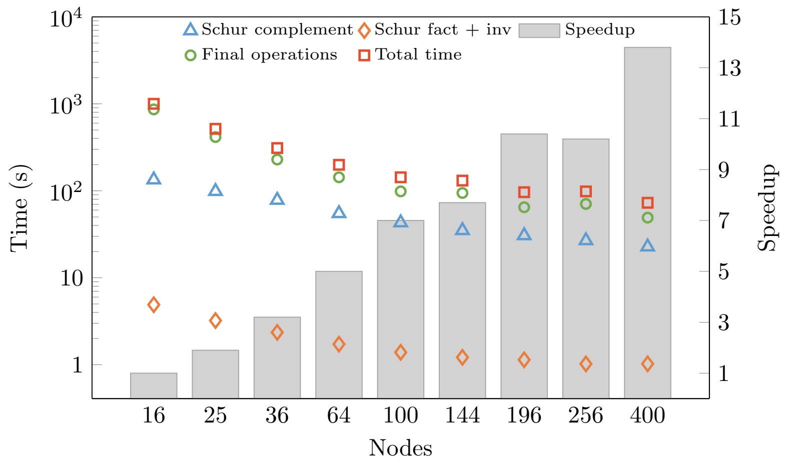

Figure 6.6. Strong scaling tests on Piz Dora as a function of the number of nodes involved on matrix pwtk. Grey bars (right axes) show speedup obtained.

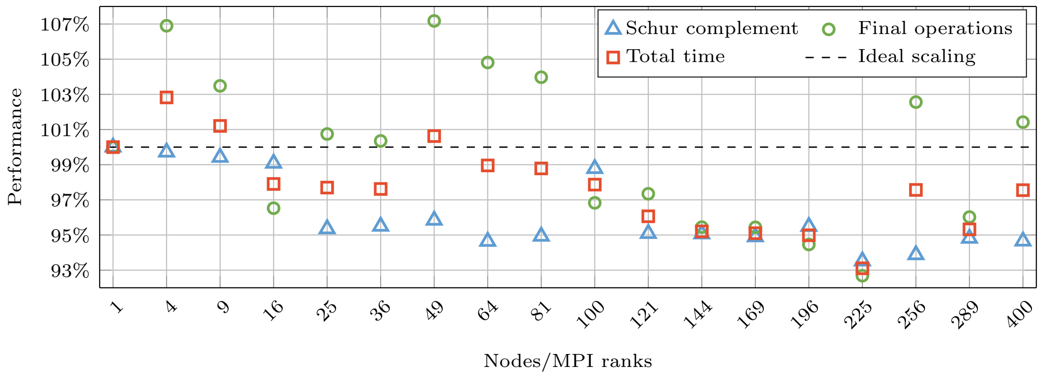

Figure 6.7. Weak scaling tests for the parallel Takahashi's method. Every processor works on a $10^6 \times 10^6$ 3D finite difference Laplacian matrix coupled with 1000 dense rows and columns.

it with a dense block D of variable size n_D. Considering an increasing number of processors arranged in a $p \times p$ grid, for $p = 1, 2, \ldots, 16, 17, 20$, we consider $n_D = 10^3 p$, i.e., $n_D = 1000, 2000, \ldots, 17000, 20000$, in order to keep the workload constant on each processor. For every processor (i, j) in the grid, in fact, the subblock C_{ij} is constrained to be a $(n_A + 1000) \times (n_A + 1000)$ matrix consisting of the sparse block A coupled with 1000 dense rows and columns, for a total addition of around two billion nonzeros and reaching a total of $1 + \frac{2}{7}1000 \approx 287$ times the nonzeros in the Laplacian matrix (see section 5.1 for details).

Using this configuration, we run the weak scaling experiments on Piz Dora, keeping one MPI task per node, and test the scalability of both the Schur-complement construction and the final communication routines. The plot in Figure 6.7 reports the performance for building the Schur-complement (blue triangles), collecting and updating the final elements of the diagonal of the inverse (green circles), and the overall performance of the code (red squares). Again, in order to evaluate the performance (speedup), we simply divide the execution time for the single computing stages on one node by the corresponding time on the other configurations. A value smaller than 1 indicates a loss of performance with respect to the single-node execution, while a greater value indicates a boost. In the plot we report the corresponding percentage, where 100% corresponds to speedup equal to 1. Looking at the speedup in details, one can see that there is no loss nor gain bigger than 7%; additionally, the speedup for the total execution time depends for the 57% on the Schur-complement build and for the 43% on the collecting routines. This confirms that the good scalability of the two most time-consuming parts makes the total execution time scale as well, despite the presence of other non-scaling operations, such as the inversion of the Schur-complement.

Part III

Selected inversion applications in genetics and nanoelectronics

Chapter 7

Gaussian process regression in genomic prediction frameworks

We now present an application in the field of genetics where the scalable Takahashi's method can be successfully used. In genomic prediction, genetic data are used to predict the performance of hybrid breeds to be developed in the future or to detect superior genotypes in a population. The underlying statistical framework requires the design of a linear mixed model allowing the simultaneous modelling of both fixed effects and random effects assumed to be, in general, normally distributed. This is a Gaussian process regression. In the case of plant breeding, the effects of the genotypes may vary in different environmental conditions, hence the model should include the so-called marker-by-environment effects. Such effects are computationally challenging, since they make the size of the matrices involved to grow consistently, however the sparsity of the data can be exploited to reduce the computational burden. The parallel Takahashi's method (section 2.7) fits perfectly the requirements a scalable fast solver should have to tackle this problem, where the parameter estimation process requires the computation of a selected inverse, precisely, the diagonal of the inverse. In the following sections we introduce the genomic prediction problem and the linear mixed model, then we examine the performance of the parallel Takahashi's method in the maximum likelihood estimation process.

7.1 Genomic prediction models for plant and animal breeding

The field of genomic prediction studies the effects of environmental conditions on plant and animal breeds. The design of a mathematical model describing such effects requires the study of genetic markers usually referred to as SNP (single-nucleotide polymorphism) markers. As an example, one way to approach breeding selection is to consider the effects of the environmental factors on the genotype of the single animals and include this contributions into the final model for the selection of the phenotypes. Given a set of observations of a characteristic important for breeding, each of the plants (or animals) can be considered as a sample drawn from some definable population of elements. Those members of the population presenting the same quality (e.g., a quantitative measurement) do not necessarily share the same genome, so the random variable describing the genetic value contributing to the development of the physical characteristic mentioned above remains unobservable, although realized. The goal of the genomic prediction analysis becomes then the definition of a set of estimators for the parameters that would describe the genetic features responsible for the interesting phenotypic traits [Searle, 1997]. One commonly used method to model is the construction of a linear mixed model, where the effects of the SNPs are drawn from a normal distribution. Additionally, quantitative traits important to breeders are usually regulated by a large number of loci (i.e, the significant positions on a chromosome who can be occupied by a gene or a genetic marker), giving high-density SNP information, for thousands of markers. The introduction of genome-wide analysis has increased dramatically the complexity of the method, although providing benefits to both plants and animal breeders [Crossa et al., 2010; Aguilar et al., 2010].

Especially in plants breeding, it is important to take into account the environmental effects [Cooper et al., 2005]. In different environmental conditions, different environment characteristics such as high soil moisture or low solar radiation impact differently the phenotypic traits, and the markers may present different effects. These effects are referred to as “genotype by environment” effects and are usually assumed to be normally distributed. A relatively novel approach [Hayes et al., 2009; De Coninck et al., 2014] suggests the inclusion of the “marker by environment” effects in the model, in order to explicitly model the interaction between markers and environment, an aspect that is ignored when studying the simple genotype by environment effects. In the following sections, we are going to study a linear mixed model including both the effects, for large-scale data. The inclusion of these effects leads to a huge number of effects to be estimated, e.g., when studying the effects of a few hundreds of environments on thousands of markers, leading to the order of 10^5 equations. Considering, however, that observations are usually made in very specific environments, the modelling matrix is highly sparse, reducing, at a first glance, the computational burden of large-scale datasets. We will analyze the structure of the model and the nature of the mathematical structures involved, and exploit the presence of both dense and sparse data through the formulation of a sparse/dense solver.

7.2 The linear mixed model for genomic prediction

The methods typically adopted to design genomic selection and prediction problems are called "linear mixed models" and couple fixed effects together with random genetic effects [Meuwissen et al., 2001] in the form

$$y = X_e b_e + Z_s u_s + Z_m u_m + e\,. \tag{7.1}$$

Here,

y is the vector of n observations, typically, phenotypic data,

b_e is a vector of k fixed environmental effects,

u_m is a vector of l random marker-by-environment interaction effects,

u_s is a vector of m random genetic marker effects, and

e is the residual error term.

The random genetic effects modeled by u_m and u_s are introduced by assigning one effect to each marker used to genotype the individuals. The key assumption that only a small portion of such markers has a significant effect on the observations makes the matrices sparse: the design matrices for the effects are X_e ($n \times k$, modeling the fixed effects), Z_m ($n \times m$, model for the marker by environment interactions), and Z_s ($n \times s$, describing the random genetic effects). Typical size ranges for the variables are listed in Table 7.1. Notice that Z_m is the encoding matrix for the marker by environment effects, defining whether a particular allele is homozygous or not[1]. The coding standard defines a relation

[1]An allele is a variation of a gene, and the combination of one or more of them can result in different phenotypic traits. If the most frequent allele at a locus (gene) coincide, we talk about "homozygosity", we talk about "heterozigosity" otherwise.

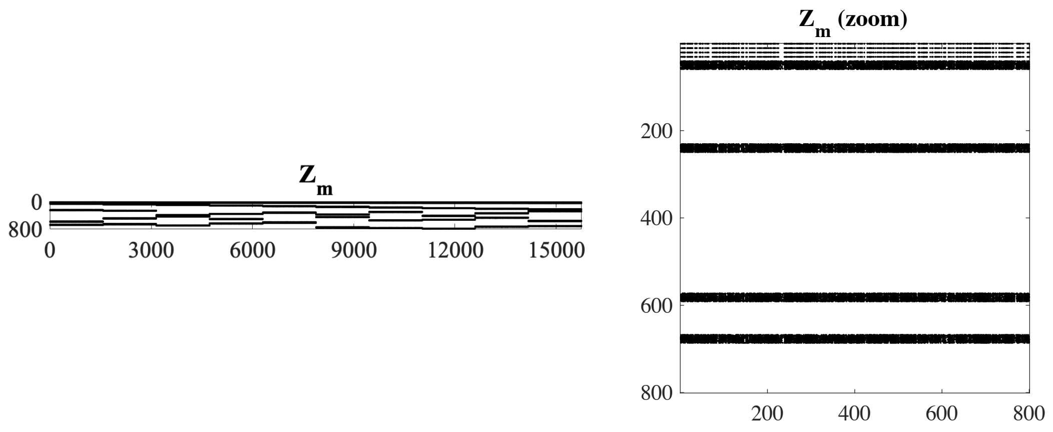

Figure 7.1. Example of a marker by environment matrix Z_m. The sparsity pattern (left) and a zoom on a square submatrix (right).

between the alleles of a chromosome and the environment effects and reports a 0 if there is homozigosity on the most frequent allele, 1 for heterozigosity on the most frequent, and 2 for heterozigosity in the less frequent one. The 0/1/2 standard is chosen to ensure the sparsity of the Z_m (refer to [Strandén and Christensen, 2011] for more details about allele coding).

Example. Let us consider now a small example considering $n = 800$ observations, $m = 1575$ QTLs (quantitative trait loci) and $k = 10$ environments. Figure 7.1 shows the nonzero pattern of Z_m: we can see that the matrix is sparse, with 4% nonzeros and about 640 nonzeros per row. On the right, there is a zoom of the first 800 rows and columns. Analogously, also the matrix modelling the fixed effects X_e is highly sparse, with one nonzero per row. We see that the chosen coding ensures high sparsity for the matrices. The random effects are instead modelled through Z_s, which is highly dense since about 40% of the entries are nonzeros (Figure 7.2).

As mentioned before, genetic marker effects can be drawn from different distributions, however it is common practice to adopt a normal distribution $\mathcal{N}$, mainly for its simplicity, computational efficiency, and predictive performance compared to other distributions [De Coninck et al., 2014, 2015, 2016]. The

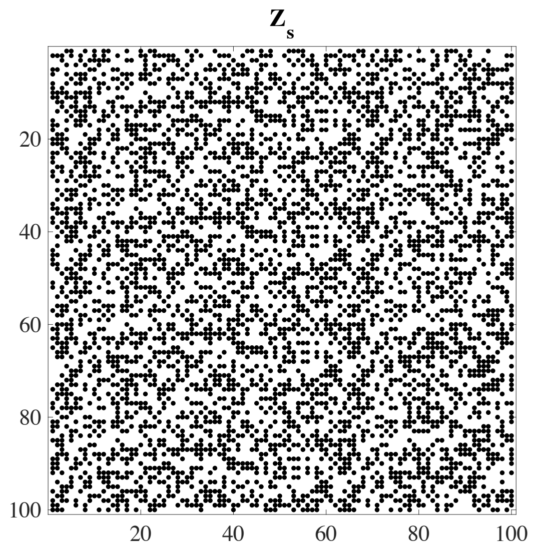

Figure 7.2. First 100 rows and columns of the matrix Z_s modelling the random environmental effects in the linear mixed model (7.1). The nonzeros cover about 40% of the structure.

choice just described translates into the model

$$\begin{cases} \begin{pmatrix} u_s \\ u_m \\ e \end{pmatrix} \sim \mathcal{N}\left(0, \sigma^2 \begin{pmatrix} \phi E & 0 & 0 \\ 0 & \gamma G & 0 \\ 0 & 0 & I_n \end{pmatrix}\right) \\ y \sim \mathcal{N}(X_e b_e, V) \\ V = \sigma^2\big(I_n + \phi\, Z_e E Z_e^\top + \gamma\, Z_s G Z_s^\top\big) \end{cases} \tag{7.2}$$

where the normal distribution has zero mean, I_n is an identity matrix of order n, and E and G are constant, symmetric, positive definite matrices of order l and k, respectively. Additionally, the parameters γ and ϕ, describing the variance components of the regression model, are unknown a priori and will be estimated through the maximization of the likelihood of the model. The assumption that each random effect correlates to a small number of the remaining effects makes the covariance matrices E and G to be sparse; this is the setup we are going to consider throughout the chapter to compute the estimates and predictions of the parameters in the model. We will see later that, under particular circumstances, both E and G can be considered to be identity matrices.

Name	**Description**	**Range**
n	observations	10^3 — 10^6
k	environmental effects	10^0 — 10^2
l	marker-by-environment effects	10^4 — 10^6
m	genetic marker effects	10^3 — 10^5

Table 7.1. Typical sizes for the variables involved in the genomic prediction problem.

7.3 The Average-Information Maximum Likelihood estimates

In order to predict the random effects and estimate the fixed ones, the first step consists in computing the best linear unbiased predictions and estimates (BLUP and BLUE), respectively. BLUPs and BLUEs are unbiased linear estimators and predictors of the mixed model's parameters that minimize the mean squared error, and can be computed as the solutions of the mixed model equations, see e.g., [Henderson, 1963, equations (19) and (20)],

$$\begin{pmatrix} {X_e}^\top X_e & {X_e}^\top Z_m & {X_e}^\top Z_s \\ {Z_m}^\top X_e & {Z_m}^\top Z_m + \frac{1}{\phi}E^{-1} & {Z_m}^\top Z_s \\ {Z_s}^\top X_e & {Z_s}^\top Z_m & {Z_s}^\top Z_s + \frac{1}{\gamma}G^{-1} \end{pmatrix} \begin{pmatrix} \hat{b}_e \\ \hat{u}_m \\ \hat{u}_s \end{pmatrix} = \begin{pmatrix} {X_e}^\top y \\ {Z_m}^\top y \\ {Z_s}^\top y \end{pmatrix}. \tag{7.3}$$

Here, $\hat{b}_e$, $\hat{u}_m$, and $\hat{u}_s$ are the estimates/predictions for b_e, u_m, and u_s, respectively, and the vector of the observations y is known. The coefficient matrix of this system, to be referred to as C, has the following block structure

$$C = \left(\begin{array}{cc|c} {X_e}^\top X_e & {X_e}^\top Z_m & {X_e}^\top Z_s \\ {Z_m}^\top X_e & {Z_m}^\top Z_m + \frac{1}{\phi}E^{-1} & {Z_m}^\top Z_s \\ \hline {Z_s}^\top X_e & {Z_s}^\top Z_m & {Z_s}^\top Z_s + \frac{1}{\gamma}G^{-1} \end{array}\right) = \left(\begin{array}{c|c} A & B \\ \hline B^\top & D \end{array}\right), \tag{7.4}$$

and blocks A, B, and D are defined by:

$$\begin{aligned} A &= \begin{pmatrix} {X_e}^\top X_e & {X_e}^\top Z_m \\ {Z_m}^\top X_e & {Z_m}^\top Z_m + \frac{1}{\phi}E^{-1} \end{pmatrix}, \\ D &= \begin{pmatrix} {Z_s}^\top Z_s + \frac{1}{\gamma}G^{-1} \end{pmatrix}, \text{ and} \\ B &= \begin{pmatrix} {X_e}^\top Z_s \\ {Z_m}^\top Z_s \end{pmatrix}. \end{aligned} \tag{7.5}$$

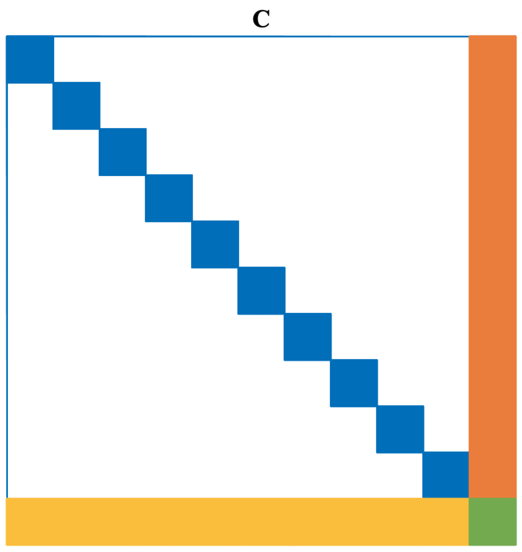

Figure 7.3. Coefficient matrix for the mixed model equations in genomic prediction C. In blue, the sparse block A, in orange B ($B^{\top}$ in yellow), and D in green.

Considering the structure of the design matrices, block A is square and sparse. This is true for mainly two reasons: first, as seen in the previous section, since every observation in the model comes from a single environment, Z_m is highly sparse and the product $X_e{}^{\top} Z_m$, even though dense, presents an exiguous number of nonzeros (no more than km). Second, assuming the observed population to be unstructured, we can speculate to have null covariance across the environments and, including the hypothesis of homoscedasticity, the covariance matrices equal the identity, i.e., $E = I_l$ and $G = I_m$ [Piepho, 2009]. Blocks B and D, on the other hand, are considered to be dense, where D is also square and is symmetric positive definite (SPD). The density of these blocks comes from the fact that Z_s is dense too, since genetic marker information are typically dense. Find an example for matrix C in Figure 7.3, where the colored parts represent the sparsity pattern of the different blocks A, B, $B^{\top}$ and D.

The bottom-right subblock of A and block D incorporate the reciprocals of the variance components of the distributions of the random effects, ϕ and γ. As mentioned in the previous section, these quantities are unknown a priori, and need to be estimated from the training data: the estimates can be computed by maximizing the log-likelihood function of the model with respect to the variance components [Gilmour et al., 1995]. The methodology we use is an iterative, gradient-based approach known as the "Average Information Restricted Maximum Likelihood" (AI-REML). The function to be maximized is the REML log-likelihood which is computed on a reduced set of data and does not take into accounts the parameters that are not interesting at a first stage (nuisance parameters). The expression for the REML log-likelihood ℓ reads

$$\ell(\sigma^2,\phi,\gamma) = -\frac{1}{2}\Big((n-k)\log\sigma^2 + l\log\phi + m\log\gamma + \log\det C + \frac{1}{\sigma^2}y^\top P y\Big), \quad (7.6)$$

where $P = V^{-1} - V^{-1}X_e({X_e}^\top V^{-1}X_e)^{-1}{X_e}^\top V^{-1}$ (refer to [Gilmour et al., 1995] and [Searle et al., 1992, Chapter 6] for more technical details). At each iteration of the AI-REML, an update vector containing the current guess for the parameters must be computed. At the kth iteration, the update vector reads $\delta t_k = (\sigma_k^2, \phi_k, \gamma_k)^\top$ and is obtained as the solution of the linear system

$$H\delta t_k = -\nabla\ell(t_k), \quad (7.7)$$

where the matrix H is the update matrix and $\nabla\ell$ indicates the gradient of ℓ. This iterative scheme is similar to the Newton's method, however, matrix H does not coincide with the Hessian of the objective function ℓ, since the Hessian can be very hard to construct. Such limitation is given by the complexity of the second derivatives of ℓ, hence the so-called AI matrix [De Coninck et al., 2014, Appendix A] is chosen as update transition matrix. The analytic expressions for the partial derivatives of ℓ with respect to ϕ and γ, i.e., the score functions, require the computation of the trace of the product of the inverse of large matrices:

$$\begin{cases} \dfrac{\partial\ell}{\partial\phi} = -\dfrac{1}{2}\left(\dfrac{l}{\phi} - \dfrac{\operatorname{tr}\left({C^{-1}}_{11}\right)}{\phi^2} - \dfrac{\hat{u}_m^\top \hat{u}_m}{\sigma_e^2\phi^2}\right) \\ \dfrac{\partial\ell}{\partial\gamma} = -\dfrac{1}{2}\left(\dfrac{k}{\gamma} - \dfrac{\operatorname{tr}\left({C^{-1}}_{22}\right)}{\gamma^2} - \dfrac{\hat{u}_s^\top \hat{u}_s}{\sigma_e^2\gamma^2}\right) \end{cases}. \quad (7.8)$$

By ${C^{-1}}_{11}$ and ${C^{-1}}_{22}$ we indicate the top-left and bottom-right blocks of C^{-1}, corresponding to blocks A and D in (7.4), however notice that ${C^{-1}}_{11} \neq A^{-1}$ and ${C^{-1}}_{22} \neq D^{-1}$. At a first glance, such blocks would require to know the whole inverse C, however the fact that only the trace is actually required, one simply needs to compute diag(C^{-1}).

The AI-REML algorithm

Given the expression for the log-likelihood (7.6) and the score functions (7.8), the AI-REML method implements the iteration (7.7) in Algorithm 6. Starting

Algorithm 6 . AI-REML–Average Information Restricted Maximum Likelihood estimation.

```
Input: observation vectors $y, u_m, u_s$, mixed model matrix $C$, initial guess
    $[\gamma_0, \phi_0]$, relative improvement $\epsilon$, maximum iterations maxit
Output: estimates $[\sigma^2, \gamma, \phi]$
 1: function LOG_LIKELIHOOD($C$, $\sigma^2$, $\gamma$, $\phi$)
 2:     return $-\big(k\log(\gamma) + l\log(\phi) + (n-m)\log(\sigma^2) + \log\det(C) + n - m\big)/2$
 3: end function
 4: $[\gamma, \phi] \leftarrow [\gamma_0, \phi_0]$
 5: for k = 1, ..., maxit do
 6:     $[A, B, D] \leftarrow$ UPDATE_C_BLOCKS($C$, $\gamma$, $\phi$)
 7:     $\sigma^2 \leftarrow$ EVAL_SIGMA($A$, $B$, $D$, $\gamma$, $\phi$)
 8:     $l \leftarrow$ LOG_LIKELIHOOD($C$, $\sigma^2$, $\gamma$, $\phi$)
 9:     $[s_\sigma, s_\gamma, s_\phi] \leftarrow$ SCORE($C$, $y$, $\sigma^2$, $\gamma$, $\phi$)
10:     if $\big(|s_\gamma/\gamma| < \epsilon\big)$ OR $\big(|s_\phi/\phi| < \epsilon\big)$ OR $\big(|l - l_{old}|/|l_{old}| < \epsilon\big)$ then
11:         break
12:     end if
13:     $H \leftarrow$ CALCULATE_AI_MATRIX($y$, $A$, $B$, $D$)        ▷ [De Coninck et al., 2014]
14:     $\delta \leftarrow$ SOLVE_SYSTEM($H$, $[s_\sigma, s_\gamma, s_\phi]$)        ▷ compute update $\delta = H^{-1}\nabla\ell$
15:     $[\sigma^2, \gamma, \phi] \leftarrow [\sigma^2, \gamma, \phi] + \delta$
16: end for
```

from an initial guess for the parameters σ^2, γ and ϕ, the expression for blocks A and D are computed according to (7.5) and used to compute the value of the log-likelihood (7.6). After that, the score functions and the gradient

$$\nabla\ell = \left(\frac{\partial\ell}{\partial\gamma}, \frac{\partial\ell}{\partial\phi}\right)$$

are evaluated; this is where the trace of the inverse is required and using a parallel solver might be necessary in order to make the computation feasible and efficient. The values of the score functions are arranged in a vector $(s_\phi, s_\gamma)^\top$ used as a right-hand side for the linear system (7.7). Notice that, since it does not involve any selected inversion process, we omit the estimation of σ^2. We just say that, provided that the values of (the estimates of) γ and ϕ are available, given the nature of ℓ, $\sigma^2 = \frac{1}{n-m} y^\top P y$. The solution of the linear system is the direction update to be added to the current vector of the estimates $(\sigma^2, \gamma, \phi)^\top$. This concludes the process, which should be iterated until at least one between the relative update of the log-likelihood, the relative increment of γ, or the relative increment of ϕ is less than a fixed threshold ϵ.

7.4 Numerical results

The tests presented in this section are designed to show the performance and capabilities of the sparse-dense treatment of genomic prediction data. The successful use of the parallel Takahashi's method results in the successful treatment of large-scale cases. The sparse-dense genomic prediction framework is designed for the estimation of parameters (γ and ϕ) to be chosen for maximizing the log-likelihood. The optimization process is presented in algorithm 6, and uses the parallel Takahashi's framework described in section 2.7 to compute, at every iteration, the gradient of the log-likelihood, whose expression requires the evaluation of the trace of large sparse matrices.

7.4.1 Setup of the numerical tests

The dataset we use is based on a series of simulated data, and presents a set of trials for plants with 1575 QTLs and span a variety of parameters, from 10,000 to 250,000 observations, where the largest dataset consists of 100,000 observations on 100 different environments and a total of 157,500 marker by environment effects. The software, additionally to the the phenotypic observations (vector y), reads from input files the model matrices X_e, Z_m, Z_s. The first are compressed in CSR format (section 6.1) due to their sparsity, while Z_s, encoding the random genetic effects, is dense and stored in a binary file. The root process reads the input and stores in its local memory the matrix-matrix products ${X_e}^\top X_e$, ${Z_m}^\top X_e$, ${Z_m}^\top Z_s$ and so on, and with these it completes two tasks: first, it assembles a copy of B and stores it in its memory until it is sent in strips to the processes for the parallel computation of the Schur-complement; second, it allocates A, computes its entries using the initial guess for the parameter ϕ_0, and sends it to all the other processes. All the processes read from file both X_e and Z_m, since they are sparse and this operation is in general quick, while they read Z_s stripe-by-stripe. The number and size of the Z_s stripes needed by each process to construct its own block D_{ij} is of course determined by the 1D/2D cyclic distribution standard described by BLACS/ScaLAPACK protocol and recalled in chapter 6. Once that the input phase is completed, the AI-REML iteration starts. The initial guess for both the parameters (γ and ϕ) is set to 0.001; the AI-REML iteration stops when either the ratio between two consecutive estimates of the parameters or the relative increment of the log-likelihood is less than 1%, hence we set $\epsilon = 10^{-2}$. If none of these conditions verifies before the maximum number of iteration is hit, i.e., 20, the algorithm fails.

The software is an implementation of the method defined before in Algo-

Table 7.2. Execution time and memory consumption for 1575 QTL genetic markers. The average time is intended per iteration.

Obs. (n)	Environm. effects (k)	M × E[†] effects (l)	Size of block A	Iter.	Average wall time (s)	Max. mem. per node (GB)
10,000	10	15,750	15,760	4	16.3	0.79
100,000	10	15,750	15,760	3	32.7	1.5
250,000	10	15,750	15,760	4	85.1	2.8
50,000	50	78,750	78,800	5	85.5	3.1
100,000	50	78,750	78,800	5	94.9	3.8
100,000	100	157,500	158,600	5	174.6	6.2

[†]: marker by environment

rithm 2 (chapter 2), and is a C/MPI/OpenMP framework. Analogously to what described in chapter 6, the code is supported by the Intel MKL ScaLAPACK and BLACS libraries [Blackford et al., 1997], and compiled with the GNU C++ compiler. All the tests are performed on a cluster consisting of 16 nodes, each one equipped with two 10-core Intel Xeon E5-2650 v3 @2.3 GHz and 128 GB of working memory. Each run is performed on 16 MPI tasks on 16 nodes, one task per node, while PARDISO uses 8 OpenMP tasks.

7.4.2 Analysis of the results

The main results are shown in Table 7.2, where we report the convergence of the AI-REML algorithm, an overview of the wall time needed by the method, and the average per-node memory. The dataset is ordered with respect to increasing average wall time per iteration, which coincide with a reordering done with respect to the environmental effects and the number of observations. An increasing number of observations for a fixed number of fixed effects, however, does not cause the memory consumption to grow excessively, although we see that the computing time increases due to a higher time consumption needed for the construction of the coefficient matrix.

We underline that the complexity of the AI-REML process would increase dramatically should the matrices be handled as dense, since the factorization and selected inversion process dominate the execution time. It is straightforward to see that the average time per iteration remains bounded by the cubic complexity expected from an approach where the matrices are treated as dense.

Chapter 8

Stochastic estimation of the non-equilibrium Green's functions

The goal of this section is to illustrate the results achieved by applying the stencil-based stochastic estimation of the diagonal of the inverse to the non-equilibrium Green's functions (NEGF) formalism in nanoelectronic device simulation.

The difficulties in simulating nanoscale semiconductors are due in large part to the problems involved in building and operating devices only a few nanometers in width. Modeling nanoscale transistors requires the effort of going beyond the classical drift-diffusion approach [Bank et al., 1983] and other effective mass approximations. The main direction of recent research in these areas aims at the reduction of the size of such electronic components, playing a central role in modern technological industry. The challenges of the huge size reduction experienced in the designed of transistors from the first models created in the 1950s, are enormous and affect the whole industry, pushing the producers to investigate alternatives to silicon transistors, such as single-atom transistors [Fuechsle et al., 2012]. These alternatives require to reformulate the existing models in order to substitute the previously made approximations and assumptions with quantum and atomistic governing equations. As a result, the community must implement an immense improvement in the class of simulations relying on computer aided design and based on first principles, i.e., without any empirical data, commonly referred to as "ab initio" simulations. The reason why this kind of simulations is needed, is dictated by the fact that at a nanometer scale empirical data and simulations previously made at a larger scale (such as micrometer) become unreliable because of quantum phenomena. Additionally, the quality of a device's design might be tested and validated only after it is built, a process that could lead to a waste of resources due to the high cost of producing such devices.

The problems arising from the ab-initio nanoelectronic device simulations are faced relying on a density functional theory framework augmented with quantum electron transport simulations. The setup of such a framework and the engagement of the ab initio simulations entail numerous challenges; some of them are successfully addressed by Calderara et al. [2015], combining CP2K, a package for density functional theory [VandeVondele et al., 2005], and the nanodevice simulator OMEN [Luisier et al., 2011].In order to perform successfully said simulations, it is convenient to build a mathematical model based on the nonequilibrium Green's functions [Luisier et al., 2006]. This is one of the most efficient techniques to perform this task and it has been widely used to study the electronic and thermal properties of nanoscale transistors and molecular switches on massively parallel architectures. In the following sections we are going to introduce a technique to avoid the limitations of the state-of-the-art methods used to compute the retarded Green's function on three-dimensional nanodevices, estimating its entries via a stochastic estimator based on the one introduced in chapter 3. The estimator will be tested on a dataset design to mimic the structures in OMEN, and we are presenting both its scaling performance and its accuracy.

8.1 The NEGF formalism in nanoelectronic device simulations

The main goal for the simulation of quantum devices is the solution of the single-particle stationary Schrödinger equation for every particle involved in the design of the device, in order to determine the energy state of the system [Luisier et al., 2011]. The idea at the basis of density functional theory is the possibility to use the electron density alone to express the energy of a system and, therefore, the particles' wave functions. For non-equilibrium systems such as transistors—because of the presence of a current flowing on the device—we need to also consider quantum electron transport. From here, some conceptual computational complications arise: nanotransistors, in fact, should be treated as finite systems, since they have finite boundaries, however they do not present, in principle, periodic boundaries. This assumption collides with the fact that transistors should be modelled with open boundary conditions (OBCs) to fit the hypothesis of a flowing current [Calderara, 2016, Chapter 2]. In order to treat nanotransistors or nanowires, it is possible to reduce the infinite domain to a finite equivalent one. The idea is to split the domain intro three parts: the active region and the two contact regions, L and R, on the boundary, immediately adjacent to the active region.The Hamiltonian of the system is computed on the active region and on the contact regions, excluding however the conceptually infinite remainder of the contact region external to L and R. This setup allows the use of the NEGF formalism, whose governing equations are

$$\left(I - H - \Sigma^L(E) - \Sigma^R(E)\right) G^R(E) = I\,, \tag{8.1}$$

where H is the block tridiagonal Hamiltonian matrix obtained by solving the Schrödinger equation for the given nanostructure and expressed through localized basis functions, Σ^L and Σ^R are the self-energy matrices containing the open boundary conditions, and I is the identity matrix. The unknown, $G^R(E)$, is the retarded Green's function at the energy level E for the injected electrons. Equation (8.1) describes then a matrix inversion problem, where $G^R(E)$, unknown, is the inverse of the quantity in parentheses; however, as described in [Lake et al., 1997] and [Calderara et al., 2015], in formula (8.1), the only entries to be explicitly evaluated are the ones on the diagonal of $G^R(E)$ and, in general, a subset of the ones belonging to the nonzero pattern of A. This is a selected inversion problem, according to definition 1, chapter 2. The solution of the selected inversion problem just introduced is usually addressed either via the so-called recursive Green's function (RGF) algorithm [Svizhenko et al., 2002] or using other algorithms [Kuzmin et al., 2013; Lin et al., 2011; Jacquelin et al., 2016; Zhao et al., 2016]. In many ways, the type of matrix updates performed in both algorithms is similar to those performed in the LU factorization and they represent, e.g., the main computational task for each MPI process in OMEN. The focus of the new scalable algorithms to be developed should be on the fast evaluation of the diagonal of the inverse performing better than, e.g., the evaluation of the entire inverse matrix based on successive application of a sparse direct LU decomposition of A. As the number of cores is expected to increase significantly faster than the memory, extreme-scale algorithms for computing the diagonal entries of the inverse matrix not only need to use a high number of cores per compute node, but also should incorporate different memory hierarchies and, in particular, should require much less memory per core compared, e.g., to the RGF method.

The particular structure of the matrix involved in the selected inversion problem recalls the structure of the finite difference discretization of the Laplace operator on a regular grid [Saad, 2003, Chapter 2]. In addition to that, the supplementary blocks describing the boundary conditions encoded in the Σ's are included. The discretization of the second order operator on a grid generates a multidiagonal symmetric matrix with low bandwidth (that could become nonsymmetric when the boundary conditions are included). The peculiar structure of these matrices shows up in various applications and it is of particular interest, mainly since the action of such a matrix on vectors can be emulated with stencil operations applied on a regular grid. This means that the matrix-vector operations may be computed in a matrix-free fashion, using stencil code, possibly optimized. In the following section we introduce and discuss the aspects of a selected inversion algorithm exploiting the stencil-related properties just mentioned, and aiming at the evaluation of the diagonal of the inverse using an implicit-matrix formulation for the basic linear algebra operations. The design of a solver for the selected inversion problem in the NEGF formalism translates then into the design of a selected inversion algorithm that involves only highly parallelizable matrix-vector and vector-vector operations. Recalling the properties of the stencil-structured matrices (definition 3, chapter 3), and the Krylov-subspace based formulation of the stochastic estimator for the diagonal of the inverse (Algorithm 4, page 34), we are going to apply the method to the computation of the diagonal of the retarded Green's function.

8.2 Stencil structure of the NEGF matrices

For the sake of simplicity, we define as A the NEGF matrix whose inverse is $G^R(E)$, introduced in (8.1):

$$A := I - H - \Sigma^L(E) - \Sigma^R(E)\,. \tag{8.2}$$

We want to study now the nonzero pattern of A and verify its stencil structure in order to express the matrix-vectors products through a stencil. In order to understand the structure, recalling that H is the Hamiltonian of the nanostructure, we introduce the finite difference discretization matrix for the Laplace operator.

8.2.1 The discretized Laplace operator on a regular grid

Let us consider a finite closed domain D, e.g., a square in $\mathbb{R}^2$, and a regular grid sampling D at equidistant points, analogous to the one introduced in Section 3.3.1 and in Figure 3.1:

$$\Omega_h = \{x_{ij} = (x_0 + ih, y_0 + jh)\,, \text{for } 1 \le i, j \le n\}\,, \quad h > 0\,.$$

Let us consider a function ϕ defined on Ω_h describing a 1:1 mapping of the points on the domain to another grid-shaped domain $\mathscr{G}$, i.e., $\phi : \Omega_h \longrightarrow \mathscr{G}$, as the discretization (sampling) of a class C^2 function $\varphi : D \longrightarrow \varphi(D)$ such that $\phi(x_{ij}) \overset{\text{def}}{=} \varphi(x_{ij})$ for all i, j. We consider $\mathscr{G}$ to be the image of the discrete function, hence a grid of real values shaped like Ω_h. Considering the central finite differences, one can discretize the first partial derivatives of φ at any point in the grid as the mean of the values assumed by the function on the previous and following point, in both directions

$$\begin{cases} \dfrac{\partial \varphi}{\partial x}(x_{ij}) \approx \dfrac{\phi_{(i+1)j} - \phi_{(i-1)j}}{2h} \\ \dfrac{\partial \varphi}{\partial y}(x_{ij}) \approx \dfrac{\phi_{i(j+1)} - \phi_{i(j-1)}}{2h} \end{cases}, \tag{8.3}$$

where $\phi_{ij} := \varphi(x_{ij}) = \phi(x_{ij})$. It follows immediately that, composing twice these definitions, we obtain the following expressions for the second partial derivatives:

$$\begin{cases} \dfrac{\partial^2 \varphi}{\partial x^2}(x_{ij}) \approx \dfrac{\phi_{(i+1)j} - 2\phi_{ij} + \phi_{(i-1)j}}{h^2} \\ \dfrac{\partial^2 \varphi}{\partial y^2}(x_{ij}) \approx \dfrac{\phi_{i(j+1)} - 2\phi_{ij} + \phi_{i(j-1)}}{h^2} \end{cases}. \tag{8.4}$$

From here, we can formulate a discretized version of the Laplace operator $\Delta = \nabla^2 = \sum \frac{\partial^2}{\partial x_i^2}$ as the stencil operator

$$\mathscr{L}\phi(x_{ij}) = \frac{\phi_{(i+1)j} + \phi_{(i-1)j} + \phi_{i(j+1)} + \phi_{i(j-1)} - 4\phi_{ij}}{h^2}, \tag{8.5}$$

described on the left of Figure 5.1. The value of the discretized Laplacian of the function ϕ at point x_{ij} is then computed as the weighted average of the values assumed by the function at x_{ij} and its four orthogonal neighbors; notice that this expression can be easily extended to $\mathbb{R}^3$ considering the six orthogonal neighbors to a point x_{ijk}, and analogously for higher dimensions. An example is shown in Figure 8.1. Given the discretized Laplace operator $\mathscr{L}$, we address the discretization of the Poisson's differential equation $\Delta\varphi = \mathfrak{f}$ on D for a given function $\mathfrak{f}$ defined on the interior of D. Calling f the discretization of $\mathfrak{f}$ on Ω_h, then the equation can be discretized and translated into $N = n^2$ (for a two-dimensional domain) or $N = n^3$ (in three dimensions) equations, one for each point in the grid, in the form

$$4\phi_{ij} - \phi_{(i+1)j} - \phi_{(i-1)j} - \phi_{i(j+1)} - \phi_{i(j-1)} = h^2 f_{ij} \qquad 1 \leq i, j \leq n$$

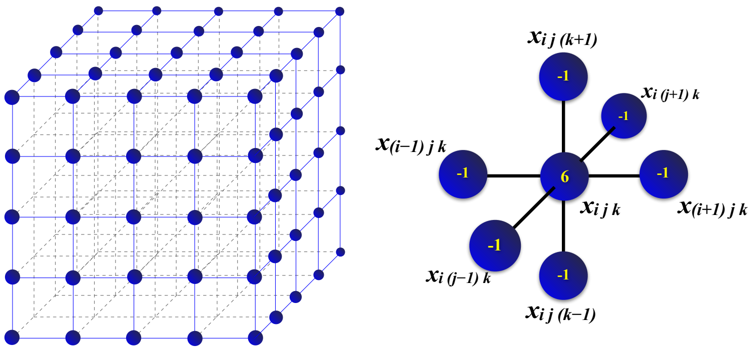

Figure 8.1. Discrete three-dimensional grid with $5 \times 5 \times 5$ points (left). Weighted three-dimensional stencil example: the Laplace operator (right).

for $f_{ij} := f(x_{ij})$. All these equations can be rearranged in form of a linear system reordering the N unknowns in a single vector containing the unknowns in the natural ordering $u = (\phi_{11}, \phi_{21}, \dots, \phi_{(n-1)n}, \phi_{nn})$ and assembling the corresponding coefficient matrix (see [Saad, 2003] for further details).

Assuming that L is the coefficient matrix coming from the discretized Poisson equation, in the electronic nanodevice simulation environment, the matrix (8.2) shares the structure with L and, in addition, presents the contribution of the OBCs defined in matrices Σ^L and Σ^R [Calderara, 2016]. The final nonzero pattern of A is then

$$\mathcal{N}(A) = \mathcal{N}(L) + \mathcal{N}(\Sigma) \quad \text{where} \quad \Sigma = \begin{pmatrix} \Sigma^L & & & \\ & & & \\ & & & \\ & & & \Sigma^R \end{pmatrix}. \tag{8.6}$$

Notice that Σ has only nonzero entires in the red blocks Σ^L and Σ^R, who are the boundary self-energy matrices and include the open boundary conditions. See a 2D example in Figure 8.2. In the NEGF framework introduced before, the diagonal of the inverse of matrix A in (8.6) must be computed up to some level of accuracy which is, in general, larger than machine precision; under this particular circumstance, the usage of a stochastic estimator for the diagonal of the inverse can be preferable among the suitable selected inversion techniques.

In the following section, we introduce the stochastic estimator based on the sampling of the diagonal that can be successfully used to estimate the entries of

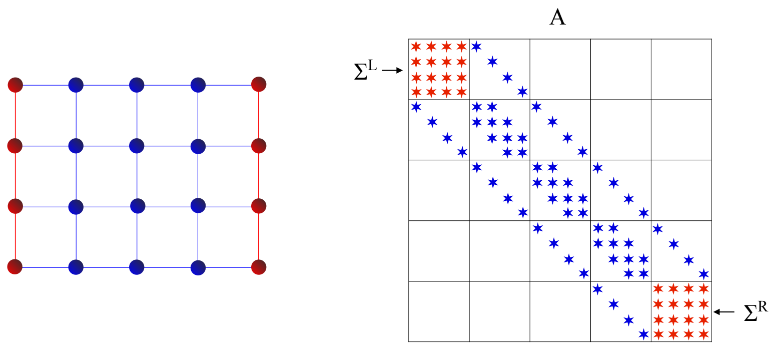

Figure 8.2. Left: discrete two-dimensional 4×5 rectangular domain with boundary conditions (in red). Right: resulting matrix, presenting the two blocks Σ^L and Σ^R in red.

the retarded Green's function $G^R(E)$ introduced in (8.1). As an additional characteristic, we are going to evaluate the diagonal of the inverse in a matrix-free framework: the matrix-vector operations involving A will be performed using a stencil operator via a function $\mathscr{A}$, formulated according to (3.19). From (8.6), we extend the stencil $\mathscr{L}$, to include also the action of the Σ's, and we call it $\widetilde{\mathscr{L}}$. We do this for computational purpose, exploiting the additivity of the two nonzero patterns. The final expression of $\mathscr{A}$ is then

$$\mathscr{A} = r \circ \widetilde{\mathscr{L}} \circ r^{-1},$$

where r is the reshape operator (3.18). The formulation of the implicit matrix stochastic estimation algorithm reflect then Algorithm 4, chapter 3, where the Krylov-subspace method used is the stencil-based conjugate gradient (Algorithm 5) and $f_A = \mathscr{A}$.

8.3 Stencil-based, matrix-free Krylov-subspace kernel for the NEGF model

We analyze now the implementation of the stochastic estimation algorithm for the retarded Green's function in a parallel, distributed memory C++ framework, called SEDI (stochastic estimation of the diagonal of the inverse). In the rest of the section we describe and test two different implementations: the first is based on a pure MPI approach [Snir et al., 1998], while the second includes the support of GridTools [Fuhrer et al., 2014], a library—introduced later—optimized for taking advantage of efficient caching strategies in order to increase the arithmetic intensity of the algorithm. Both the implementations present the implicit-matrix implementation of a scalable conjugate gradient algorithm with a simple Jacobi preconditioner (the cheapest preconditioner to be implemented when working on an implicit matrix framework, requiring simply the scaling of the stencil coefficients). The scalable matrix-free Krylov-subspace implementation is based on a domain splitting technique allowing the distributed computation of the stencils, making different MPI tasks evaluate the stencils on different blocks of the 3D grid, and build the final result through communication routines.

Introduction to GridTools

GridTools is a C++ template library providing a set of tools for PDE solvers on grids [Gysi et al., 2015] and is very versatile, since it can be used in several applications. Despite the apparent simplicity of stencil computations and the fact that current hardware architectures are ideal for mapping the computation structure, a programmer might still need to perform some fine-tuning on the computation paradigm in order to exploit the full computational power of the platform. The GridTools library aims at raising the level of abstraction of stencil computations, which allows application programmers to concentrate on problem specific aspects instead of low-level details that are relevant for performance. By doing so, the programmer is required to specify only the high-level application model and provide the stencil operators. This should help to detach the model developer from the implementation details, providing high-level abstraction and tools to translate stencil computations into a set of operators to be applied to the data. As an example, we show the kernel of the GridTools implementation of the 7-points 3D Laplacian operator in Figure 8.3. In order to simplify the abstraction from the user's point of view, the user specifies stencil operations at a high level of abstraction, leaving to the library's developers the burden of providing multiple hardware-specific implementations of different code backends. Additionally, the computation of stencils is tightly related to the prior communication of the different domain's neighboring points (halo update) so GridTools integrates a modified version of GCL (generic communication layer) [Bianco and Varetto, 2012]. As a last, important characteristics, GridTools applies a cache blocking strategy to achieve data reuse and minimize data transfers from main memory, maximizing data locality. This feature provides better performance when stencils with different shapes are grouped together.

```
struct d3point7{
    using out = accessor<0, inout>;
    using in  = accessor<1, in, extent<-1,1,-1,1,-1,1> >;

    template <typename Evaluator> GT_FUNCTION
    static void Do(Evaluatior eval) {
        eval(out{}) = 6.0 * eval(in{})
                    - (eval(in{-1,0,0}) + eval(in{+1,0,0}))
                    - (eval(in{0,-1,0}) + eval(in{0,+1,0}))
                    - (eval(in{0,0,-1}) + eval(in{0,0,+1}));
    }
};
```

Figure 8.3. Seven-points stencil for the 3D Laplace operator in GridTools.

8.3.1 The scalable SEDI implementation

Both the implementations of SEDI are based on the division of the discrete grid Ω_h into subdomains, one for each MPI task, where each subdomain computes its local part of the solution. The subdomains are computed by splitting Ω_h into parallelepiped, slicing it using orthogonal planes. When a task starts the computation of the stencil on the points belonging to the bordering faces of its own region, it needs to know the values on the adjacent face (if it exists), who will be owned by another MPI task. This communication of the boundary values, commonly referred to as "halo layer", could cause degrading of the performance unless the computation and the communication are somehow overlapped.

At each iteration of the stochastic estimator, the components of the sample are either generated by the master process and sent to the others via MPI send/receive calls (as in the GridTools implementation) or generated locally by the different processes (as the pure MPI code does). In the second case, however, the separate generation of the values on different nodes may not guarantee the independence of the numbers, introducing bias empirically detected. For this reason, the MPI implementation uses dedicated libraries for the generation of random numbers, guaranteeing to provide independent local parts across the nodes. Once that the local subdomains are filled with random values (the chunks of the vectors $\mathscr{V}_k$ in algorithm 4), the conjugate gradient can start. After the method has reached the convergence criterion, each of the MPI tasks will own the local part of $\mathscr{X}_k$, i.e., the sample required by the stochastic estimator to evaluate the diagonal of the inverse. At this point, each of the tasks can separately compute the Hadamard products of the local components of $\mathscr{X}_k$ and $\mathscr{V}_k$ (lines 6–7). When all the processes have completed this task, a new random right-hand side $\mathscr{V}_k$ is generated and the conjugate gradient solves the new system.

This process is iterated until all the samples are processed; after that, each task allocates a new local grid designated to host the estimate for the diagonal of the inverse and fills it with the Hadamard ratio in line 9 (local part of $\mathscr{D}$). As a last step, all the chunks of $\mathscr{D}$ are collected by an `MPI_Gather` call on the master process who assembles the complete grid, reshapes it in vector format, and writes it into a file.

The MPI implementation

The pure MPI implementation takes care of the domain decomposition by defining the scopes of the single subdomains and takes care of the halo communication by implementing an overlap of the communication and computation phases. In order to do so, the code integrates an asynchronous MPI communication scheme, by dividing the stencil operations into three stages: `initiate`, `compute_local`, and `finalize`. In the first phase, the requests for asynchronous border layer send/receive are posted. The compute phase performs a stencil operation on the inner cells of the local domain, while the last stage waits for all the communication routines to be completed and applies the stencil operator to the borders of the local domain. This concerns the strictly stencil-related operations. The boundary conditions are evaluated in the `finalize` stage, since their structure imposes a communication pattern within the two groups of processes that hold corresponding entries of the domain, i.e., the points lying on the two edges of the 3D domain. This requires the global MPI communicator to be split into subcommunicators including only the processes that need to communicate between each other, and allowing the use of collective MPI communication routines, namely, `MPI_Allgather`. In this way, the processes involved in the boundary conditions computation can perform the dense matrix-vector multiplication and add the result to the domain computed by the stencil operation.

The GridTools implementation

For creating the second implementation of SEDI, we have extended GridTools. The computational kernel of the conjugate gradient is still the same as the one in the pure MPI implementation, however we use the GridTools routines to perform the virtual matrix-vector operations. Here, dot products are also represented by a simple stencil kernel of elementwise multiplication followed by a global reduction performed over all local subdomains: this enables us to apply the GridTools library and its features in all the computational aspects of the conjugate gradient algorithms. The implementation is logically split into two sections: a prologue and an iteration section. In the prologue, stencil operators, iteration spaces, storages, and partitioning scheme are defined. The conjugate gradient steps are translated into different stencils, one for each of the operations needed in the algorithm, i.e., matrix-vector to compute the residuals, vector-vector (analogous to the BLAS's AXPY routine) to update the search directions and the residuals, and dot products to evaluate the scalar coefficients. In the iterative part, instead, the stencils are finally applied to the distributed domains defined in the prologue. Here, the explicit calls to the halo communication routines are performed as well as the communication related to the evaluation of the boundary conditions, who requires to split the MPI communicator since they are shared by a reduced set of processes.

8.4 Multinode scalability

In this section we present the performance of SEDI on a set of simulated NEFG matrices. We start by analyzing the scaling of the domain-splitting strategy on a multinode cluster, then we study the accuracy of the estimates. We do this in two steps: first, by evaluating the validity of the code by pushing the Krylov subspace method to operate on the diagonal of the inverse at a tolerance close to machine precision, and then, by measuring the mean error of the estimates for realistic tolerances of the conjugate gradient.

8.4.1 Benchmark setup

In order to evaluate the scalability of the code, we designed a dataset of matrices shaped as the Laplacian, but with additional boundary conditions, structured as in the NEGF problem in (8.6), section 8.2. The domain we considered is the unitary 3D cube partitioned into a regular 3D grid made by n^3 equidistant points for $n \in \{128, 256, 512, 1024\}$. We fixed the number of random samples for the estimator (s) to a common number for all the problems, as well as the number of iterations for the Krylov-subspace method, in order to force the code to perform all of them, independently on the numerical convergence. By doing so, we ensure the computational effort to be the same for the different implementations on each of the test cases even though the numerical results are not accurate enough (the accuracy will be evaluated later on). The nonzero pattern of the matrices is exactly the one coming from the nanoelectronic device simulation described in (8.6), section 8.2. The values of the nonzeros, however, reflect the ones of the Laplacian matrix L for the first part, while the two self-energy matrices (encoding the open boundary conditions) Σ^L and Σ^R are filled with fictitious values, preserving the symmetry of the stencils (hence, the stencils). This configuration provides for the difficulty encountered in retrieving real datasets, guaranteeing at the same time the equivalence in the performance analysis.

The discrete domain Ω_h is always partitioned into $p_1 \times p_2 \times p_3$ orthogonal parallelepipeds, where the product $p = p_1 p_2 p_3$ equals the number of MPI ranks. Once that $p \geq 1$ is fixed, the p_i's must be powers of two and are chosen to be "as balanced as possible," i.e., for $p = 1$ MPI rank we have $(p_1, p_2, p_3) = (1, 1, 1)$, and then the smallest number from the right is multiplied by 2 whenever the number of ranks is doubled. The next configurations are then $(1, 1, 2)$, $(1, 2, 2)$, $(2, 2, 2)$, $(2, 2, 4)$, and so on. The tests are performed on Piz Daint, a machine presenting 1431 multicore compute nodes, each of them equipped with two Intel Xeon E5-2695 v4 @ 2.10 GHz (2 x 18 cores, 64/128 GB RAM) located at the Swiss National Supercomputing Center. We have empirically observed that on this machine 4 MPI tasks per node can utilize the compute node while the scaling still holds. Increasing the task density per node, instead, the scaling starts to break down because the individual tasks compete for available node-level memory, degrading in turn the overall performance. This general setting is valid for both the MPI and the GridTools implementations.

8.4.2 Strong scaling results

As seen in Algorithm 3, chapter 3, the estimation of the diagonal of the inverse requires the generation of independent random vectors, the application of the iterative method, and the update of the solution found at the previous iteration. The solution of the linear systems with random right-hand sides is intuitively the the most expensive part from the computational point of view, therefore we focus the performance analysis on this area.

We report strong scaling performance of the MPI and GridTools implementations in Figures 8.4 and 8.5, respectively. We fixed 5 random samples and 50 CG iterations for this tests and report in the plots the time required by the individual components of the iterative algorithm: application of stencil operator and boundary conditions, dot products, and halo layers communication. Both the implementations perform reasonably with the smallest grid size used, 128^3, but only up to 64 MPI tasks. For such a grid size we notice that the dot products included in the conjugate gradient solver generate a volume of communication between the nodes that tends to break the scalability when the number of subdomains becomes large. The same behavior can be verified on the 256^3 grid at 256 tasks for the MPI code and at 128 tasks for GridTools. For larger domain sizes, instead, we observe nearly ideal linear scaling. Concerning the GridTools implementation, dot products appear to be the most significant parallelization bottleneck. The reason is that the library provides tools exclusively for local dot products, while it is optimized for multiple stencils to be fused together, requiring the user to complete the design of the communication pattern in order to achieve the global result.

In both implementations, apart from the dot products and stencil computations, the remaining operations, such as the handling of the OBCs with Σ^L and Σ^R, are orders of magnitude lower in terms of compute time, therefore, they are not critical for the overall performance. This is an extremely interesting result, since the presence of the open boundary conditions does not affect significantly the performance of the stencil code. In a matrix-based approach, on the other hand, the boundary conditions translate into dense submatrices, see (8.6), that could effect negatively the performance of a direct solver, while in both stencil implementations they do not worsen the performance. This embodies the main reason why we chose to explore the stochastic estimation process in NEGF-based problems.

8.5 Accuracy results

In this section, we validate the implementation and test its accuracy and performance with respect to the exact value of diag(A^{-1}) computed by the direct solver PARDISO. The structure of the matrix used for both accuracy and performance is the one introduced in section 8.4.1, but the size is sensibly smaller: the number of nonzeros in the LU factors computed by a direct method, in fact, reaches almost 290 times the nonzeros of A for $n = 256$, a number of entries that sparse direct solvers such as PARDISO cannot manage. We choose then $N = 16^3, 32^3, 64^3, 128^3$, and 256^3. For this kind of tests, of course, the bound on the fixed number of iterations set before in the performance is removed, so that the code is free to run up to convergence.

As a preliminary result, we compare the PARDISO and SEDI memory consumption together with the PARDISO relative fill-in in Table 8.1. We underline that the introduction of fill-in entries in the matrix factors caused by the LU-based approach in PARDISO generates a dramatic growth of the memory requirements. SEDI, on the other hand, thanks to its iterative nature, allows to save a considerable amount of memory, up to 99%.

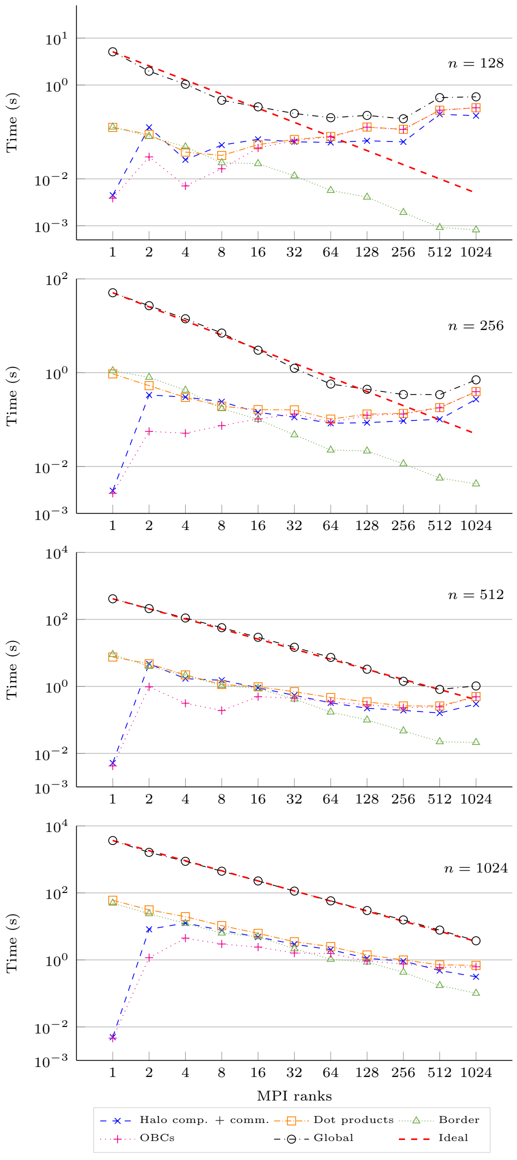

Figure 8.4. Strong scaling of MPI-SEDI for n^3 3D grid, 4 MPI tasks per node.

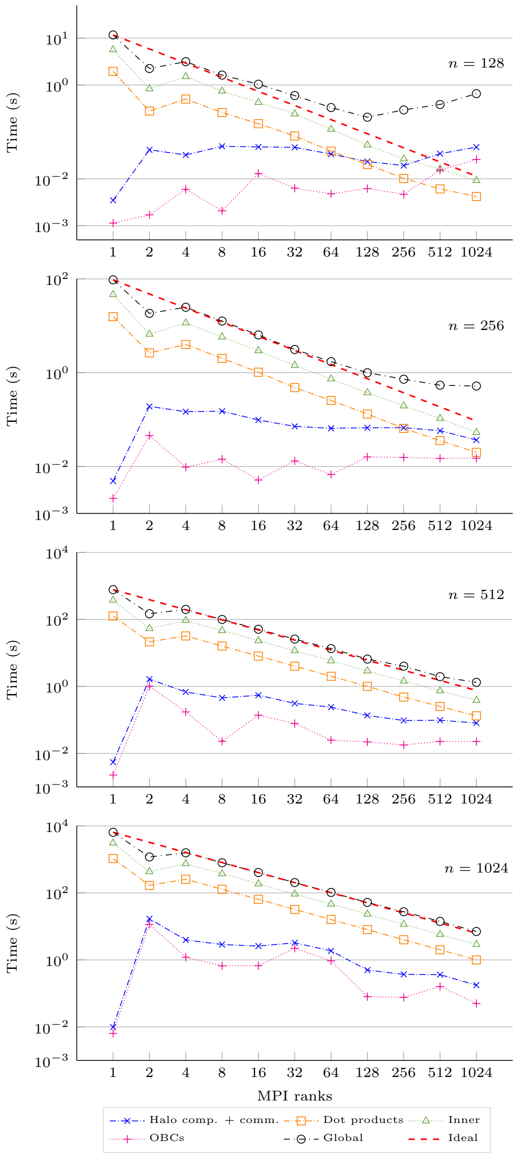

Figure 8.5. Strong scaling of GridTools-SEDI for n^3 3D grid, 4 MPI tasks per node.

8.5.1 Validity of the implementation

In order to validate the code, we prove the reliability of the computation of the diagonal entries in a deterministic setup. We force SEDI to compute the estimator on N right-hand sides consisting of the columns of the identity matrix, e_k for $k = 1, \ldots, N$, and compare the estimates with the diagonal computed

Table 8.1. PARDISO fill-in and memory usage for PARDISO and the MPI implementation of SEDI.

	Nonzeros		PARDISO fill-in			Memory [MB]		
n	$\frac{\mathrm{nnz}(A)}{10^6}$		$\frac{\mathrm{nnz}(L+U)}{10^6}$		$\frac{\mathrm{nnz}(L+U)}{\mathrm{nnz}(A)}$	PARDISO	SEDI	% saved
16	0.016	(0.094%)	0.3	(1.8%)	17.5	26	49	—
32	0.129	(0.012%)	6.3	(0.59%)	49.1	210	746	—
64	1.040	(0.0015%)	129.4	(0.19%)	124.4	1 130	1 058	6.8%
128	8.356	(0.00019%)	2 247.8	(0.051%)	269.7	17 200	1 420	91.7%
256	64.099	(0.000024%)	39 389.6	(0.014%)	614.5	312 000	1 907	99.4%

by PARDISO. As expected, we verify that the algorithm converges very quickly to the diagonal of the inverse varying the tolerance of the Krylov-subspace method ϵ (see Table 8.2). In particular, we see that $\epsilon = 10^{-6}$ grants a relative error already on the order of 10^{-12}, while $\epsilon = 10^{-8}$ is enough to provide almost the same solution as PARDISO up to machine precision. Additionally, decreasing ϵ to 10^{-10} does not produce substantial improvement any further. Although it plays a central role in the deterministic setup, in the following section, we will observe that big variations of the tolerance ϵ affect the relative error of the estimator in a negligible way.

8.5.2 Accuracy of the estimates

In order to measure the accuracy of SEDI, we again compare the error between the output of SEDI and the diagonal of the inverse computed by PARDISO, for different numbers of the samples s and tolerance values of the conjugate gradient, ϵ. The results, achieved for three different values of the grid size n, are reported in Table 8.3; the average error is computed as

$$\eta = \frac{1}{N}\sum_{j=1}^{N}\left|\frac{d_j - \widehat{d}_j}{d_j}\right|, \tag{8.7}$$

where d and $\widehat{d}$ are the exact (computed by PARDISO) and estimated (computed by SEDI) diagonals of the inverse, respectively; N is the order of the matrix ($N = n^3$).

Table 8.2. Relative error between the diagonals computed by PARDISO (called d) and the ones computed by the deterministic version of SEDI ($\widehat{d}$), considering the $N = n^3$ columns of the identity as samples. The error is computed as $\|d - \widehat{d}\|_2 / \|d\|_2$ and is reported as decreasing values of the tolerance of the conjugate gradient (ϵ).

	Tolerance ϵ				
n	10^{-2}	10^{-4}	10^{-6}	10^{-8}	10^{-10}
16	3.75e–04	1.83e–08	1.30e–12	6.86e–16	5.98e–16
32	7.10e–04	4.20e–08	2.89e–12	1.14e–15	9.64e–16
64	1.27e–03	9.85e–08	6.57e–12	1.99e–15	1.64e–15

8.6 Numerical results

We know that the error of the stochastic estimator depends on three parameters: the number s of random vectors considered, the accuracy of the iterative method adopted (ϵ), and the variance of the estimator (depending on the off-diagonal values of the inverse matrix). While other works focus on different heterogeneous types of matrix structures, as in [Bekas et al., 2009; Tang and Saad, 2012], the accuracy study presented in this section aims at the estimation of the average error for the estimator (3.15) on NEGF matrices. Notice that, for completeness, for the case $n = 16$ the estimates with 6400, 12800, and 25600 samples are reported, although the number of samples exceeds the order of the matrix ($16^3 = 4096$) and, therefore, also exceeds the number of right-hand sides needed by the application of a deterministic iterative method. For small values of the grid size, the memory and time requirements for a direct solver are more than affordable, so PARDISO remains the best approach; however, for sufficiently large grid size, i.e., for considerably larger matrix size, the stochastic estimator becomes a useful suitable choice in the NEGF framework, instead.

8.6.1 Independence from the Krylov-subspace tolerance

Table 8.3 shows how a decreasing tolerance for the Krylov-subspace method minimally affects the estimator, suggesting that the best practice to improve the estimates is to increase the number of samples. We also observe that the usage of a relatively small number of right-hand sides for the stochastic estimator

Table 8.3. SEDI mean relative error, $\eta = \frac{1}{N}\sum_j \left|(d_j - \widehat{d}_j)/d_j\right|$, as a function of the number of samples s and the tolerance ϵ. The underlined numbers represent the configurations achieving 95% accuracy.

Grid size n	16 ($N = 16^3$)		
Tolerance ϵ	10^{-2}	10^{-6}	10^{-10}
Samples	**Error** η		
400	$\underline{0.0457}$	$\underline{0.0461}$	$\underline{0.0461}$
800	0.0330	0.0334	0.0334
1600	0.0232	0.0234	0.0234
3200	0.0167	0.0168	0.0168
6400	0.0116	0.0116	0.0116
12800	0.0083	0.0084	0.0084
25600	0.0058	0.0060	0.0059
	32 ($N = 32^3$)		
400	0.0658	0.0677	0.0677
800	$\underline{0.0463}$	$\underline{0.0476}$	$\underline{0.0476}$
1600	0.0327	0.0336	0.0336
3200	0.0230	0.0236	0.0236
6400	0.0162	0.0167	0.0167
12800	0.0115	0.0118	0.0118
25600	0.0082	0.0083	0.0084
	64 ($N = 64^3$)		
400	0.0902	0.0989	0.0989
800	0.0628	0.0683	0.0683
1600	$\underline{0.0438}$	$\underline{0.0476}$	$\underline{0.0476}$
3200	0.0308	0.0335	0.0335
6400	0.0218	0.0236	0.0236
12800	0.0155	0.0167	0.0167
25600	0.0110	0.0118	0.0118

can provide interesting accuracy results. For instance, for $n = 32$, the use of 800 samples, i.e., less than 3% of the size of the matrix, guarantees a mean relative error smaller than 5% (underlined in Tables 8.3 and 8.4). The same level of accuracy is ensured for $n = 16$ with 400 samples (ca. 10%) and $n = 64$ with 1600 samples (<1%), in a fashion that appears to decrease linearly with respect to the order of the grid n. A more detailed view of mean relative error η is presented in Figure 8.6: the accuracy presents an initial exponential behavior as a function of the random samples. The strong dependency of the accuracy on the number of samples reveals an interesting characteristic of SEDI and suggests its employment in a parallel multinode framework to best exploit its potential.

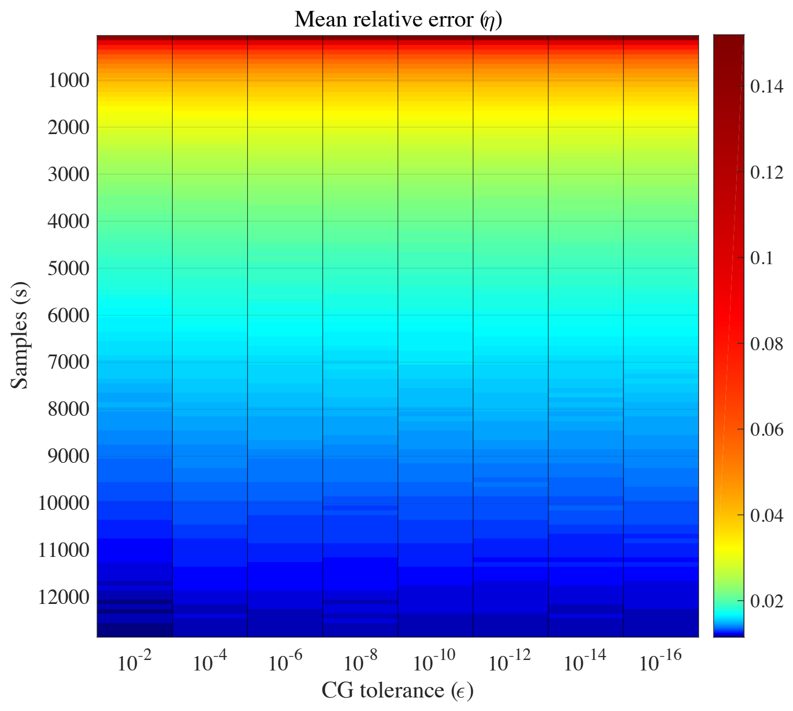

Figure 8.6. Mean relative error (η) of MPI-SEDI on the $32 \times 32 \times 32$ grid as a function of the number of samples s and the tolerance of the CG method ϵ, for $s = 100, 200, \ldots, 12800$ and $\epsilon = 10^{-2}, 10^{-4}, \ldots, 10^{-16}$

8.6.2 Overall time consumption and scalability with respect to the samples

The overall time required by the pure MPI implementation of SEDI is reported in Table 8.4 and it is compared with PARDISO. Both the solvers are required to perform a complete selected inversion process for the evaluation of the diagonal of the inverse (for PARDISO, this includes symbolic factorization, factorization, and evaluation of the inverse entries). We report then the results for a sequential

Table 8.4. Time consumption for single-threaded PARDISO and SEDI on a single task for fixed tolerance $\epsilon = 10^{-2}$. The underlined numbers represent the configurations achieving 95% accuracy (cf. Table 8.3).

n	16	32	64	128	256
	PARDISO				
	1.72e–01	1.33e+00	5.09e+01	5.57e+03	OOM*
samples	SEDI				
400	$\underline{\text{2.24e–01}}$	2.56e+00	2.30e+01	3.81e+02	4.39e+03
800	4.58e–01	$\underline{\text{4.90e+00}}$	4.59e+01	7.39e+02	8.73e+03
1600	9.15e–01	9.86e+00	$\underline{\text{9.21e+01}}$	1.49e+03	1.74e+04
3200	1.84e+00	1.92e+01	1.83e+02	3.00e+03	3.45e+04
6400	3.66e+00	3.88e+01	3.67e+02	5.97e+03	6.82e+04
12800	7.31e+00	7.71e+01	7.29e+02	1.19e+04	1.36e+05†

*: Out-of-memory.

†: Estimated time (execution time exceeds the allocation limits on Piz Daint).

version of PARDISO, i.e., using a single OpenMP thread, and MPI-SEDI on a single MPI rank. The matrices used and the hardware involved are the ones introduced before in section 8.4.1.

Depending on the level of accuracy requested, SEDI is extremely competitive and can easily replace a direct solver. Recalling the example introduced in the previous section, in fact, we focus again on a 95% accuracy: looking at the underlined cells in Table 8.4, we can see how the stochastic estimation framework demands an amount of time in the same order of magnitude as the time consumed by PARDISO on large enough cases. For instance, on the 32^3 grid the stochastic estimator is roughly 4 times slower, while it is roughly 2 times slower for the 64^3 one. Assuming a "natural" behavior of the accuracy for larger grid sizes, we speculate that, for the 128^3 grid, the 95% accuracy threshold could be reached for 3200 samples, i.e., in about $\frac{1}{2}$ of the time needed by PARDISO.

Finally, We observe a nearly perfect linear dependence between the computation time and the number of samples. In the previous sections, we analyzed the scalability of the code with respect to the domain splitting, now it is reasonable to assume a scalable behavior of the stochastic estimator with respect to the random right-hand sides provided they are independent and identically distributed. Looking again at Table 8.4 for a fixed value of n, in fact, we can see that doubling the number of samples roughly doubles the execution time (the multiplicative factor is 1.99, on average). This mainly comes from the fact that each of the right-hand sides is randomly generated and the conjugate gradient takes roughly the same number of iterations to converge to each solution vector, on average. Concerning the case $n = 256$, the time consumed by PARDISO cannot be reported due to an out-of-memory error (OOM), since the solver requires more than the 120 GB available on the Piz Daint nodes (see Table 8.1).

Chapter 9

Conclusions and outlook

In this work we have treated the selected inversion problem and interpreted it from the point of view of different applications, proposing large-scale algorithms for the evaluation of the diagonal of the inverse of sparse matrices.

We can ideally split the work into two parts, orthogonal to the chapters, given the different nature of the arguments treated. In the first part of this work, we approached the problem using a direct solver, based on the LU factorization of the matrix and the Takahashi's recursive formula. Keeping in mind the field of application (genetics) and the structure of the mathematical models it requires, we coupled the sparse formula with a distributed algorithm for the computation of the Schur-complement. The combination of the two leads to the formulation of a sparse/dense approach to the evaluation of the diagonal of the inverse suitable for any dense matrix presenting a large, sparse block. The performance of the algorithm have been enhanced following two lines of action: (i) the use of the block-wise recursive Takahashi's formula to achieve the best performance for the dense part, i.e., where the BLAS, PBLAS, and LAPACK libraries are used; (ii) the formulation of the parallel Schur-complement computation, allowing the distributed evaluation of the Schur-complement block from different MPI tasks; (iii) the memory- and computation-oriented optimization of resources allowing the software to reduce to the minimum redundant computations, reusing the LU factorization of the sparse blocks computed by PARDISO in different parts of the code (selected inverse, solution of systems, computation of the Schur-complement). These aspects completed the design of the large-scale, scalable, sparse/dense framework for selected inversion (section 2.7). We analyzed the scaling performance of the solver on sparse/dense datasets (chapter 6) on the Swiss National Supercomputing Center's (CSCS) Cray cluster, up to 400 nodes. The weak scaling tests revealed an almost ideal scaling for the distributed parallel Schur-complement evaluation, keeping an optimal load balance on the different nodes. The strong scaling tests, on the other hand, showed a general worsening of the performance due to the communication routines and the one-dimensional distribution of the data, however good enough compared to the direct solution through a sparse solver.

The parallel Takahashi's scheme have been successfully tested in the AI-REML framework designed for the estimation of parameters in the Gaussian regression process designed for genomic prediction problems. The sparse/dense approach was tailored to fit the requirements of the maximum-likelihood optimization algorithm for the prediction of fixed and random environment effects on SNP markers of plant breeds: the inclusion of marker-by-environment effects into the datasets introduces a rather high number of equations into the linear mixed models, leading towards large-scale problems. Fortunately enough, the inclusion of these effects increases the sparsity of the coefficient matrix, hence we treat the evaluation of the trace of its inverse, required to compute the gradient of the likelihood function to be maximized, exploiting the sparse/dense potential of our frameworks.

In the second part, instead, we studied how to evaluate the diagonal of the inverse of a large, sparse matrix using a stochastic approach. We analyzed the design of the Hutchinson's estimator for the trace of a matrix and a state-of-the-art strategy to extend it to the evaluation of the trace of the inverse. The combination of the estimator and the Monte Carlo-like method for the sampling of the action of the inverse have been combined in a single framework, always keeping in mind the application perspective. Considering the NEGF in quantum device simulations, we have formulated a parallel, scalable stochastic estimation scheme to be suitable for a set of large-scale simulations. The presence of the open boundary conditions has been considered an obstacle for state-of-the-art solvers in the treatment of three-dimensional nanotransistors, causing computational issues and sometimes prohibitive execution times, memory limitation problems, and expensive simulations. We studied then a parallel matrix-free version of the problem (SEDI: stochastic estimation of the diagonal of the inverse) based on a Krylov-subspace solver, able to overcome the limitations given by the boundary conditions and allowing an agile treatment of large-scale datasets.

In chapter 8, we have established the scaling performance and accuracy of the stencil-based SEDI on a set of simulated data. The stencil-like structure of the NEGF matrices inspired a stencil-based formulation of the conjugate gradient method for the solution of linear systems, where the matrix-vector and vector-vector operations are expressed by applying appropriate stencils to a grid of numerical values. The tests made on SEDI and the study of its accuracy in function of the number of random samples required to estimate the diagonal revealed good results, achieving a 95% accuracy by solving a number of linear systems in the order of 10% of the matrix size. We also analyzed the parallelization of two different implementations, based on a domain-splitting technique, revealing close to ideal scaling for increasing values of the matrix size on up to 1024 MPI tasks on 256 nodes the CSCS Cray supercomputers. We compared the SEDI's time requirements with PARDISO, proving that the execution time for SEDI can be kept in the order of magnitude of PARDISO's time for the 95% accuracy case. Finally, we proved that our framework grants a memory reduction up to 99.4% (compared with PARDISO), allowing the treatment of three-dimensional NEGF problems on structured grids presenting up to billions of grid points (1024^3 and more), avoiding the memory bottlenecks that direct LU-based solvers may encounter during the factorization phase. The solid scaling performance, the ease of parallelization, and the small memory requirements make SEDI a valid alternative to the direct solvers and is sometimes the only one suitable.

In conclusion, in this work we have provided two novel, application-oriented strategies for solving heterogeneous selected inversion problems by combining together state-of-the-art techniques for the treatment of sparse matrices, and extending them for approaching large-scale data. Both the parallel Takahashi's algorithm and the parallel stochastic estimator for the diagonal of the inverse contribute to the field of large-scale selected inversion.

In light of the results, we can focus the attention of the following main topics to be further analyzed to extend them.

First, considering the parallel Takahashi's performance, we may speculate the possibility to enhance the scaling by reducing the communication between nodes. The algorithm might benefit from the introduction of communication-avoiding factorization in the parallel treatment of the Schur-complement. Additionally, a general redesign of the communication pattern among the processes may be useful to increase the scaling factors, so far limited by the ScaLAPACK performance.

Second, the considerations made while designing the stochastic estimator for the diagonal of the inverse can be investigated in order to extrapolate heuristics for optimizing the execution time. For a given, required accuracy of the estimates, the study of the spectral properties of the NEGF matrices, and other stencil-structured ones, might suggest an optimal number of samples and tolerance for the Krylov-subspace method in order to achieve said accuracy. Second, a new framework for the parallel computation of several samples might be designed for exploit the scalability with respect to the samples.

Bibliography

Aguilar, I., Misztal, I., Johnson, D., Legarra, A., Tsuruta, S. and Lawlor, T. [2010]. Hot topic: A unified approach to utilize phenotypic, full pedigree, and genomic information for genetic evaluation of holstein final score1, Journal of Dairy Science **93**(2): 743 – 752.
URL: *http://www.sciencedirect.com/science/article/pii/S0022030210715174*

Amestoy, P., Duff, I., L'Excellent, J. and Koster, J. [2001]. A fully asynchronous multifrontal solver using distributed dynamic scheduling, SIAM Journal on Matrix Analysis and Applications **23**(1): 15–41.
URL: *https://doi.org/10.1137/S0895479899358194*

Amestoy, P., Duff, I., L'Excellent, J., Robert, Y., Rouet, F. and Uçar, B. [2012]. On computing inverse entries of a sparse matrix in an out-of-core environment, SIAM Journal on Scientific Computing **34**(4): A1975–A1999.
URL: *https://doi.org/10.1137/100799411*

Amestoy, P., Duff, I., L'Excellent, J. and Rouet, F. [2015]. Parallel computation of entries of A^{-1}, SIAM Journal on Scientific Computing **37**(2): C268–C284.
URL: *https://doi.org/10.1137/120902616*

Amestoy, P. R., Davis, T. A. and Duff, I. S. [1996]. An approximate minimum degree ordering algorithm, SIAM J. Matrix Anal. Appl. **17**(4): 886–905.
URL: *https://doi.org/10.1137/S0895479894278952*

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A. and Sorensen, D. [1999]. LAPACK Users' Guide, 3rd edn, Society for Industrial and Applied Mathematics, Philadelphia, PA.

Bank, R., Rose, D. and Fichtner, W. [1983]. Numerical methods for semiconductor device simulation, SIAM Journal on Scientific and Statistical Computing

4(3): 416–435.
URL: *https://doi.org/10.1137/0904032*

Bekas, C., Curioni, A. and Fedulova, I. [2009]. Low cost high performance uncertainty quantification, Proceedings of the 2Nd Workshop on High Performance Computational Finance, WHPCF '09, ACM, New York, NY, USA, pp. 8:1–8:8.
URL: *http://doi.acm.org/10.1145/1645413.1645421*

Bekas, C., Curioni, A. and Fedulova, I. [2012]. Low-cost data uncertainty quantification, Concurrency and Computation: Practice and Experience **24**(8): 908–920.
URL: *http://dx.doi.org/10.1002/cpe.1770*

Bekas, C., Kokiopoulou, E. and Saad, Y. [2007]. An estimator for the diagonal of a matrix, Applied Numerical Mathematics **57**(11-12): 1214–1229.
URL: *http://dx.doi.org/10.1016/j.apnum.2007.01.003*

Ben-Israel, A. [1965]. An iterative method for computing the generalized inverse of an arbitrary matrix, Mathematics of Computation **19**(91): 452–455.
URL: *http://www.jstor.org/stable/2003676*

Benzi, M., Golub, G. H. and Liesen, J. [2005]. Numerical solution of saddle point problems, Acta Numerica **14**: 1–137.
URL: *https://doi.org/10.1017/S0962492904000212*

Bianco, M. and Varetto, U. [2012]. A generic library for stencil computations, CoRR **abs/1207.1746**.
URL: *http://arxiv.org/abs/1207.1746*

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R. C. [1997]. ScaLAPACK Users' Guide, Society for Industrial and Applied Mathematics.

Bollhöfer, M. and Schenk, O. [2006]. Combinatorial aspects in sparse elimination methods, GAMM-Mitteilungen **29**(2): 342–367.
URL: *https://onlinelibrary.wiley.com/doi/abs/10.1002/gamm.201490037*

Bollhöfer, M., Schenk, O. and Verbosio, F. [2019]. High performance block incomplete LU factorization, Submitted to ACM Transactions on Mathematical Software (under review) .

Bunch, J. and Kaufman, L. [1975]. Some Stable Methods for Calculating Inertia and Solving Symmetric Linear Systems, Department of Computer Science, University of Colorado.
URL: *https://books.google.com/books?id=conXtwAACAAJ*

Calderara, M. [2016]. SplitSolve, an Algorithm for Ab-Initio Quantum Transport Simulations, PhD thesis, Diss. Technische Wissenschaften ETH Zürich, Nr. 23566, 2016.
URL: *http://dx.doi.org/10.3929/ethz-a-010781848*

Calderara, M., Brück, S., Pedersen, A., Bani-Hashemian, M. H., VandeVondele, J. and Luisier, M. [2015]. Pushing back the limit of ab-initio quantum transport simulations on hybrid supercomputers, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, ACM, New York, NY, USA, pp. 3:1–3:12.
URL: *http://doi.acm.org/10.1145/2807591.2807673*

Choi, J., Dongarra, J., Ostrouchov, S., Petitet, A., Walker, D. W. and Whaley, R. C. [1996]. A proposal for a set of parallel basic linear algebra subprograms, Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, PARA '95, Springer-Verlag, London, UK, UK, pp. 107–114.

Chow, E. and Saad, Y. [1998]. Approximate inverse preconditioners via sparse-sparse iterations, SIAM Journal on Scientific Computing **19**(3): 995–1023.
URL: *https://doi.org/10.1137/S1064827594270415*

Cooper, M., Podlich, D. W. and Smith, O. S. [2005]. Gene-to-phenotype models and complex trait genetics, Australian Journal of Agricultural Research **56**(9): 895–918.
URL: *https://doi.org/10.1071/AR05154*

Crossa, J., Campos, G. d. l., Pérez, P., Gianola, D., Burgueño, J., Araus, J. L., Makumbi, D., Singh, R. P., Dreisigacker, S., Yan, J., Arief, V., Banziger, M. and Braun, H.-J. [2010]. Prediction of genetic values of quantitative traits in plant breeding using pedigree and molecular markers, Genetics **186**(2): 713–724.
URL: *http://www.genetics.org/content/186/2/713*

Cuthill, E. and McKee, J. [1969]. Reducing the bandwidth of sparse symmetric matrices, Proceedings of the 1969 24th National Conference, ACM '69, ACM, New York, NY, USA, pp. 157–172.
URL: *http://doi.acm.org/10.1145/800195.805928*

Davis, T. A. [2004]. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method, ACM Trans. Math. Softw. **30**(2): 165–195.
URL: *http://doi.acm.org/10.1145/992200.992205*

Davis, T. A. and Hu, Y. [2011]. The University of Florida sparse matrix collection, ACM Trans. Math. Softw. **38**(1): 1:1–1:25.
URL: *http://doi.acm.org/10.1145/2049662.2049663*

De Coninck, A., De Baets, B., Kourounis, D., Verbosio, F., Schenk, O., Maenhout, S. and Fostier, J. [2016]. Needles: Towards large-scale genomic prediction with marker-by-environment interaction, Genetics .
URL: *http://www.genetics.org/content/early/2016/02/29/genetics.115.179887*

De Coninck, A., Fostier, J., Maenhout, S. and De Baets, B. [2014]. DAIRRy-BLUP: A high-performance computing approach to genomic prediction, Genetics **197**(3): 813–822.
URL: *http://www.genetics.org/content/197/3/813*

De Coninck, A., Kourounis, D., Verbosio, F., Schenk, O., De Baets, B., Maenhout, S. and Fostier, J. [2015]. Towards parallel large-scale genomic prediction by coupling sparse and dense matrix algebra, 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015, pp. 747–750.
URL: *https://doi.org/10.1109/PDP.2015.94*

Demmel, J., Eisenstat, S., Gilbert, J., Li, X. and Liu, J. [1999]. A supernodal approach to sparse partial pivoting, SIAM Journal on Matrix Analysis and Applications **20**(3): 720–755.
URL: *https://doi.org/10.1137/S0895479895291765*

Duff, I. S., Erisman, A. M., Gear, C. W. and Reid, J. K. [1988]. Sparsity structure and gaussian elimination, SIGNUM Newsl. **23**(2): 2–8.
URL: *http://doi.acm.org/10.1145/47917.47918*

Fuechsle, M., Miwa, J. A., Mahapatra, S., Ryu, H., Lee, S., Warschkow, O., Hollenberg, L. C. L., Klimeck, G. and Simmons, M. Y. [2012]. A single-atom transistor, Nat Nano **7**(4): 242–246.
URL: *http://dx.doi.org/10.1038/nnano.2012.21*

Fuhrer, O., Bianco, M., Bey, I. and Schär, C. [2014]. Grid Tools project description, `http://www.pasc-ch.org/projects/projects/grid-tools`.

George, A. [1973]. Nested dissection of a regular finite element mesh, SIAM Journal on Numerical Analysis **10**(2): 345–363.
URL: *https://doi.org/10.1137/0710032*

Gilmour, A. R., Thompson, R. and Cullis, B. R. [1995]. Average information REML: an efficient algorithm for variance parameter estimation in linear mixed models, Biometrics pp. 1440–1450.
URL: *https://www.jstor.org/stable/2533274*

Golub, G. H. and Van Loan, C. F. [2013]. Matrix computations, Vol. 3, Johns Hopkins University Press.

Grigori, L., Demmel, J. W. and Xiang, H. [2011]. Calu: A communication optimal lu factorization algorithm, SIAM J. Matrix Anal. Appl. **32**(4): 1317–1350.
URL: *http://dx.doi.org/10.1137/100788926*

Gysi, T., Osuna, C., Fuhrer, O., Bianco, M. and Schulthess, T. C. [2015]. Stella: A domain-specific tool for structured grid methods in weather and climate models, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, ACM, New York, NY, USA, pp. 41:1–41:12.
URL: *http://doi.acm.org/10.1145/2807591.2807627*

Hayes, B., Bowman, P., Chamberlain, A. and Goddard, M. [2009]. Invited review: Genomic selection in dairy cattle: Progress and challenges, Journal of Dairy Science **92**(2): 433 – 443.
URL: *https://dx.doi.org/10.3168/jds.2008-1646*

Henderson, C. R. [1963]. Selection index and expected genetic advance, Statistical genetics and plant breeding **982**: 141–163.
URL: *https://www.nap.edu/catalog/20264/statistical-genetics-and-plant-breeding*

Hetmaniuk, U., Zhao, Y. and Anantram, M. [2013]. A nested dissection approach to modeling transport in nanodevices: Algorithms and applications, International Journal for Numerical Methods in Engineering **95**(7): 587–607.
URL: *https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.4518*

Higham, N. J. [1986]. Newton's method for the matrix square root, Math. Comput. **46**(174): 537–549.
URL: *http://dx.doi.org/10.2307/2007992*

Hutchinson, M. [1990]. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines, Communications in Statistics - Simulation and Computation **19**(2): 433–450.
URL: *https://doi.org/10.1080/03610919008812866*

Jacquelin, M., Lin, L., Jia, W., Zhao, Y. and Yang, C. [2018]. A left-looking selected inversion algorithm and task parallelism on shared memory systems, Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018, ACM, New York, NY, USA, pp. 54–63.
URL: *http://doi.acm.org/10.1145/3149457.3149472*

Jacquelin, M., Lin, L. and Yang, C. [2016]. PSelInv — A distributed memory parallel algorithm for selected inversion : the symmetric case, ACM Trans. Math. Softw. **43**(3): 21:1–21:28.
URL: *http://doi.acm.org/10.1145/2786977*

Karypis, G. and Kumar, V. [1998]. A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. **20**(1): 359–392.

Karypis, G. and Kumar, V. [2009]. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0, `http://glaros.dtc.umn.edu/gkhome/views/metis`.

Kourounis, D., Fuchs, A. and Schenk, O. [2018]. Toward the next generation of multiperiod optimal power flow solvers, IEEE Transactions on Power Systems **33**(4): 4005–4014.
URL: *https://doi.org/10.1109/TPWRS.2017.2789187*

Kuzmin, A., Luisier, M. and Schenk, O. [2013]. Fast methods for computing selected elements of the Green's function in massively parallel nanoelectronic device simulations, Euro-Par 2013 Parallel Processing, Vol. 8097 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 533–544.
URL: *https://doi.org/10.1007/978-3-642-40047-6_54*

Lake, R., Klimeck, G., Bowen, R. C. and Jovanovic, D. [1997]. Single and multiband modeling of quantum electron transport through layered semiconductor devices, Journal of Applied Physics **81**(12): 7845–7869.
URL: *https://doi.org/10.1063/1.365394*

Lass, M., Mohr, S., Wiebeler, H., Kühne, T. D. and Plessl, C. [2018]. A massively parallel algorithm for the approximate calculation of inverse p-th roots of large sparse matrices, Proceedings of the Platform for Advanced Scientific

Computing Conference, PASC '18, ACM, New York, NY, USA, pp. 7:1–7:11.
URL: *http://doi.acm.org/10.1145/3218176.3218231*

Lawson, C. L., Hanson, R. J., Kincaid, D. R. and Krogh, F. T. [1979]. Basic linear algebra subprograms for fortran usage, ACM Trans. Math. Softw. **5**(3): 308–323.
URL: *http://doi.acm.org/10.1145/355841.355847*

Li, S., Ahmed, S., Klimeck, G. and Darve, E. [2008]. Computing entries of the inverse of a sparse matrix using the FIND algorithm, Journal of Computational Physics **227**(22): 9408 – 9427.
URL: *http://www.sciencedirect.com/science/article/pii/S0021999108003458*

Li, S. and Darve, E. [2012]. Extension and optimization of the FIND algorithm: Computing Green's and less-than Green's functions, Journal of Computational Physics **231**(4): 1121 – 1139.
URL: *http://www.sciencedirect.com/science/article/pii/S002199911100338X*

Li, X. S. and Demmel, J. W. [2003]. SuperLU_dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Trans. Math. Softw. **29**(2): 110–140.
URL: *http://doi.acm.org/10.1145/779359.779361*

Lin, L., Chen, M., Yang, C. and He, L. [2013]. Accelerating atomic orbital-based electronic structure calculation via pole expansion and selected inversion, Journal of Physics: Condensed Matter **25**(29): 295501.
URL: *https://doi.org/10.1088/0953-8984/25/29/295501*

Lin, L., Lu, J., Ying, L., Car, R. and E, W. [2009]. Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems, Comm. Math. Sci. **7**: 755.
URL: *http://dx.doi.org/10.1088/0953-8984/25/29/295501*

Lin, L., Lu, J., Ying, L. and E, W. [2012]. Adaptive local basis set for Kohn-Sham density functional theory in a discontinuous Galerkin framework I: Total energy calculation, Journal of Computational Physics **231**(4): 2140 – 2154.
URL: *http://www.sciencedirect.com/science/article/pii/S0021999111006875*

Lin, L., Yang, C., Meza, J. C., Lu, J., Ying, L. and E, W. [2011]. SelInv—An algorithm for selected inversion of a sparse symmetric matrix, ACM Trans. Math. Softw. **37**(4): 40:1–40:19.
URL: *http://doi.acm.org/10.1145/1916461.1916464*

Liu, W. and Sherman, A. [1976]. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices, SIAM Journal on Numerical Analysis **13**(2): 198–213.
URL: *https://doi.org/10.1137/0713020*

Luisier, M., Boykin, T. B., Klimeck, G. and Fichtner, W. [2011]. Atomistic nanoelectronic device engineering with sustained performances up to 1.44 PFlop/s, 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–11.

Luisier, M., Schenk, A., Fichtner, W. and Klimeck, G. [2006]. Atomistic simulation of nanowires in the $sp^3d^5s^*$ tight-binding formalism: From boundary conditions to strain calculations, Phys. Rev. B **74**: 205323.
URL: *http://link.aps.org/doi/10.1103/PhysRevB.74.205323*

Meuwissen, T. H. E., Hayes, B. J. and Goddard, M. E. [2001]. Prediction of total genetic value using genome-wide dense marker maps, Genetics **157**(4): 1819–1829.
URL: *http://www.genetics.org/content/157/4/1819*

Meyerhenke, H., Sanders, P. and Schulz, C. [2015]. Parallel graph partitioning for complex networks, Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS '15, IEEE Computer Society, Washington, DC, USA, pp. 1055–1064.
URL: *http://dx.doi.org/10.1109/IPDPS.2015.18*

Paige, C. and Saunders, M. [1975]. Solution of sparse indefinite systems of linear equations, SIAM Journal on Numerical Analysis **12**(4): 617–629.
URL: *https://doi.org/10.1137/0712047*

Pan, V. and Reif, J. [1989]. Fast and efficient parallel solution of dense linear systems, Computers & Mathematics with Applications **17**(11): 1481 – 1491.
URL: *http://www.sciencedirect.com/science/article/pii/0898122189900813*

Pan, V. and Schreiber, R. [1991]. An improved newton iteration for the generalized inverse of a matrix, with applications, SIAM Journal on Scientific and Statistical Computing **12**(5): 1109–1130.
URL: *https://doi.org/10.1137/0912058*

Pan, V. Y., Barel, M. V., Wang, X. and Codevico, G. [2004]. Iterative inversion of structured matrices, Theoretical Computer Science **315**(2): 581 – 592. Alge-

braic and Numerical Algorithms.
URL: *http://www.sciencedirect.com/science/article/pii/S0304397504000404*

Petra, C. G., Schenk, O., Lubin, M. and Gärtner, K. [2014]. An augmented incomplete factorization approach for computing the Schur complement in stochastic optimization, SIAM J. Scientific Computing **36**(2).
URL: *https://doi.org/10.1137/130908737*

Petra, C., Schenk, O. and Anitescu, M. [2014]. Real-time stochastic optimization of complex energy systems on high-performance computers, Computing in Science Engineering **16**(5): 32–42.
URL: *https://doi.org/10.1109/MCSE.2014.53*

Piepho, H.-P. [2009]. Ridge regression and extensions for genomewide selection in maize, Crop Science - CROP SCI **49**.
URL: *https://dx.doi.org/10.2135/cropsci2008.10.0595*

Saad, Y. [2003]. Iterative methods for sparse linear systems, SIAM.

Sahasrabudhe, H., Fonseca, J. and Klimeck, G. [2014]. Accelerating nano-scale transistor innovation with NEMO 5 on Blue Waters.
URL: *https://www.semanticscholar.org/paper/Accelerating-Nano-scale-Transistor-Innovation-with-Sahasrabudhe-Fonseca/43a987f751a5cc2157bcb6658764f12ebd8001ea*

Schenk, O. [2000]. Scalable parallel sparse LU factorization methods on shared memory multiprocessors, PhD thesis, Diss. Technische Wissenschaften ETH Zürich, Nr. 13515, 2000.
URL: *http://dx.doi.org/10.3929/ethz-a-003876213*

Schenk, O. and Gärtner, K. [2002]. Two-level dynamic scheduling in pardiso: Improved scalability on shared memory multiprocessing systems, Parallel Computing **28**(2): 187 – 197.
URL: *http://www.sciencedirect.com/science/article/pii/S0167819101001351*

Schenk, O. and Gärtner, K. [2004]. Solving unsymmetric sparse systems of linear equations with PARDISO, Future Gener. Comput. Syst. **20**(3): 475–487.
URL: *http://dx.doi.org/10.1016/j.future.2003.07.011*

Schenk, O., Gärtner, K. and Fichtner, W. [1999]. Scalable parallel sparse factorization with left-right looking strategy on shared memory multiprocessors, in P. Sloot, M. Bubak, A. Hoekstra and B. Hertzberger (eds), High-Performance

Computing and Networking, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 221–230.
URL: *https://doi.org/10.1023/A:1022326604210*

Schenk, O., Gärtner, K., Fichtner, W. and Stricker, A. [2001]. PARDISO: A high-performance serial and parallel sparse linear solver in semiconductor device simulation, Future Gener. Comput. Syst. **18**(1): 69–78.
URL: *http://dx.doi.org/10.1016/S0167-739X(00)00076-5*

Searle, S., Casella, G. and McCulloch, C. [1992]. Variance Components, Wiley Series in Probability and Statistics, Wiley.
URL: *https://books.google.ch/books?id=CWcPAQAAMAAJ*

Searle, S. R. [1997]. The matrix handling of BLUE and BLUP in the mixed linear model, Linear Algebra and its Applications **264**: 291 – 311. Sixth Special Issue on Linear Algebra and Statistics.
URL: *http://www.sciencedirect.com/science/article/pii/S0024379596004004*

Simpson, T., Pasadakis, D., Kourounis, D., Fujita, K., Yamaguchi, T., Ichimura, T. and Schenk, O. [2018]. Balanced graph partition refinement using the graph p-Laplacian, Proceedings of the Platform for Advanced Scientific Computing Conference, PASC '18, ACM, New York, NY, USA, pp. 8:1–8:11.
URL: *http://doi.acm.org/10.1145/3218176.3218232*

Snir, M., Otto, S., Huss-Lederman, S., Walker, D. and Dongarra, J. [1998]. MPI-The Complete Reference, Volume 1: The MPI Core, 2nd. (revised) edn, MIT Press, Cambridge, MA, USA.

Söderström, T. and Stewart, G. [1974]. On the numerical properties of an iterative method for computing the Moore-Penrose generalized inverse, SIAM Journal on Numerical Analysis **11**(1): 61–74.
URL: *https://doi.org/10.1137/0711008*

Soler, J., Artacho, E., D Gale, J., Garcia, A., Junquera, J., Ordejón, P. and Sánchez-Portal, D. [2002]. The SIESTA method for ab initio order-N materials simulation, Journal of Physics: Condensed Matter **14**.
URL: *https://doi.org/10.1088/0953-8984/14/11/302*

Strandén, I. and Christensen, O. F. [2011]. Allele coding in genomic evaluation, Genet Sel Evol **43**(1): 25–25.
URL: *https://www.ncbi.nlm.nih.gov/pubmed/21703021*

Svizhenko, A., Anantram, M. P., Govindan, T. R., Biegel, B. and Venugopal, R. [2002]. Two-dimensional quantum mechanical modeling of nanotransistors, Journal of Applied Physics **91**(4): 2343–2354.
URL: *https://doi.org/10.1063/1.1432117*

Takahashi, K., Fagan, J. and Chin, M. [1973]. Formation of a sparse bus impedance matrix and its application to short circuit study, 8th Power Industry Computer Application Conference Proceedings p. 63.

Tang, J. M. and Saad, Y. [2012]. A probing method for computing the diagonal of a matrix inverse, Numerical Linear Algebra with Applications **19**(3): 485–501.
URL: *https://onlinelibrary.wiley.com/doi/abs/10.1002/nla.779*

VandeVondele, J., Krack, M., Mohamed, F., Parrinello, M., Chassaing, T. and Hutter, J. [2005]. Quickstep: Fast and accurate density functional calculations using a mixed gaussian and plane waves approach, Computer Physics Communications **167**(2): 103 – 128.
URL: *http://www.sciencedirect.com/science/article/pii/S0010465505000615*

Verbosio, F., Coninck, A. D., Kourounis, D. and Schenk, O. [2017]. Enhancing the scalability of selected inversion factorization algorithms in genomic prediction, Journal of Computational Science **22**: 99 – 108.
URL: *http://www.sciencedirect.com/science/article/pii/S1877750317301473*

Verbosio, F., Kardoš, J., Bianco, M. and Schenk, O. [2018]. Highly scalable stencil-based matrix-free stochastic estimator for the diagonal of the inverse, 30th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2018, Lyon, France, September 24-27, 2018.
URL: *https://doi.org/10.1109/CAHPC.2018.8645868*

Yannakakis, M. [1981]. Computing the minimum fill-in is NP-complete, SIAM Journal on Algebraic and Discrete Methods **2**.
URL: *https://doi.org/10.1137/0602010*

Zhao, Y., Hetmaniuk, U., Patil, S. R. and Qi, J.and Anantram, M. P. [2016]. Nested dissection solver for transport in 3d nano-electronic devices, Journal of Computational Electronics **15**(2): 708–720.
URL: *http://dx.doi.org/10.1007/s10825-015-0778-x*