MÉMOIRE

présenté pour obtenir le

MASTER RECHERCHE INFORMATIQUE
ET TÉLÉCOMMUNICATIONS DE L'INSTITUT
NATIONAL POLYTECHNIQUE DE TOULOUSE

Spécialité : SYSTÈMES INFORMATIQUES ET GÉNIE LOGICIEL

par

François-Henry Rouet

INPT-ENSEEIHT IRIT

Calcul partiel de l'inverse d'une matrice creuse de grande taille - application en astrophysique

—

Partial computation of the inverse of a large sparse matrix - application to astrophysics

---

Mémoire soutenu le 18 Septembre 2009 devant le jury :

Président du jury :	Patrick Amestoy	IRIT-ENSEEIHT
Membres du jury :	Alain-Christophe Bon	TOTAL
	Alfredo Buttari	IRIT-ENSEEIHT
	Emmanuel De Mones	PSA Peugeot Citroën
	Fabrice Evrard	IRIT-ENSEEIHT
	Barthélémy Marti	ATOS-ORIGIN
	Julien Morin	SUPRALOG
	Bernard Thiesse	IRIT-ENSEEIHT

Résumé

Nous nous intéressons à la résolution de systèmes linéaires creux de grande taille par des méthodes directes, et plus particulièrement à l'utilisation de ces méthodes dans le cadre du calcul d'un ensemble d'éléments de l'inverse d'une matrice creuse. Cette étude s'inscrit dans le cadre d'une collaboration avec des chercheurs du CESR (Centre d'Etudes Spatiales du Rayonnement, Toulouse), pour lesquels le calcul de la diagonale de l'inverse de matrices creuses (issues de problèmes statistiques) représente une portion critique d'un de leurs codes applicatifs.

Dans une première partie, nous présentons le problème du calcul d'un ensemble d'entrées de l'inverse d'une matrice creuse, les différentes solutions existantes, et les extensions qui ont été proposées pendant ce travail, principalement pour la résolution d'un problème combinatoire lié au regroupement de seconds membres creux.

Dans une seconde partie, nous présentons des résultats expérimentaux et des aspects liés à l'implémentation, notamment dans le cadre de la collaboration avec le CESR.

Mots-clés : matrices creuses, méthode multifrontale, permutations, moindres carrés.

Abstract

We consider the solution of large sparse linear systems with direct methods, and, more specifically, the use of these methods for the computation of a set of elements of the inverse of a sparse matrix. This study has been motivated by a collaboration with researchers from the CESR (Centre for the Study of Radiation in Space, Toulouse), who develop a numerical code for which the computation of the diagonal of the inverse of sparse matrices (from statistical problems) is critical.

In the first part of this dissertation, we present our problem (computation of a set of entries of the inverse of large sparse matrices), the different existing techniques, and the extensions which have been proposed throughout this study, especially for the solution of a combinatorial problem related to grouping right-hand sides.

We present in the second part the experimental study (tests, as well as implementation considerations), where we emphasize what has been done during our partnership with the CESR.

Keywords: sparse matrices, multifrontal method, permutations, least squares.

Acknowledgements

I want to thank sincerely my advisor, Patrick Amestoy, for his enthusiasm and his availability, which have put this study on the right track. I am also very grateful to Laurent Bouchet, our collaborator at CESR, for his help on the applicative and experimental parts of this project. I thank as well Bora Uçar, for his precious contribution on the combinatorial aspects of this work, and the fruitful discussions I have had with him. Finally, *"grazie mille!"* to Alfredo Buttari and Chiara Puglisi for their help, and whose colourful Italian language brightens up our laboratory! Having the opportunity to meet and work with all these people has magnified my motivation to do research.

Je tiens sincèrement à remercier mon superviseur, Patrick Amestoy, pour son enthousiasme et sa disponibilité, qui ont permis de mettre cette étude sur de bons rails. Je suis également très reconnaissant envers Laurent Bouchet, notre collaborateur au CESR, pour son aide sur les parties applicatives et expérimentales de ce projet. Je remercie aussi Bora Uçar pour son apport précieux sur les aspects combinatoires de ce travail et les discussions fructueuses que j'ai pues avoir avec lui. Enfin, *"grazie mille!"* à Alfredo Buttari et Chiara Puglisi pour leur aide, et dont l'italien fleuri égaye les couloirs de notre laboratoire ! Rencontrer et travailler avec toutes ces personnes n'a fait que renforcer ma motivation à faire la recherche.

Contents

Résumé **iii**

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**
1.1 General context of the study . . . 1
1.2 Collaboration with the CESR (Centre for the Study of Radiation in Space) . . . 1

I Computing entries of the inverse of a large sparse matrix **5**

2 Introduction to the field **7**
2.1 Direct methods . . . 7
2.2 Multifrontal methods . . . 9
2.3 Solver used . . . 10

3 Computing entries of the inverse **11**
3.1 A simple approach to compute the diagonal . . . 11
3.2 Computing an arbitrary set with Takahashi's equations . . . 12
3.3 Computing an arbitrary set with a traditional solution phase . . . 13
3.4 Comparison of the different methods . . . 15

4 On the influence of right-hand sides permutations **19**
4.1 Influence of orderings of right-hand sides . . . 19
4.2 A global approach: hypergraph model . . . 22
4.3 A constructive approach: structure of a solution . . . 26
4.4 A local approach: improvement of a given order . . . 33
4.5 Perspectives . . . 36

II Experimental study **39**

5 Context of the experimental study **41**
5.1 First experiments . . . 41
5.2 Simple improvements . . . 42

6 Computation of the diagonal of the inverse **45**
6.1 Experiments . . . 45
6.2 Results . . . 45

7 Influence of right-hand sides permutations **47**
7.1 Topological orderings . . . 47

7.2 Hypergraph model 48
7.3 Constructive heuristics 48

Conclusion **51**

A More details about elimination and assembly trees **53**
A.1 Construction of the elimination tree 53
A.2 Assembly trees 54
A.3 Unsymmetric case 55

B Computation of the diagonal of the inverse thanks to a factorization **57**
B.1 Left-looking vs. right-looking approaches 57
B.2 Implementation 58
B.3 Comparison 61

Bibliography **63**

Chapter 1

Introduction

1.1 General context of the study

Background

Solving linear systems is the keystone of numerous academic and industrial applications, especially physical simulations (e.g. fluid dynamics, structural mechanics, circuit design, ...). These systems tend to be large (commonly more than 100000 unknowns, and up to a few tens of millions nowadays) and sparse, which means than most of the entries (often more than 99 percent) of the matrix associated to the system are zeros; many computational methods have been and still are developed to handle such linear systems efficiently.

In our study, we focus on problems where the right-hand sides are sparse as well. This situation occurs in applications involving highly reducible matrices, or where one needs to compute the null-space of a deficient matrix, or entries in the inverse of a matrix; we focus on this latter case. Such problems arise, for example, in electromagnetics and data assimilation.

Contents

Throughout this study, we address the problem of computing a set of entries of the inverse of a large sparse matrix, with an applicative leitmotiv (collaboration with the CESR, Centre for the Study of Radiation in Space) described in the next section. In a first part of this dissertation, we present the theoretical aspects needed to understand the problem, as well as our contribution to extend previous work (mainly combinatorial aspects related to grouping right-hand sides). In a second part, we present the various experiments carried out during our study, and highlight what has been done in relation with the application from CESR.

1.2 Collaboration with the CESR (Centre for the Study of Radiation in Space)

Application context

One of the motivations for this study was a collaboration between the APO (Algorithmes Parallèles et Optimisation) team at IRIT (Institut de Recherche en Informatique de Toulouse) and the SPI/INTEGRAL team at CESR.

In the context of the INTEGRAL (INTErnational Gamma-Ray Astrophysics Laboratory) mission of ESA (European Space Agency) a spatial observatory with high resolution (both in terms of angle and energy) hardware technology was launched on October 2002. SPI is one of the main instruments aboard INTEGRAL, a spectrometer with high energy resolution and indirect imaging capabilities. To obtain a complete sky survey with SPI/INTEGRAL, processing a very large amount of data acquired by the INTEGRAL observatory is needed [11]. For example, to estimate

the total point-source emission contributions (i.e. the contribution of a set of sources to the observed field), a linear least-squares problem of about 1 million equations and 100000 unknowns must be solved.

Technical details

Here we describe formally the application developed by the researchers from the SPI team. In the physical problem, there are n_d detectors, n_p measures per detector (the p-th measure of the d-th detector is noted $D(d,p)$), and n_s sources of energy; the aim is to compute the intensity $I(s)$ of every source s according to the following statistical model :

$$D(d,p) = \sum_{s=1}^{n_s} I(s)R(d,p,s) + b(d,p)$$

where R is a transfer function and b a noise. This noise can be modeled by

$$b(d,p) = Z(p)P(d,p)$$

where Z (intensity of the noise for the p-th measure) and P (called the "empty field") are unknown. In order to preserve the linearity of the problem, we consider that P is known (hence only Z has to be determined): we start with an approximation P_0 and apply some iterative refinements (see below). Therefore, the problem is linear and can be written as:

$$\begin{bmatrix} P(1,1) & 0 & \dots & 0 & R(1,1,1) & \dots & R(1,1,n_s) \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ P(n_d,1) & 0 & \dots & 0 & R(n_d,1,1) & \dots & R(n_d,1,n_s) \\ 0 & P(1,2) & \dots & 0 & R(1,2,1) & \dots & R(1,2,n_s) \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & P(n_d,2) & \dots & 0 & R(n_d,2,1) & \dots & R(n_d,2,n_s) \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & P(1,n_p) & R(1,n_p,1) & \dots & R(1,n_p,n_s) \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & P(n_d,n_p) & R(n_d,n_p,1) & \dots & R(n_d,n_p,n_s) \end{bmatrix} \begin{bmatrix} Z(1) \\ \vdots \\ Z(n_p) \\ I(1) \\ \vdots \\ I(n_s) \end{bmatrix} = \begin{bmatrix} D(1,1) \\ \vdots \\ D(n_d,1) \\ D(1,2) \\ \vdots \\ D(n_d,2) \\ \vdots \\ D(1,n_p) \\ \vdots \\ D(n_d,n_p) \end{bmatrix}$$

We rewrite this system in the form $Bx = y$, where B is an $n_d \cdot n_p \cdot (n_s + n_d)$ matrix, and x and y vectors with respectively $n_p + n_s$ and $n_d \cdot n_p$ components. This system is *overdetermined* (there are more equations - or measures here - than unknowns), therefore there is (generally) no exact solution, but we can try to define a "best" solution [10]: a common choice, often motivated by statistical reasons (see below), is to solve the *least-squares problem*

$$\min_x ||Bx - y||$$

The solution of this problem can be computed by means of the *normal equations*:

$$x = A^{-1}z, \text{ with } A = B^T B, \text{ and } z = B^T y$$

In the CESR context, once this problem has been solved, a better approximation of P can be built (we do not provide details about this operation, but it is rather inexpensive), and the whole problem is solved in successive steps of an iterative process (solving the system[1], building a better approximation, and so on). Actually, the initial approximation of P is good, and a few iterations (10 is what is used in the code developed by the SPI team) are sufficient to converge.

[1]The update of P induces an update of B (n_p first columns), and hence of A as well.

Once these iterations have been performed, the *variances* of the components of the solution are needed as well. These variances can be obtained by computing the diagonal of A^{-1} (n.b.: in this dissertation, $a^{-1}_{i,j}$ refers to $(A^{-1})_{i,j}$):

$$Var(x_i) \propto a^{-1}_{i,i}$$

Finally, we sum up the algorithm developed by the SPI Team:

Algorithm 1.1 Solution of the least-squares problem and computation of the variances.

1: Compute $A = B^T B$.
2: **for** i=1 to 10 **do** {Iterative computation of P and x}
3: Solve $z = Ax$.
4: Compute a new approximation of P.
5: Update B.
6: Update A.
7: **end for**
8: Compute $diag(A^{-1})$ to have the variances.

What has to be kept in mind is that computing $diag(A^{-1})$ is (by far) the most expensive operation of this algorithm (see the different experiments in Chapter 5); this justifies the need for an efficient method for computing a set of entries of the inverse of a matrix. We describe in the next sections the existing methods and suggest some improvements.

Part I

Computing entries of the inverse of a large sparse matrix

Chapter 2

Introduction to the field

All the algorithms presented throughout this study rely on the use of direct methods for solving sparse linear systems; we thus first introduce such methods, then we address the problem of the computation of a set of elements of the inverse of a large, sparse matrix.

2.1 Direct methods

Direct methods for solving linear systems are commonly based on Gaussian elimination, where the aim is to *factorize* the matrix, say A, of a linear system $Ax = b$, into a product of "simpler" matrices (called *factors*), that can be used to solve the system. Such methods usually consist of three phases: *analysis*, *factorization*, and *solution*.

Analysis phase

The role of this first phase is to apply numerical and structural pretreatments in order to prepare the following phases. Numerical pretreatments are intended to avoid numerical problems during the factorization phase. *Scaling* is a typical example of such a processing: it consists in computing diagonal matrices D_r and D_c such that $D_r A D_c$ has good numerical properties.

Structural pretreatments aim mainly at reducing the *fill-in* (non-zeros elements which appear in the factors but do not exist in the initial matrix), which can induce a prohibitive memory consumption. They generally consist in a *reordering* of the rows and columns of the initial matrix; finding a reordering which induces a minimum fill-in is a NP-complete problem, and numerous heuristics have been studied to obtain efficient techniques. Reorderings also define thereafter the order in which the factorization will be performed (see below).

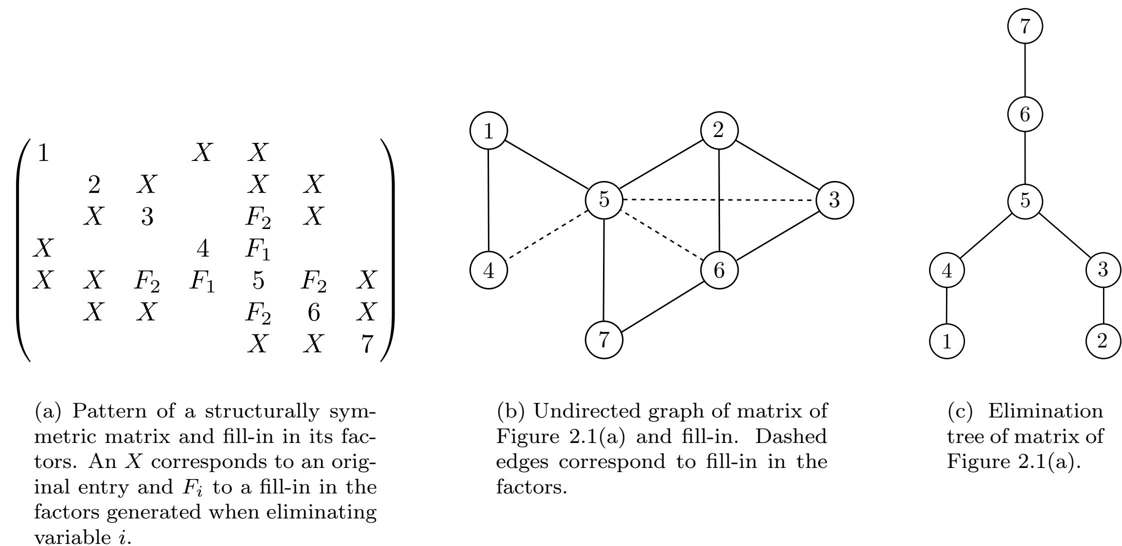
During the analysis phase, the *symbolic factorization* of the matrix, which computes the structure of the factors, is performed. The symbolic factorization is achieved by manipulating graphs associated to the matrix to factorize. Here, we consider only the case where the matrix is structurally symmetric: this means that its *pattern* (the set of non-zero elements) is symmetric, but the matrix is not necessarily symmetric numerically.

Definition 2.1 - *Undirected graph associated to a symmetric matrix.*
The undirected graph associated with a symmetric matrix A, $G(A) = (V, E)$, is defined as follows:

- V corresponds to the rows and columns of A, i.e. $V = [\![1, n]\!]$.
- E corresponds to the non-zero entries, i.e. there is a edge between two nodes, say i and j, if and only if $a_{i,j} \neq 0$.

Figure 2.1 shows an example of a structurally symmetric matrix (2.1(a)) and its associated graph (2.1(b)). This graph representation makes the description and implementation of the symbolic factorization simple; we apply the following process: each time a variable (an unknown of

the linear system) is eliminated, its adjacent nodes in $G(A)$ (variables not yet eliminated) are connected. The new edges that appear in $G(A)$ correspond to fill-in, i.e. non-zero elements which exist in the pattern of the factors but not in the pattern of the initial matrix. Figure 2.1(b) shows the edges that appear during the symbolic factorization: for example, the edges between nodes 3 and 5 and between nodes 5 and 6 appear when node 2 is eliminated, because 3,5 and 6 were adjacent to 2. This implies that elements (3,5), (5,3), (5,6) and (6,5) in the factors will be non-zeros. However, there was already an edge between node 2 and node 6: elements (2,6) and (6,2) in the factors will be non-zero, but they are not a fill-in elements, because $a_{2,6}$ and $a_{6,2}$ were already non-zeros.



(a) Pattern of a structurally symmetric matrix and fill-in in its factors. An X corresponds to an original entry and F_i to a fill-in in the factors generated when eliminating variable i.

(b) Undirected graph of matrix of Figure 2.1(a) and fill-in. Dashed edges correspond to fill-in in the factors.

(c) Elimination tree of matrix of Figure 2.1(a).

Figure 2.1: Example of symbolic factorization of a structurally symmetric matrix.

The undirected graph associated to the matrix also expresses dependencies between the variables: an edge between two nodes (two variables of the underlying system) implies that these variables cannot be eliminated independently. These dependencies can be represented by a compressed version of the initial graph called the *elimination tree*. For the sake of clearness, we provide details about the construction of elimination trees in Appendix A.1, as well as details about the unsymmetric case and the factorization. The elimination tree is processed from the leaves (1 and 2 in Figure 2.1(c)) to the root (node 6 in Figure 2.1(c)). This traversal represents the order in which the unknowns of the underlying linear system are eliminated: a variable cannot be eliminated before its descendants, but two variables without a descent relation (i.e. which are not descendant of one another) can be eliminated in any order, or at the same time. The elimination tree thus expresses independent tasks which can be computed in parallel.

Figure 2.1(c) shows the elimination tree of the matrix of Figure 2.1(a): variables 1 and 2 can be eliminated independently, as well as variables 3 and 4, but variable 1 has to be eliminated before variable 4, and so on.

Finally, in a parallel context, where several processors are used to achieve the whole computation, a mapping of the tasks on the different processors is computed, which can be constrained to balance the load, memory consumption, etc. This mapping relies on the elimination tree as well.

Factorization phase

The aim of this phase is to transform the initial matrix A (or the matrix modified by the analysis phase) into a product of factors; several factorizations exist, such as LU *factorization* ($A = LU$

with L a lower unitary triangular matrix, and U an upper triangular matrix), $L\,D\,L^T$ *factorization* for symmetric matrices ($A = L\,D\,L^T$ with L a lower unitary triangular matrix, and D a diagonal matrix), and *Cholesky factorization* for symmetric positive definite matrices ($A = L\,L^T$).

The factorization phase tries to follow as much as possible the preparation performed during the analysis phase, but has sometimes to adapt dynamically because of numerical issues (typically, a division by a very small diagonal entry, which implies round-off errors in the following operations): this can be done by *numerical pivoting* (permutations of rows and columns).

Solution phase

Once the matrix has been factorized, the linear system is finally solved. For example, in the case of an $L\,U$ factorization, the system $Ax = L\,Ux = b$ is solved in two steps (two solutions of triangular systems):

$$\begin{cases} y = L^{-1}b & \text{“forward substitution”} \\ x = U^{-1}y & \text{“backward substitution”} \end{cases}$$

Remark: an iterative refinement can be performed to improve the numerical quality of the solution [9]. Note than one of the main interests of direct methods is that once the factorization has been performed, it is easy to solve several linear systems $Ax = b_1, \ldots, Ax = b_n$: since the matrix is already factorized, only the solution phase needs to be performed.

2.2 Multifrontal methods

In our study, we consider a variant of direct methods called the *multifrontal method*. We present it briefly in this section: first, we show how the numerical factorization relies on the structure of elimination tree presented above, and then we present the solution phase and the solver used in this study.

Numerical factorization

The multifrontal method was initially developed for indefinite sparse symmetric matrices [19], then extended to the unsymmetric case [20]. Here we only outline this method; more details (amalgamation of the nodes and management of the unsymmetric case) are given in Appendix A.2.

A multifrontal factorization is processed as follows:

- It follows a topological ordering of the elimination tree, i.e. an ordering for which parent nodes are numbered after their children.
- Each node of the elimination tree is processed (eliminated) as follows:
  1. (for non-leaf nodes only) the node receives the contributions of its children.
  2. a so-called *frontal matrix* corresponding to the node is factorized. This matrix is dense, and one can take advantage of efficient BLAS kernels (Basic Linear Algebra Subroutines, specialized in high performance dense computations [17]).
  3. (for non-root nodes) the node sends a so-called *contribution block* to its father[1].

One of the main interests of this approach is that it expresses several levels of parallelism:

- *Tree-level parallelism:* the elimination tree exhibits tasks which can be performed independently.
- *Node-level parallelism:* one can take advantage of parallel dense kernels to process the nodes of the tree.

[1] Other approaches, such as *left-looking* and *right-looking* factorizations use the same tree, but do not send and receive contributions in the same way.

Solution phase

The solution phase, which consists of two successive triangular solutions (forward substitution on L and backward substitution on U for an LU factorization), also relies on traversals of the elimination tree associated to the matrix of the linear system. This is detailed and intensively used in the next chapter, where we show that, for our problem of computation of a set of entries of the inverse of a matrix, these traversals can be limited.

2.3 Solver used

MUMPS, Multifrontal Massively Parallel Solver

The experimental part and the developments performed during this study are based on `MUMPS` (MUltifrontal Massively Parallel Solver) [2, 5, 6]. This solver can handle any type of matrices (symmetric definite positive, general symmetric and general unsymmetric), and is aimed at solving large systems on massively parallel architectures (i.e. with many processors). It provides a large panel of functionalities, and especially the ability to run *out-of-core*: it offers the possibility to use units of storage like hard disks to store the factors when the main memory is not large enough; this feature has been developed in [1, 29], and we use some of the associated metrics in the following chapters.

Experiments

During this study, some experimental features have been implemented in `MUMPS` in order to perform experiments, essentially for the problem of right-hand sides permutations (metrics computations and heuristic algorithms; see Chapters 4 and 7).

Chapter 3

Computing entries of the inverse

We showed in the introduction that computing a subset of the inverse of a sparse matrix is requested in many applications, especially least-square problems and circuit studies. The first works related to this topic are quite old, and some of them were related to the latter (Takahashi, 1973 [31]).

In practical applications, computing the whole inverse operator A^{-1} is very seldom done, because using A^{-1} is far less efficient than using the factors, for example L and U ($\forall x, A^{-1}x = U^1 \left(L^{-1}x\right)$, computed by solving successively two linear systems, on L and U respectively). One of the main reasons for this phenomenon is that A^{-1} is usually much denser than the initial matrix or the factors[1].

In our study, we focus on the computation of a set of entries of the inverse of a sparse matrix. This set can have a peculiar structure (a diagonal, a block,...) or not (and some of our experiments were run with random sets). First, we describe a so-called simple method to compute the diagonal of the inverse of a symmetric matrix, which relies on a $L\,D\,L^T$ factorization; this approach is based on standard algorithms and was developed jointly with CESR researchers. Even if we show that this simple method is not extremely efficient, it has interesting features with respect to other, more efficient, methods.

We then describe two methods to compute an arbitrary set of entries; one is based on Takahashi's equations, and the other uses a "traditional" solution phase. We emphasize especially the latter, and we propose some extensions and improvements.

3.1 A simple approach to compute the diagonal

This approach is based on a very simple algebraic result: once one has an $L\,D\,L^T$ factorization of the symmetric matrix A to process, the diagonal elements can be computed as:

$$\forall i \in [\![1, n]\!], a_{i,i}^{-1} = \sum_{k=1}^{n} \frac{1}{d_{k,k}} {l_{k,i}^{-1}}^2$$

Hence, once one has an $L\,D\,L^T$ factorization of A, the diagonal elements can be computed rather simply by computing L^{-1}. This can be done by using basic algorithms (described in [30] for example, and in Appendix B), which compute L^{-1} by solving every system $Lx = e_i$.

This approach has the following interesting properties:

- It takes into account the sparsity of the right-hand sides, and the sparsity of the matrix L can be used easily.
- It is independent of the solver used (any $L\,D\,L^T$ factorization providing an explicit access to L is eligible).

[1] To quote Tim Davis, "Don't let that "inv" go past your eyes; to solve that system, factorize!" [15].

- It is very easy to implement, and several variants of implementation, such as "left-looking" and "right-looking" are possible. These implementations are presented in detail and compared in terms of performance in Appendix B.

However, this approach suffers several drawbacks:

- This method is difficult to extend in order to handle multiple right-hand sides (to solve several systems $Lx = e_i$ at the same time), because of the management of the working space.
- Adapting this approach to the case of an LU factorization is rather difficult, since one would have to compute U^{-1} as well.
- Extending this method in order to compute non-diagonal elements is rather difficult as well.

This method was an interesting starting point for this study[2] and a good reference approach to highlight the qualities of the other methods. This is the reason why we have tried to improve it (see the different implementations in Appendix B), and we have used it for our benchmarks.

3.2 Computing an arbitrary set with Takahashi's equations

This approach, developed by Takahashi, Fagan and Chin [31] was the first one to make a direct use of the factors L and U (and not their inverse, as in the previous method). It is based on an LDU factorization of A (a slight variant of the LU factorization, where D is simply obtained from the diagonal entries of U). The so-called *Takahashi's equations* relate the factors L, D and U to $Z = A^{-1}$, and are derived from the equation $Z = U^{-1}D^{-1}L^{-1}$:

$$\begin{cases} Z = D^{-1}L^{-1} + (I - U)Z & (1) \\ Z = U^{-1}D^{-1} + Z(I - L) & (2) \end{cases}$$

By deriving these equations, one can compute the entries in the upper part of Z and the entries in the lower part of Z by using the U and L respectively. These computations are illustrated in Figure 3.1: for example, if we use (1), L is a lower triangular matrix and D a diagonal matrix, thus $D^{-1}L^{-1}$ is lower triangular and does contribute to the elements of the upper part of Z; furthermore, $I - U$ is upper triangular: therefore, when computing an element (i, j) of the product $(I - U)Z$, elements $z_{1,j}$ to $z_{i,j}$ are multiplied by zero. This is illustrated on Figure 3.1(a), and the same ideas hold for the computation of an element of the lower part of Z (2) (cf. Figure 3.1(b)). Finally, we obtain:

$$\begin{cases} \forall i \leqslant j, z_{i,j} = d_{i,j}^{-1} - \sum_{k>i}^{n} u_{i,k} z_{k,j} \\ \forall i \geqslant j, z_{i,j} = d_{i,j}^{-1} - \sum_{k>j}^{n} u_{i,k} z_{k,j} \end{cases}$$

Using these equations recursively, one can compute the whole matrix (starting with $z_{n,n}$). However, as we said before, it is not advisable to compute the whole inverse. Erisman and Tinney [21], proposed a method to compute a subset of elements. First, they state the following theorem, which provides a recursive algorithm to compute the entries of Z which belong to the pattern of $(L + U)^T$:

Theorem 3.1 - *Theorem 1 in [21].*
Any $z_{i,j}$ which belongs to the pattern of $(L+U)^T$ can be computed as a function of L, U and $z_{p,q}$ where $z_{p,q}$ belongs to the pattern of $(L+U)^T$ $(q \geqslant j, p \geqslant j)$.

[2] Historically, this method had been experimented by the SPI team before they contacted the `MUMPS` team, using the `LDL` package by Tim Davis [14].

(a) Computation of the upper part of Z.

(b) Computation of the lower part of Z.

Figure 3.1: Computations of the entries of Z with Takahashi's equations.

It should be noticed that this theorem provides a sufficient but not necessary set of elements to compute before reaching the requested element(s), i.e. it provides a sequence of elements to compute in order to be able the obtain the requested elements, but this is sequence is not guaranteed to be the shortest. The same work also extends the previous result to compute entries of Z which do not belong to the pattern of $(L + U)^T$.

Finally, Campbell and Davis [12] provided a multifrontal-like approach to compute a set of elements in the case of a numerically symmetric matrix using Takahashi's equations. This method uses the elimination tree of the initial matrix (processed from the root to the leaves), and take advantage of Level 3 BLAS routines (matrix-matrix computations).

3.3 Computing an arbitrary set with a traditional solution phase

This approach represents the core and main motivation of our study. It is based on the work of Gilbert and Liu [22] and has been extended and implemented in `MUMPS` package (see [4, 29]). Here we present this method and propose some improvements in the next section.

Computing a single entry

Using the traditional phase of direct methods, the whole inverse A^{-1} can be computed by solving $AX = I$, where I is the identity matrix. Using the LU factors of A, this equation becomes:

$$\begin{cases} LY = I \\ UX = Y \end{cases}$$

Similarly, to compute a particular entry $a^{-1}_{i,j}$ of the inverse, we use $a^{-1}_{i,j} = (A^{-1}e_j)_i$. Using the factors, we obtain:

$$\begin{cases} y = L^{-1}e_j \\ a^{-1}_{i,j} = (U^{-1}y)_i \end{cases} \tag{3.1}$$

One can notice that in the forward substitution phase, the right-hand side (e_j) is sparse (only one non-zero entry), and that in the backward step only one entry of the solution of a triangular system is required. These properties can be exploited to enhance the solution phase.

First, we use a theorem stated by Gilbert and Liu [22]:

Theorem 3.2 - *Structure of the solution of a triangular system (Theorem 2.1 in [22]).*
For any lower triangular matrix L, the structure of the solution vector x of $Lx = b$ is given by the set of nodes reachable from nodes associated with the right-hand side entries by paths in the directed graph $G(L^T)$ of the matrix L^T.

Using the LU factors of a given matrix A, this property can be extended to predict the structure of the solution of $Ax = b$ (Property 8.4 in [29]). In the case of the computation of an element $a^{-1}_{i,j}$ of A^{-1}, this property can be rewritten to express which path is followed in the elimination tree, i.e., given that each node correspond to a part of L and U, which factors are actually used:

Theorem 3.3 - *Factors to load to compute a particular entry of the inverse (Property 8.9 in [29]).*
To compute a particular entry $a^{-1}_{i,j}$ in A^{-1}, the only factors which have to be loaded are the L factors on the path from node j up to the root node, and the U factors going back from the root to node i.

This property is illustrated on Figure 3.2: if one wants to compute $a^{-1}_{3,1}$, then the tree has to be traversed from node 1 to the root for the forward substitution, then from the root to node 3 for the backward substitution. Note that nodes 6 and 5 are loaded twice (there is no way to spare these accesses).

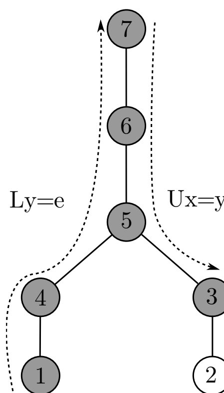


Figure 3.2: Traversal of the tree for the computation of $a^{-1}_{3,1}$.

Thanks to this result, the elimination tree can be "tidied-up" of all the useless nodes: this technique is called *tree pruning*, and is implemented in MUMPS with an algorithm called *branch detection*[3] [29]. Then the solution phase relies on this so-called *pruned tree* instead of the initial tree, which of course, makes the solution phase more efficient by sparing accesses to useless nodes and related factors.

[3]Other *prunings*, such as *subtree detection* for the computation of nullspaces, have been used in Slavova's thesis.

Computing multiple entries

In practice, several entries (a diagonal, a block, ...) are requested at the same time. In this case, one has to solve the system

$$AX = E$$

where E is formed with vectors of the canonical basis (i.e. columns of the identity): a vector e_j is a column of E if and only if at least one entry of the j-th column of A^{-1} is requested. For example, if $a^{-1}_{1,3}$, $a^{-1}_{2,6}$ and $a^{-1}_{5,6}$ are requested, then $E = [e_3, e_6]$.

Here we focus on an *out-of-core* context, where the hard disk is used to store the factors (when the main memory is not large enough). In this situation, it is capital to minimize the number of accesses to the disk, hence, during the solution phase, the number of times the factors are read. Therefore, computing several entries of the inverse at the same time instead of computing them independently is particularly interesting: indeed, if the requested row (or column) entries share a common ancestor in the tree, then all the nodes between the root and this node have to be loaded only once when the entries are computed simultaneously, while they would have to be loaded as many times as there are entries requested, if these entries were computed independently.

More formally, say the entries $a^{-1}_{i,j}, i \in I, j \in J$ are requested. We note $P(k)$ the path from a node k to the root. During the forward phase, all the nodes on all the paths from nodes of J to the root have to be loaded; if all the entries are requested simultaneously, all the nodes in $\bigcup_{j \in J} P(j)$ are loaded, hence there are $\left|\bigcup_{j \in J} P(j)\right|$ accesses, whereas, if the entries were computed independently, there would be $\sum_{j \in J} |P(j)|$ accesses, which is a higher amount. The same argument holds for the backward phase, where there are $\left|\bigcup_{i \in I} P(i)\right|$ accesses instead of $\sum_{i \in I} |P(i)|$.

For example, say one wants to compute $a^{-1}_{3,1}$ and $a^{-1}_{4,1}$ simultaneously on the example of Figure 3.2: during the forward phase, an access to 4,5,6 and 7 is spared, and during the backward phase, an access to 7, 6, and 5 is spared.

This property is extremely interesting, but it assumes that one is able to compute many entries at the same time, i.e. solve a linear system with many right-hand sides, which is not always possible because of memory constraints. Ideally, a maximum gain is ensured when all the requested entries are computed at the same time, but in practice, it is seldom feasible to solve the associated system. Therefore, the requested entries are partitioned into blocks (16 is a typical size used during the tests). The remaining question is to find how to form these blocks such that the numbers of accesses to the factors is minimized (or nearly minimized). This is a major point of our study; in the next chapter, we address this combinatorial problem, present what has been done in previous works and propose some extensions.

3.4 Comparison of the different methods

Here we compare the different approaches in terms of amount of accesses to the factors. First, we show why the traditional solution phase outperforms the simple approach relying on an $L\,D\,L^T$ factorization (see 3.1) thanks to the structure of elimination tree. Then we compare the traditional solution phase with Takahashi's approach, and show that they perform the same accesses to the factors (the entries needed to compute another entry are the same).

Simple computation of the diagonal vs. traditional solution phase

Here we consider a case where the whole diagonal of A^{-1} is requested. First, we derive from Theorem 3.3 a simple corollary:

Corollary 3.1 - *Application of Theorem 3.3 to compute $diag(A^{-1})$, with A symmetric and its $L\,D\,L^T$ factorization.*
To compute $diag(A^{-1})$, the factors accessed are, for each diagonal entry i, the columns of L on the path from i up to the root node.

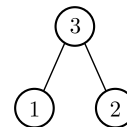
This result highlights the role of the elimination tree; we illustrate it on a simple example: we choose a 3×3 symmetric matrix; Figure 3.3 shows the structure of the matrix, the structure of its factor L and the structure of the associated elimination tree.

$$\begin{pmatrix} 1 & & X \\ & 2 & X \\ X & X & 3 \end{pmatrix}$$

(a) Initial matrix.

$$\begin{pmatrix} 1 & & \\ & 2 & \\ X & X & 3 \end{pmatrix}$$

(b) Factor L.



(c) Elimination tree.

Figure 3.3: An simple example for the computation of $diag(A^{-1})$.

On this example, the different algorithms perform the following accesses:

- With the simple approach, relying on an $L\,D\,L^T$ factorization, $n = 3$ triangular systems are solved:
  - solution of $Lx = e_1$: access to $L_{*,1}$, $L_{*,2}$ and $L_{*,3}$ ($L_{*,j}$ stands for the j-th column of L).
  - solution of $Lx = e_2$: access to $L_{*,2}$ and $L_{*,3}$.
  - solution of $Lx = e_3$: access to $L_{*,3}$.
- With the traditional solution phase (the elements of the diagonal are computed one by one):
  - computation of $a^{-1}_{1,1}$: access to $L_{*,1}$ and $L_{*,3}$.
  - computation of $a^{-1}_{2,2}$: access to $L_{*,2}$ and $L_{*,3}$.
  - computation of $a^{-1}_{3,3}$: access to $L_{*,3}$.

What has to be noticed is that the elimination tree expresses the independence of x_1 and x_2: in order to solve $Ax = b$, one does not need to know x_1 to compute x_2 and conversely (this is characterized on the tree by the fact that node 1 is not an ancestor nor a descendant of node 2, and conversely). This is why, in our example, the traditional solution phase avoids a useless access to $L_{*,2}$. This shows how following paths in the elimination tree avoids useless accesses. Furthermore, in this example, the entries are computed one by one: more accesses could be spared by computing them at the same time (see the previous section).

Takahashi's approach vs. traditional solution phase

To finish, we compare the amount of LU factors which have to be accessed between the method based on Takahashi's equations and the method based on a traditional solution phase. We quote the following property:

Theorem 3.4 - *Property 8.11 in [29].*
Let A be an unsymmetric irreducible matrix with symmetric pattern and T be its corresponding elimination tree. To compute an off-diagonal entry $a^{-1}_{i,j}$, every column of L from node i of T to the root and every row of U from node j to the root need to be loaded with both approaches.

Proof. For the traditional solution phase, this is Theorem 3.3. For the approach based on Takahashi's equations, this comes from a property in [21] which states that, to compute an entry $z_{i,j}$, all the inverse entries from the root node to nodes i and j have to be computed. This property can be extended to show that the only factors in L and $D^{-1}U$ which need to be loaded are on the path from node i up to the root node and the path from node j up to the root respectively. ☐

Finally, in terms of amount of loaded factors, there is no difference between these two methods. An interesting quality of the traditional solution phase is that this way of handling sparse right-hand sides can be extended to other problems, such as the computation of the nullspace of a deficient matrix: in this case, one wants to solve $Ux = 0$, and this solution can be strongly improved by pruning the elimination tree through a mechanism called *subtree detection*. This has been implemented in `MUMPS` [29], and both A^{-1} and nullspace functionalities share a common framework.

Chapter 4

On the influence of right-hand sides permutations

Here we focus on the computation of a set of elements of the inverse of a matrix using the method based on the traditional solution phase. We showed in the previous chapter that computing several entries at the same time could lead to spare some accesses to the factors: nodes which belong to several paths in the elimination tree between a node corresponding to a requested row or column and the root node are loaded only once. Therefore, if all requested entries are computed at the same time, a minimum number of factors is loaded; unfortunately, it might be too expensive in terms of memory consumption to solve the potentially very large associated system.

In order to overcome this problem, the requested entries are computed by blocks. Hence, we wonder if it is possible to form these blocks such that the number of accesses to the factors is minimized, or nearly so. We begin by showing that ordering the right hand sides associated to the requested entries has a strong influence on the number of accesses to the factors; therefore, the need for efficient orderings is justified. We then show that simple orderings, based on topological orderings induced by the elimination tree, are interesting but not efficient enough in general; thus, we address the problem of the construction of an efficient ordering through three complementary approaches: a global approach based on an hypergraph model, a constructive approach based on the shape of a solution, and local approach based on a succession of improvements of an initial ordering.

4.1 Influence of orderings of right-hand sides

Influence of block size

We have showed previously that the ideal case was obtained when all the requested entries are computed at the same time; conversely, a block size of 1 (i.e. computing the entries independently) leads to a maximum number of accesses. An interesting remark is that in an *in-core* context, where all the computations are done in the active memory, this situation is inverted: in this case, what matters is the number of *flops* (floating-point operations). When processing several right-hand sides at the same time, blocked computations are processed on the union of the structures of these vectors, hence there are more computations than there would be if these right-hand sides were processed one by one (of course, the interest is to benefit from dense kernels and thus to process the right-hand side by blocks of reasonable size). Therefore, a block size of 1 minimizes the number of flops, while processing all the right-hand sides at the same time maximizes it.

The context of our study is the following:

- Out-of-core context (hence the metrics is the number of accesses to the factors).
- Sequential execution (i.e. only one processor; some ideas to handle the parallel case are suggested in [29]).

- There is only one non-zero entry per right-hand side: this means that only one entry per column of the inverse is requested at most. The following results are invalid when this assumption is not satisfied (see below).
- We work with a fixed given block size.

An interesting quantity is the lower bound of the amount of factors to load (in a sequential environment). To compute it, we introduce the notion of *number of requests* (for the forward or backward substitution phase), which expresses the number of times a request lies on a path from a requested entry to the root:

Definition 4.1 - *Number of requests of a node.*
The number of requests nb_requests of a node i is recursively defined by :

$$\text{nb_requests}(i) = \sum_{j \in children(i)} \text{nb_requests}(j) + \text{req}(i)$$

where req(i) is the number of requested entries corresponding to i, i.e. the number of times where i is a column subscript (for the forward phase), or row subscript (for the backward phase).

Theorem 4.1 - *Lower bound of the amount of factors to load in a sequential environment.*
Let i be a node of the elimination tree. We note $\text{nb_requests}_L(i)$ and $\text{nb_requests}_U(i)$ the number of requests of i in the forward and backward substitution phases respectively. Node i has a size size(i). The block size is block_size. Therefore, the lower bound of the amount of factors to load in a sequential environment is :

$$\sum_{all\,nodes} \text{size}(node) \left\lceil \frac{\text{nb_requests}_L(node)}{\text{block_size}} \right\rceil + \sum_{all\,nodes} \text{size}(node) \left\lceil \frac{\text{nb_requests}_U(node)}{\text{block_size}} \right\rceil$$

Proof. The sum of the lower bound of the forward phase and the lower bound of the backward phase provides a lower bound for the whole process. For any phase, the ratio nb_requests($node$) over block_size is the lowest number of blocks a node can belong to (and the "ceiling" indicates that this number is an integer, at least 1). Hence it represents the lowest number of accesses to this node. Finally, by multiplying by the size of the node and summing on all the nodes of the tree, we obtain the lower bound, which, of course, might not be accessible, because it might be impossible to reach the minimum number of accesses to every node at the same time, and to reach the lower bound of the forward and backward phase at the same time. □

We show on Figure 4.1 an example of lower bound for the forward phase: the requested entries are $a^{-1}_{2,2}$, $a^{-1}_{4,4}$ and $a^{-1}_{5,5}$. The number of requests of node 3 is 1 (because of node 2) + 0 (because it is not requested); for node 5 it is 1 (because of node 3) + 1 (because of node 4) + 1 (because it is requested). Hence the lower bound (for any phase), with a block size of 2, is:

$$\begin{aligned} \text{Lower-bound} &= \left\lceil \frac{0}{2} \right\rceil + \left\lceil \frac{1}{2} \right\rceil + \left\lceil \frac{1}{2} \right\rceil + \left\lceil \frac{1}{2} \right\rceil + \left\lceil \frac{3}{2} \right\rceil + \left\lceil \frac{3}{2} \right\rceil + \left\lceil \frac{3}{2} \right\rceil \\ &= 0 + 1 + 1 + 1 + 2 + 2 + 2 \\ &= 9 \end{aligned}$$

A first try: topological orderings

Here we present a first natural idea, which provided interesting results and justified the need for an advanced study. For the sake of simplicity, we consider the case where one wants to compute diagonal entries. A simple observation shows that when using a topological order of a tree (i.e. an order which numbers child nodes before their father), such as pre-orders or post-orders, all nodes

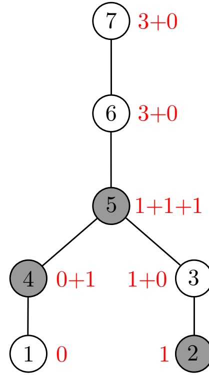


Figure 4.1: Example of lower bound for the forward phase, with a block size of 2; entries $a^{-1}_{2,2}$, $a^{-1}_{4,4}$ and $a^{-1}_{5,5}$ are requested; and the number of requests are indicated for each node.

in a given subtree are numbered consecutively. Therefore, one can expect that using a topological order will provide a good locality for factor reuse between consecutive columns of the right-hand sides: intuitively, consecutive right-hand sides for a topological ordering will correspond to nodes which have "good chances" to be close in the elimination tree, hence to share a "long" common path to the root.

Therefore, one can try to use topologically-based permutations, such as pre-order or post-order. Note that when the number of right-hand sides is a multiple of the block size, there is no difference between these two orderings; in the opposite case, they will be different, and for a post-order, the last block will contain nodes in the upper part of the tree, with short paths to the root.

These orderings have been tested, first in the diagonal case with matrices from CESR, then in the general case, with matrices of all kinds. These experiments are described in Chapter 7 and in [8][1]; they essentially show that topological orderings provide interesting (quasi-optimal) results for some problems, but can also provide very bad results (a factor of 3 compared to the lower-bound) on some generic problems (come from real applications).

We also provide a structural example which shows a situation were a pre or post-order provides the worst permutation possible, while a simple permutation, based on the structure of the tree, reaches the minimum. A minimal instance is illustrated in Figure 4.2, and its generalization is shown in Figure 4.3. In the minimal instance, we have:

$$\begin{aligned}\text{Lower-bound} &= \left\lceil\frac{1}{2}\right\rceil + \left\lceil\frac{1}{2}\right\rceil + \left\lceil\frac{2}{2}\right\rceil + \left\lceil\frac{1}{2}\right\rceil + \left\lceil\frac{3}{2}\right\rceil \\ &= 1+1+1+1+2 \\ &= 6\end{aligned}$$

$$\begin{aligned}\text{Post-order (red, small dashes)} &= 4 \text{ (accesses to 1,2,3,5) } + 3 \text{ (accesses to 3,4,5)} \\ &= 7\end{aligned}$$

$$\begin{aligned}\text{Simple permutation (blue, large dashes)} &= 3 \text{ (accesses to 2,3,5) } + 3 \text{ (accesses to 1,4,5)} \\ &= 6 \text{ (an access to 3 has been spared)} \\ &= \text{Lower-bound}\end{aligned}$$

[1]This extended abstract, written with P. Amestoy and B. Uçar, has been accepted for a presentation at the SIAM conference CSC09 (Combinatorial Scientific Computing).

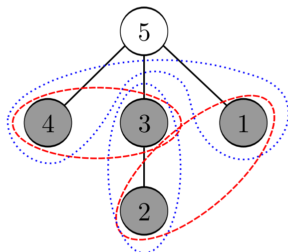


Figure 4.2: Minimal instance of the structural example. Post-order partitioning is represented with red, small dashes, and optimal partitioning with blue, large dashes.

This example can be generalized to p blocks of two right-hand sides, as illustrated on Figure 4.3: there are r_i nodes between each (requested) leaf and the requested node above, and for these nodes, the number of requests is 1. There are q_i nodes between each requested node and the root node, and for these nodes, the number of requests is 2. Therefore, we have:

$$\text{Lower-bound} = s + \sum_{i=1}^{p-1} r_i + \sum_{i=1}^{p-1} q_i + t + p$$

$$\begin{aligned}
\text{Post-order (red, small dashes)} &= s + \sum_{i=1}^{p-1} r_i + 2\sum_{i=1}^{p-1} q_i + t + p \\
&= \text{Lower-bound } + \sum_{i=1}^{p-1} q_i \\
&= \sum_{\substack{\text{requested}\\ \text{entries}}} (\text{length}(\text{root} \to \text{entry}) - 1) \\
&= \text{Max} \\
\text{Simple permutation (blue, large dashes)} &= s + \sum_{i=1}^{p-1} r_i + \sum_{i=1}^{p-1} q_i + t + p \\
&= \text{Lower-bound}
\end{aligned}$$

Here we notice that the post-order performs the highest number of accesses possible, and that an arbitrary overhead can be produced (asymptotically, if $q_i \to \infty$, there is a factor of 2).

4.2 A global approach: hypergraph model

First, we present an approach which tackles the whole problem by showing that it is equivalent to an hypergraph partitioning problem. We first introduce the notion of hypergraph and the general problem of partitioning an hypergraph, and then we show how these notions can be applied to our problem.

A brief introduction to hypergraphs

Informally, hypergraphs are a generalized form of graphs where edges are allowed to connect more than two vertices. These edges are called *hyperedges* or *nets*. Hypergraphs and hypergraph partitioning are widely used in VLSI (Very Large Scale Integration) circuit design and parallel computing (sparse matrix-vector product, matrix reordering, . . .) for example.

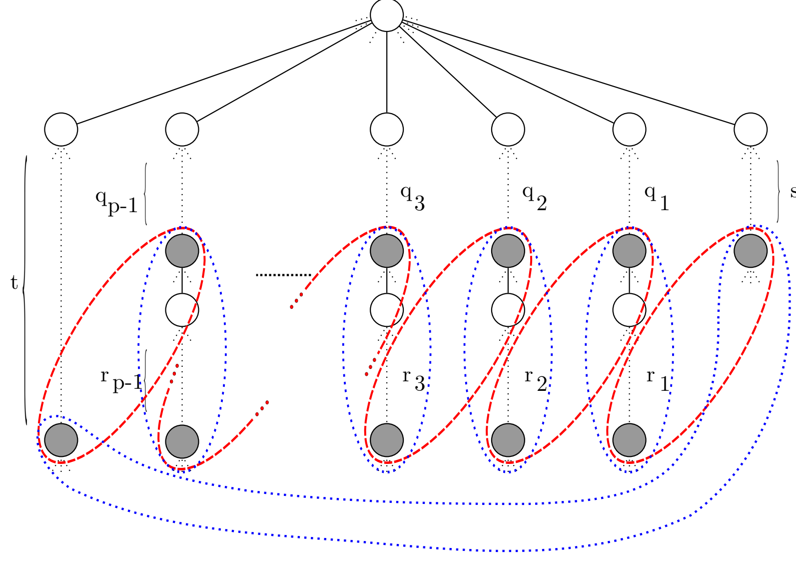


Figure 4.3: Generalization of the structural example. Post-order partitioning is represented with red, small dashes, and optimal partitioning with blue, large dashes.

Definition 4.2 - *Hypergraph.*
A *hypergraph* $H = (V, N)$ is defined as a set of *vertices* V and a set of *nets* N, where every net is a subset of vertices.

Definition 4.3 - *K-way vertex partition.*
$\Pi = \{V_1, \dots, V_K\}$ is a *k-way vertex partition* of a hypergraph $H = (V, N)$ if and only if:

- Each part V_k is a non-empty subset of V: $\forall k \in [\![1, K]\!], V_k \neq \varnothing \wedge V_k \subseteq V$.
- Parts are pairwise disjoint: $\forall (k, k') \in [\![1, K]\!]^2, k \neq k' \Rightarrow V_k \cap V_{k'} = \varnothing$.
- Union of the parts is equal to V: $\bigcup_{k=1}^{K} V_k = V$.

Definition 4.4 - *Connectivity of a net and partitioning problem.*
Given a partition Π, the *connectivity* $\lambda(i)$ of a net n_i of N is the number of parts of Π connected by n_i:

$$\lambda(i) = |\{V_k \in \Pi \ ; \ \exists v \in V_k / v \in n_i\}|$$

A cost $c(i)$ is associated to each net n_i, and in the *hypergraph partitioning problem*, the objective is to find a partition Π which minimizes the *cutsize*:

$$\text{cutsize}(\Pi) = \sum_{n_i \in N} c(i)(\lambda(i) - 1)$$

The hypergraph partitioning problem is known to be NP-hard. Several strategies, which often appear to be generalizations of methods used in graph partitioning exist: local heuristics like Kernighan-Lin algorithm, global methods like multilevel partitioning, etc. [27]. Several hypergraph partitioning packages exist, and the one we used is PaToH (Partitioning Tools for Hypergraphs), which implements a multilevel approach [13].

Model for diagonal entries of A^{-1}

Here we show how the problem of finding an optimal permutation of right-hand sides (to minimize the numbers of factors loaded) can be transformed into a hypergraph partitioning problem. First, we introduce a model for diagonal entries developed in [29].

We begin with explaining intuitively the model; the main idea is the following: when a node, say i, is requested, all nodes on the path from this node to the root have to be loaded when processing the associated right-hand side. Some of these nodes (likely near i) are loaded only when i is processed (they belong to $P(i)$ only - we recall that $P(i)$ is defined as the path from i up to the root node); others nodes belong to other paths, hence can be loaded several times. Therefore, the idea is to cut a path into different pieces, corresponding to the intersections with other paths. These pieces of paths will define the nets of the hypergraph, in the sense that all requested nodes sharing this piece of path will belong to the nets; the cost of the nets is thus the size of the corresponding piece of path. The beginnings of each path (piece of path between the requested entry and the first intersection with another path) will form a first set of nets, and the other pieces (pieces of path from an intersection to another) will form a second set. The vertices will be the requested entries, and an entry, say i, belongs to a net, say n, if $n \subseteq P(i)$, i.e. if processing this entry requires to load the piece of path corresponding to n. The cost of a net will be the cost of loading the associated path; therefore, when partitioning the vertices (the requested entries), the connectivity $\lambda(i)$ of a net will correspond to the number of times the associated piece of path is loaded, and finally, the cutsize will correspond to the overhead generated by the partition, compared to the lower bound (n.b.: actually only a half of the overhead, because we count only a phase - backward of forward solution). Therefore, minimizing the cutsize of our hypergraph will be equivalent to minimizing the cost for computing the requested entries. We thus define the model formally:

Definition 4.5 - *Hypergraph model for diagonal entries.*
We define the set of vertices, the set of nets and the associated costs:

- Vertices: the vertex set V is equal to the set of requested entries.
- Nets: there are two types of nets:
  - N_1: there is a net in N_1 for each requested node.
  - N_2: there is a net in N_2 for every node f_i, where f_i is the lowest node in the intersection $P(i) \cap \bigcup_{i' \neq i} P(i')$, called the *least common ancestor* or *lowest ancestor* of node i with another node.

  Nets are defined as follows: a net n_i is defined as the set of requested nodes and lowest ancestors in the subtree rooted at i.
- Costs: the cost of a net n_i is $c(n_i) = \text{Cost}(i, f_i) = \sum_{k \in P(i,f_i) \setminus f_i} \text{size}(k)$.

Theorem 4.2 - *Structure of the hypergraph model.*
Each net n_i is the union of any net n_k where i is an ancestor of k in the elimination tree.

Theorem 4.2 shows that the hypergraph modelling our problem has a very peculiar structure, since there are chains of inclusions between the nets. Unlike the general hypergraph partitioning problem, knowing whether partitioning such a hypergraph is feasible in a polynomial time or not remains an open question.

Figure 4.4 shows an example of such a hypergraph: the requested entries are $a^{-1}_{3,3}$, $a^{-1}_{4,4}$, and $a^{-1}_{14,14}$. Therefore, $V = \{3, 4, 14\}$ and $N_1 = \{n_3, n_4, n_{14}\}$. Moreover, $N_2 = \{n_7\}$ because 7 is the lowest node in $P(3) \cap P(4)$; each net is the union of the sets associated to its descendants, therefore

: $n_3 = \{3\}, n_4 = \{4\}, n_7 = \{3,4\}, n_{14} = \{3,4,14\}$. For this example, assuming that each node has size 1, the cutsize is:

$$\begin{aligned} \text{cutsize} &= c(n_3)(\lambda(n_3)-1) + c(n_4)(\lambda(n_4)-1) + c(n_7)(\lambda(n_7)-1) + c(n_{14})(\lambda(n_{14})-1) \\ &= 1(1-1) + 2(1-1) + 1(2-1) + 1(2-1) \\ &= 2 \end{aligned}$$

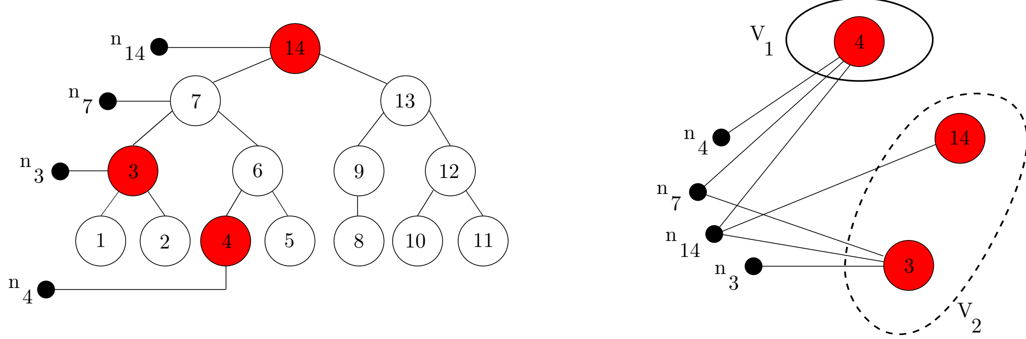


Figure 4.4: Example of hypergraph model for diagonal entries: $a^{-1}_{3,3}$, $a^{-1}_{4,4}$, and $a^{-1}_{14,14}$ are requested.

Extension to the general case

We have extended the previous model in order to handle the case where some requested entries are non-diagonal[2]. The idea is to use the fact that $a^{-1}_{i,j} = \left(A^{-1}e_j\right)_i$ and Equation 3.1: basically, the hypergraph obtained appears like a "sewing" of the two hypergraphs the initial model would give for the diagonal entries associated to the column subscripts (forward phase) and the diagonal entries associated to the row subscripts (backward phase):

Definition 4.6 - *Hypergraph model for the general case.*
We define the set of vertices, the set of nets and the associated costs:

- Vertices: a node for each requested element: there is a node (i,j) if and only if $a_{i,j}$ is requested.
- Nets: the requested i's (row subscripts) and j's (column subscript)s define their sets of nets :
  - N_1^{up}: a net for each j of each requested element, with a single vertex (to the corresponding requested element). Remark: a situation where two nodes (i_1, j) and (i_2, j) are requested is not allowed, because the model does not express the fact that these entries are necessarily processed simultaneously.
  - N_2^{up}: a net for each node f_i in the intersection of a number of paths to the root: each net n_l^{up} is the union of any net n_k^{up}, where node j is ancestor of node k in the elimination tree
  - N_1^{down}: a net for each requested i. Remark: if two elements in the same row are requested (say A^{-1}_{i,j_1} and A^{-1}_{i,j_2}), two nets are created: $n^{\text{up}}_{i_1}$ linking to node (i, j_1) and $n^{\text{up}}_{i_2}$ linking to node (i, j_2).
  - N_2^{down}: same thing than N_2^{up}, but using the "down" nodes
- Costs: same thing than in the diagonal model : $c(n_i) = Cost(i, f_i)$.

[2]However, there is still only one non-zero entry per column of the right-hand side.

Figure 4.5 shows an example where the requested entries are $a^{-1}_{8,3}$, $a^{-1}_{5,4}$ and $a^{-1}_{12,14}$. The elimination trees (and the nets) as well as the hypergraph are represented.

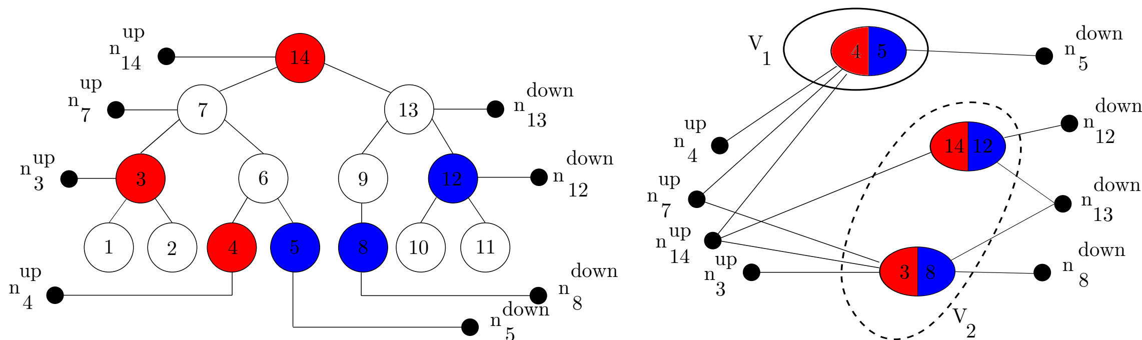


Figure 4.5: Example of hypergraph model for the general case: $a^{-1}_{8,3}$, $a^{-1}_{5,4}$ and $a^{-1}_{12,14}$ are requested.

Several remarks can be done:

- This kind of "sewing" is referred in other applications (such as sparse matrix partitioning) as *vertex amalgamation* [32].
- The model does not work for the case where a column of the right-hand side contain more than one non-zero entry (the fact that two entries in the same column are automatically grouped is not taken into account).
- In the diagonal case, the "up" set of nets is equal to the "down" set; hence the cutsize of this model is twice the cutsize of the diagonal model, i.e. the real overhead.

4.3 A constructive approach: structure of a solution

Here we present a constructive approach which tries to build an optimal solution. First, we give a necessary and sufficient condition for optimality, and then we present a heuristic algorithm which tries to build a solution driven by this condition.

Structure of an optimal solution

The property we propose uses the notion of *encompassing tree*:

Definition 4.7 - *Encompassing tree of a set of nodes.*
Let $S = \{s_1, \ldots, s_n\}$ be a set of nodes of a tree T. The *encompassing tree* of S is the minimal tree T^e_S which contains S. Let f_S be the lowest common ancestor of the nodes in S; then

$$T^e_S = \bigcup_{s_i \in S} P(s_i, f_S)$$

We recall that $P(s_i, f_S)$ is the path from node s_i to the root node f_S. We also need the notion of *number of blocks below a node*:

Definition 4.8 - *Number of blocks below a node.*
Let n be a node of the tree and $\Pi = \{S_1, \ldots, S_p\}$ a partition of the requested entries (p blocks of size b). The *number of blocks below* n is defined as:

$$\text{nb_blocks_below}_\Pi(n) = |\{S \in \Pi / T^e_S \subset \text{Subtree}(n)\}|$$

We first provide several lemmas which re-express the number of requests and the number of accesses of a node as functions of the number of blocks below a node, and which will be useful to prove the necessary and sufficient condition.

Lemma 4.1

$$\forall n, \text{nb_blocks_below}_\Pi(n) = \sum_{c \in \text{Children}(n)} \text{nb_blocks_below}_\Pi(c) + |\{S \in \Pi / T^e_S \text{ is rooted at } n\}|$$

Proof. This is simply due to the fact that blocks below each children of n form disjoint sets (hence the first part of the expression), and that blocks with encompassing trees rooted at n are not counted in any number of blocks below a child (hence the second part). □

We then provide a lemma which relates the number of requests of a node (we recall that it is defined as the number of times a node lies on a path from a requested entry to the root node) to a partition of the requested entries, although the number of requests is independent of the partition.

Lemma 4.2

$$\forall n, \text{nb_requests}(n) = b \cdot \text{nb_blocks_below}_\Pi(n) + \sum_{S \in \mathcal{S}_n} |S \cap \text{Subtree}(n)|$$
$$\text{with } \mathcal{S}_n = \{S \in \Pi / n \in T^e_S \wedge T^e_S \text{ not rooted at } n\}$$

Proof. This is proven recursively, following a topological order on the nodes of the tree.

- base case: for a leaf l of the tree, $\text{nb_blocks_below}_\Pi(l) = 0$, and $\sum_{\mathcal{S}_l} |S \cap \text{Subtree}(n)| =$ 1 if n is requested, 0 otherwise ($\mathcal{S}_l$ is reduced to the block containing l if l is requested, and is empty otherwise.
- general case: for a node n of the tree, we have, on the one hand:

$$\begin{aligned}
\text{nb_requests}(n) &= \sum_{c \in \text{Children}(n)} \text{nb_requests}(c) + \text{req}(n) \\
&= \sum_{c \in \dots} \left(b \cdot \text{nb_blocks_below}_\Pi(c) + \sum_{S \in \mathcal{S}_c} |S \cap \text{Subtree}(c)| \right) + \text{req}(n) \\
&= \text{nb_blocks_below}_\Pi(n) \\
&\quad - b \cdot |\{S / T^e_S \text{ is rooted at } n\}| + \sum_{c \in \dots} \sum_{S \in \mathcal{S}_c} |S \cap \text{Subtree}(c)| + \text{req}(n) \qquad (*)
\end{aligned}$$

On the other hand, using $\text{Subtree}(n) = \{n\} \cup \bigcup_{c \in \text{Children}(n)} \text{Subtree}(c)$, we have:

$$\begin{aligned}
\sum_{S \in \mathcal{S}_n} |S \cap \text{Subtree}(n)| &= \sum_{S \in \mathcal{S}_n} \left| S \cap \left(\{n\} \bigcup_{c \in \text{Children}(n)} \text{Subtree}(c) \right) \right| \\
&= \sum_{S \in \mathcal{S}_n} \sum_{c \in \text{Children}(n)} |S \cap \text{Subtree}(c)|
\end{aligned}$$

Similarly, $\mathcal{S}_n$ can be decomposed, and this gives the equality with $(*)$. The main idea is that the number of requests of nodes in blocks which are below n but not its children will be summed, and cancel the first term in $(*)$; $\text{req}(n)$ will be summed with $|S \cap \text{Subtree}(c)|$ for the children which are in the same block that n.

□

Figure 4.6 illustrates this property: for each node n, the number of blocks below the node, the number of requests, and the ratio $\left\lceil \frac{\text{nb_requests}_{\Pi}(n)}{b} \right\rceil$ is given. For example, for node "n" (marked in red), the number of blocks below is 1, the number of requests is 5, and the ratio is 2.

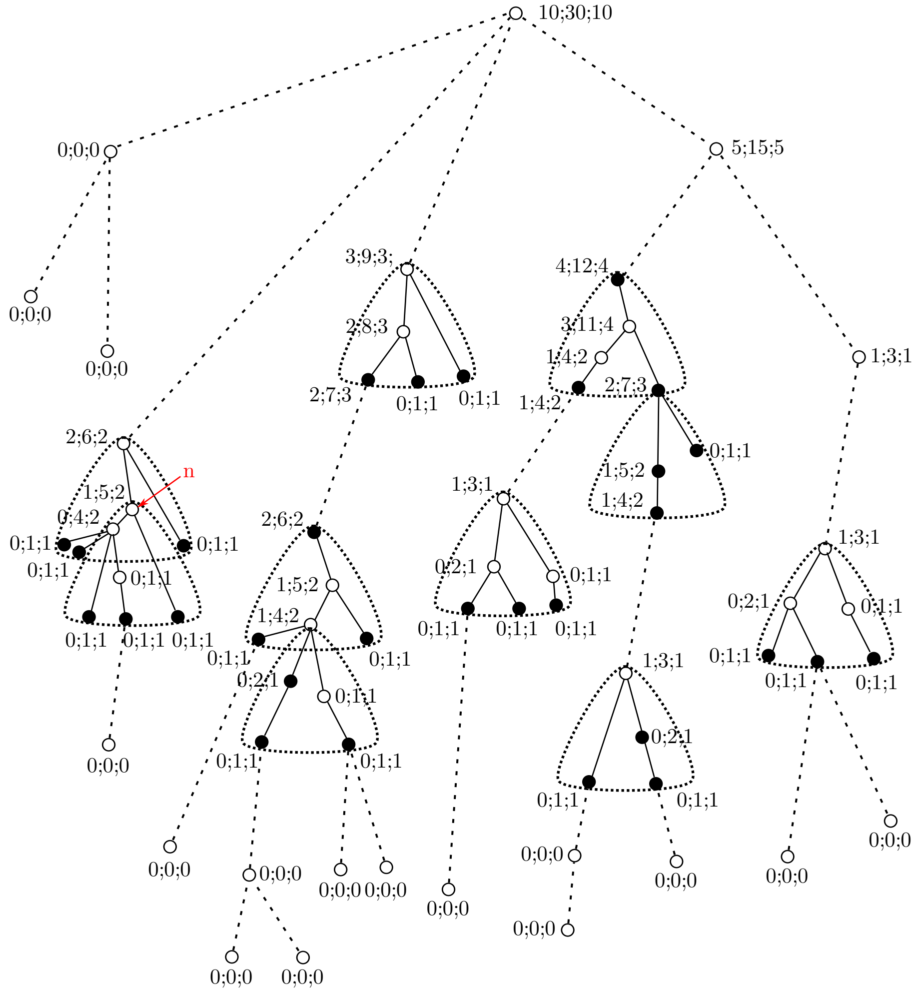


Figure 4.6: Illustration of the proof of the necessary and sufficient condition. For every node n, the number of blocks below the node, the number of requests, and the ratio $\left\lceil \frac{\text{nb_requests}_{\Pi}(n)}{b} \right\rceil$ is given.

Similarly, we relate the number of accesses to the partition, thanks to the notion of number of blocks below a node:

Lemma 4.3

$$\forall n, \text{nb_accesses}(n) = \text{nb_blocks_below}_{\Pi}(n) + |\mathcal{S}_n|$$

Proof. The number of accesses to n is the number of blocks such that n is on the path from (at least) a node of this block to the root. This is the case with blocks which are "below" n, and with

blocks with encompassing trees including n (except those rooted at n, because they are already counted in $\text{nb_blocks_below}_\Pi(n)$. □

Now that we have expressed the number of requests of a node (which is used to compute the lower bound of the number of accesses) and the number of accesses of a node as functions of the partition of the requested entries, it is rather simple to know whether a partition can reach the lower bound or not. Since both quantities contain $(\text{nb_blocks_below}_\Pi(n)$, the key is to compare the other terms, $\sum_{S\in\mathcal{S}_n} |S \cap \text{Subtree}(n)|$ and $(|\mathcal{S}_n| - 1)$ (which is the term which can implies non-optimality):

Theorem 4.3 - *Necessary and sufficient condition for reaching the lower bound (diagonal case).*
Let $\Pi = \{S_1, \ldots, S_p\}$ a partition of the requested entries (p blocks of size b). The number of accesses achieved when processing the right-hand sides according to Π is equal to the lower bound of Theorem 4.1 if and only if:

$$\forall n, \sum_{S\in\mathcal{S}_n} |S \cap \text{Subtree}(n)| > b \cdot (|\mathcal{S}_n| - 1)$$

Proof. Since for each node n, the ratio $\lceil \frac{\text{nb_requests}(n)}{b} \rceil$ defines the minimum number of accesses to n, the total number of accesses achieved when processing the right-hand sides according to Π is equal to the lower bound of Theorem 4.1 if and only if

$$\begin{aligned}
&\forall n, \text{nb_accesses}(n) \leqslant \left\lceil \frac{\text{nb_requests}(n)}{b} \right\rceil \\
\Leftrightarrow &\forall n, \text{nb_blocks_below}_\Pi(n) + |\mathcal{S}_n| \leqslant \left\lceil \frac{b \cdot \text{nb_blocks_below}_\Pi(n) + \sum_{S\in\mathcal{S}_n} |S \cap \text{Subtree}(n)|}{b} \right\rceil \\
\Leftrightarrow &\forall n, |\mathcal{S}_n| \leqslant \left\lceil \frac{\sum_{S\in\mathcal{S}_n} |S \cap \text{Subtree}(n)|}{b} \right\rceil \\
\Leftrightarrow &\forall n, |\mathcal{S}_n| < \frac{\sum_{S\in\mathcal{S}_n} |S \cap \text{Subtree}(n)|}{b} + 1 \text{ (because } |\mathcal{S}_n| \text{ is an integer)} \\
\Leftrightarrow &\forall n, \sum_{S\in\mathcal{S}_n} |S \cap \text{Subtree}(n)| > b \cdot (|\mathcal{S}_n| - 1)
\end{aligned}$$

□

It is essential to understand that this condition is only able to tell whether a partition reaches the lower bound or not. However, there are cases where the lower bound is not reachable (hence where the condition is impossible to satisfy); in these cases, the condition will be unable to detect a solution which performs the minimal number of accesses. Figure 4.7 shows an example of such a situation: the block size is 3 and the lower bound is 12, but no partition (we the 3 possible types of partition) can reach it.

The most interesting aspect of this result is its structural interpretation: since the condition relies, for every node n of the tree, on blocks with encompassing trees which include n, the condition can be reinterpreted in terms of intersection of encompassing trees. We first derive an intuitive, weak sufficient condition: assume that for every node, say n, of the tree, $|S_n| = 0$ or 1. On the one hand, this means that n is a non-root node of 0 or 1 encompassing tree, and the root of a certain number of encompassing trees. Structurally, it is easy to check that this means that encompassing trees do not intersect, or that they intersect in one node (which is necessarily a node of a tree, and the root of all the other trees). On the other hand, this implies that the condition of the previous theorem is automatically satisfied. Therefore, we have proven that when encompassing trees do not intersect, or intersect only in one node, the lower-bound is reached.

Theorem 4.4 - *Structural sufficient condition.*
For a given partition, if any intersection of (two or more) encompassing trees of the sets of the

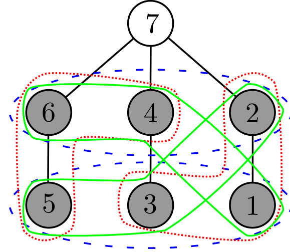


Figure 4.7: Example leading to an unreachable lower-bound. The 3 possible types of partitions are showed.

partition is either empty or a node, the number of accesses yielded by the partition is equal to the lower-bound.

The previous condition is sufficient but not necessary, since the intersections of encompassing trees can form, under a certain assumption, a path. The following result provides a structural necessary and sufficient condition:

Theorem 4.5 - *Structural necessary and sufficient condition.*
A partition reaches the lower bound if and only if any intersections of (two or more) encompassing trees of the sets of the partition is either empty, or a node, or a path such that its nodes respect the necessary and sufficient condition. The intersection cannot be a tree (with more than branch).

Proof. The different kinds of intersection are analyzed:

- Empty intersection: they correspond to nodes which are not non-root nodes of any encompassing tree (therefore $|\mathcal{S}_n| = 0$ and the condition is satisfied), or nodes which are a non-root node of only one encompassing tree (therefore $|\mathcal{S}_n| = 1$ and the condition is satisfied).
- If the intersection of k ($k \geqslant 2$) encompassing trees is a node, this node is necessarily the root of $k-1$ trees, otherwise the intersection is a path. Therefore, $|\mathcal{S}_n| = 1$ and the condition is satisfied.
- If the intersection of several encompassing trees is a path, the condition has to be checked. An interesting case is the situation where a path is the intersection of two encompassing trees (but no more): in this case, $|\mathcal{S}_n| = 2$, and it is easy to see that checking the condition at the lowest node of the path is enough.
- If the intersection of several encompassing trees is a tree, it is rather simple to show that at least one node which does not satisfy the condition can be found (e.g., in the case where the intersection of two, but no more, encompassing trees is a tree, there is a leaf l such that the condition is not satisfied).

□

Figure 4.8 shows an example of application of the previous result: $a^{-1}_{3,3}$, $a^{-1}_{4,4}$, $a^{-1}_{10,10}$, $a^{-1}_{11,11}$, $a^{-1}_{12,12}$ and $a^{-1}_{13,13}$ are requested. The blocks are represented in red, small dashes, and the encompassing trees are represented in green, larger dashes. In this example, the condition is satisfied, because : $T^e_{\{3,4\}} \cap T^e_{\{10,11\}} = \varnothing, T^e_{\{3,4\}} \cap T^e_{\{12,13\}} = \varnothing$, and $T^e_{\{3,4\}} \cap T^e_{\{10,11\}} = \{12\}$.
This property can be simply extended to the general case by applying it to the row and column subscripts of the requested entries:

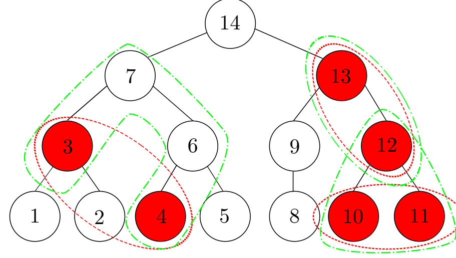


Figure 4.8: Example of application of the necessary and sufficient condition (diagonal case) where the partition is optimal; blocks are represented in red, small dashes, and encompassing trees with green, larger dashes.

Theorem 4.6 - *Necessary and sufficient condition for reaching the lower bound (general case).* Let $\Pi = \{S_1, \ldots, S_p\}$ a partition of the requested entries (p blocks of size b). The number of accesses achieved when processing the right-hand sides according to Π is equal to the lower bound of Theorem 4.1 if and only if the partition induced by the row subscripts of the elements of Π and the partition induced by the column subscripts of Π respect condition 4.3.

Proof. The result is clear since the lower bound in the general case is the sum of the lower bound for the row subscripts and the lower bound for the column subscripts.

Remark: in the general case, a row subscript can be associated to several requested entries. Hence, the corresponding node has $\text{req}(i) > 1$, and the previous proof has to be slightly adapted: $|S \cap \text{ Subtree}(n)|$ becomes $\sum_{s \in S \cap \text{Subtree}(n)} \text{req}(s)$, because counting nodes is not enough anymore (the number of times they correspond to requested entries, "req", has to be taken into account). $\square$

Remark: one has to understand that, in the general case, we do not consider a partition of the row subscripts and a partition of the column subscripts, but only a partition of the right-hand side, which induces the partitions of the subscripts. Therefore, if one chooses to build a partition according to the row subscripts, the column subscripts partition is automatically defined, and conversely. Hence, it is not possible to optimize the total cost by minimizing the number of accesses of the forward and the backward phases independently. This makes think that the condition is rather difficult to satisfy.

Figure 4.9 shows an example of application of Theorem 4.6. The requested entries are $a^{-1}_{1,3}$, $a^{-1}_{5,4}$, $a^{-1}_{4,10}$, $a^{-1}_{14,11}$, $a^{-1}_{9,12}$ and $a^{-1}_{8,13}$. The partition is : $\{\{a^{-1}_{1,3}, a^{-1}_{5,4}\}, \{a^{-1}_{4,10}, a^{-1}_{14,11}\}, \{a^{-1}_{9,12}, a^{-1}_{8,13}\}\}$; the row subscripts partition is $\{\{1,5\},\{4,14\},\{8,9\}\}$ (represented in blue, spaced dashes) and the column subscripts partition is $\{\{3,4\},\{10,11\},\{12,13\}\}$ (represented in red, small dashes). In this case, the condition is not satisfied because of the row subscripts partition: $T^e_{\{1,5\}} \cap T^e_{\{4,14\}} = \{6,7\}$, which is a path which does not respect the condition (6 is the lowest node, there are two nodes of $T^e_{\{1,5\}} \cap T^e_{\{4,14\}}$ in Subtree(6), which is greater or equal to the block size).

Heuristic construction

The previous result gives hints about the shape of an optimal solution. Here we present a simple heuristic for the diagonal case, which builds a partition of the requested entries with a traversal of the elimination tree: the idea is to try to gather nodes such that the sets form small trees. This will hopefully form a partition such that the encompassing trees of the different sets of the partition do not intersect, or intersect only in one node.

Algorithm 4.1 shows this heuristic. It collects the requested entries following a depth-first traversal[3], and relies on a preventive idea: if the current node in the traversal, say n, has ancestors

[3]In this case, it is equivalent to a reverse post-order, i.e. a pre-order.

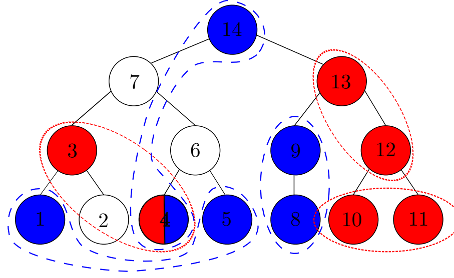


Figure 4.9: Example of application of the necessary and sufficient condition (general case) where the partition is non-optimal; blocks are represented in red, small dashes, and encompassing trees with green, larger dashes.

in the previous block S, then we try to gather the current node n, its ancestors and the nodes near n (potentially brothers, "nephews", etc.); we consider n', the highest ancestor of n:

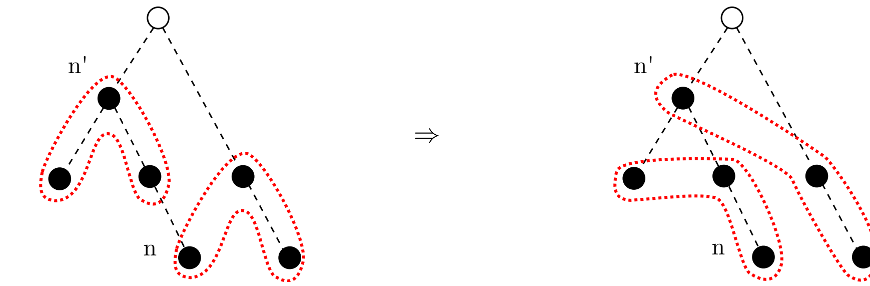
- **case I**: if there is no node in S outside Subtree(n'), then n is exchanged with n' (because, when a chain or a subtree is spread onto several blocks, it is generally better to put the lowest nodes together).
- **case II**: if there is a node n'' of S outside Subtree(n'), then it is exchanged with n, because $P(n)$ has more overlapping than $P(n'')$ with the paths from the other nodes of the previous block to the root node. If there are more than one node like n'', the tie-breaking rule is to choose the highest one (because it minimizes the overhead triggered if there is a chain of nodes which are not in Subtree(n')).

Thereby, if the requested nodes following n in the traversal are "far" from n, we avoided a potentially "large" intersection between the encompassing tree of the previous block and the encompassing tree of the block n would have belonged to. Figure 4.10 illustrates case I and case II. It is interesting to see that in case I, the intersection of the two encompassing trees obtained is reduced to a single node; in case II, the intersection is empty. Without the exchange, the intersection would have been a tree in both cases.

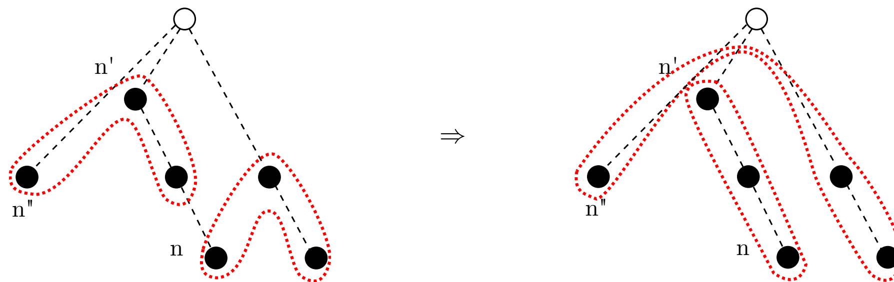
Algorithm 4.1 Heuristic construction of a solution.

```
1:  for all nodes n do {Depth-first traversal of the tree}
2:    Add n to the list of nodes.
3:    if n is a descendant of a node of the previous block S then
4:      n' = highest ancestor of n in S.
5:      if S\Subtree(n') = ∅ then
6:        Exchange n with n'. (case I)
7:      else {There is a node of S which is not below n'}
8:        n'' = highest node in S\Subtree(n').
9:        Exchange n and n''. (case II)
10:     end if
11:   end if
12: end for
```

Figure 4.11 shows an example of execution of this algorithm with a block size of 3: the elimination tree (with the requested entries in grey and the blocks given by the heuristic in red, small dashes) and the successive states of the list of nodes. In this example, the lower bound is 53. The partition given by the heuristic algorithm induces 55 accesses. It is simple to see that $\{\dots,\{5,6,7\},\{8,11,12\},\dots\}$ could be replaced by $\{\dots,\{6,7,8\},\{5,11,12\},\dots\}$, thus saving an access to 8 and 9 and reaching the lower bound.



(a) Illustration of case I: n is exchanged with its highest ancestor in the previous block, n'.



(b) Illustration of case II: n is exchanged with n'', highest node in $S\backslash\text{Subtree}(n')$.

Figure 4.10: Illustration of the different cases of Algorithm 4.1.

The main objective for further research is to extend this heuristics to the general case (i.e. not restricted to diagonal entries). The intuition is that performing our heuristics on the column subscripts and embed information on the row subscripts to perform exchanges should be a good idea.

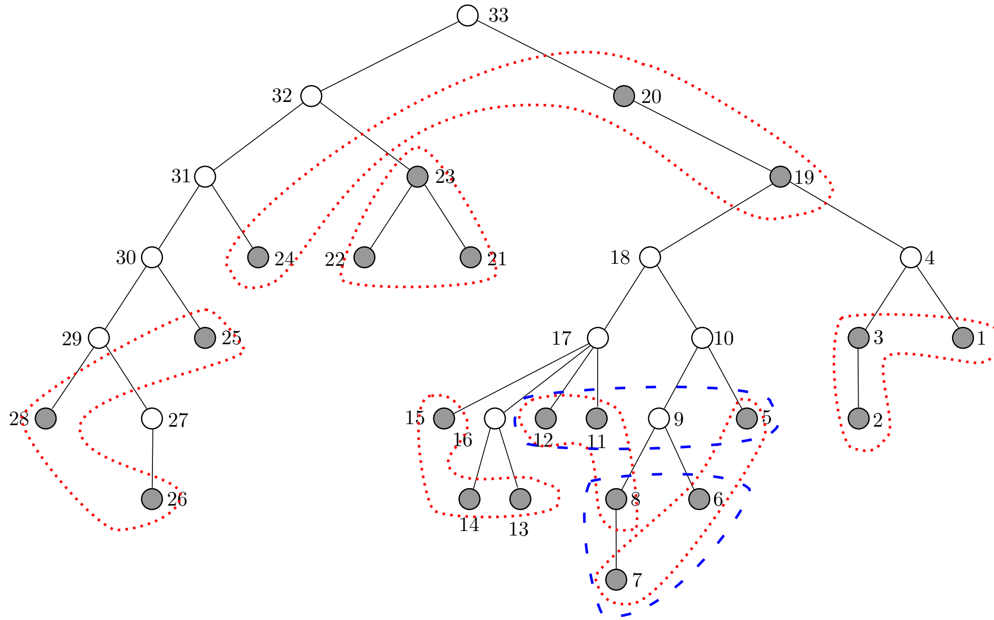
4.4 A local approach: improvement of a given order

Motivation

The structural example showed that a post-order could perform poorly because of nodes which belong to the same branch of the tree but that are split by the post-order. This is also confirmed by the structural properties exposed above. In this section, we want to improve a given order as follows: the main idea is to perform exchanges between two blocks such that nodes which belong to the same branch of the tree but in different blocks are reassembled. We thus have a local improvement strategy which can guide a heuristic algorithm:

1: List all configurations such that nodes on the same branch are split onto several blocks
2: **while** the list is non-empty **do**
3:    Select a configuration in the list
4:    Perform a local improvement
5:    Update the list
6: **end while**

Of course, there is no guarantee that this algorithm will converge, or will provide an optimal solution, but hopefully, the initial partition (a post-order for example) will be improved.



(a) Example of result of the heuristic construction.

28
28 26
28 26 25 |
28 26 25 | 24
28 26 25 | 24 23
28 26 25 | 24 23 22 |
28 26 25 | 24 23 22 | 21 — $21 \leftrightarrow 24$ (case II)
28 26 25 | 21 23 22 | 24 20
28 26 25 | 21 23 22 | 24 20 19 |
28 26 25 | 21 23 22 | 24 20 19 | 15 — $15 \leftrightarrow 24$ (case II)
28 26 25 | 21 23 22 | 15 20 19 | 24 14 — $14 \leftrightarrow 20$ (case I)
28 26 25 | 21 23 22 | 15 14 19 | 24 20 13 | — $13 \leftrightarrow 19$ (case I)
28 26 25 | 21 23 22 | 15 14 13 | 24 20 19 | 12 — $12 \leftrightarrow 24$ (case II)
28 26 25 | 21 23 22 | 15 14 13 | 12 20 19 | 24 11 — $11 \leftrightarrow 20$ (case I)
28 26 25 | 21 23 22 | 15 14 13 | 12 11 19 | 24 20 8 | — $8 \leftrightarrow 19$ (case I)
28 26 25 | 21 23 22 | 15 14 13 | 12 11 8 | 24 20 19 | 7 — $7 \leftrightarrow 24$ (case II)
28 26 25 | 21 23 22 | 15 14 13 | 12 11 8 | 7 20 19 | 24 6 — $6 \leftrightarrow 20$ (case I)
28 26 25 | 21 23 22 | 15 14 13 | 12 11 8 | 7 6 19 | 24 20 5 | — $5 \leftrightarrow 19$ (case I)
28 26 25 | 21 23 22 | 15 14 13 | 12 11 8 | 7 6 5 | 24 20 19 | 3 — $3 \leftrightarrow 24$ (case II)
28 26 25 | 21 23 22 | 15 14 13 | 12 11 8 | 7 6 5 | 3 20 19 | 24 2 — $2 \leftrightarrow 20$ (case I)
28 26 25 | 21 23 22 | 15 14 13 | 12 11 8 | 7 6 5 | 3 2 19 | 24 20 1 | — $1 \leftrightarrow 19$ (case I)
28 26 25 | 21 23 22 | 15 14 13 | 12 11 8 | 7 6 5 | 3 2 1 | 24 20 19 |

(b) Successive states of the partition.

Figure 4.11: Example of application of the heuristic construction.

Local improvement

Here we only provide the main issues guiding the local improvement strategy. Assume that there exist two nodes, i and j, such that $P(j) \subset P(i)$ (i.e. j is on the path between i and the root node)

and which belong to different blocks of an initial partition of the right-hand sides. We say that the block containing j is the "upper block", and the block i belongs to is called the "lower block". The first idea is to try to transfer one of these nodes from one block to another, so that they are reassembled. However, the constraint on the number of nodes per blocks has to be satisfied: therefore, a node has to be given "in return" from the block which received the node to the other.

We first provide a result which indicates if transferring a node from one block to the other induces a positive gain, without considering the problem of giving a node in return:

Theorem 4.7
Transferring a requested node from one block to the other induces a positive gain if and only if this node is the only one of its block which belongs to Subtree(j).

Figure 4.12 illustrates the previous result: we choose $i = 4, j = 11$. Subtree(11) is represented with green, large dashes.

- Is it interesting to perform a transfer from the lower block to the upper block ? 4 is not the only node of the lower in Subtree(11), therefore such a transfer would generate an overhead. Without 2 and 5 (i.e. if they were not requested or were in another block), transferring would be interesting.

- Is it interesting to perform a transfer from the upper block to the lower block ? 11 is not the only node of the upper block in Subtree(11), therefore such a transfer would generate an overhead. Without 8, it would not be interesting either, because of 10; however, without 8, transferring 10 would be interesting.

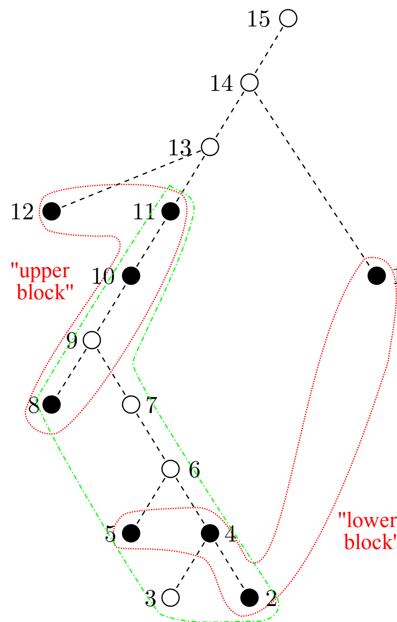


Figure 4.12: Illustration of a single exchange.

If one chooses to transfer a node from one block to the other, a node has to be given in return. We provide some rules for the selection of this node; for the sake of simplicity we assume that i was transferred from the lower block to the upper block, but the following results can be easily transformed to fit the other case.

Theorem 4.8 - *Selection of a node in return.*
The selection of a node is guided by the following results:

1. There is an *isolated node* in the lower block, i.e. a node k such that there is no other node of the lower block on $P(k)$ and Subtree(k). Such nodes can be given in return “for free” (i.e. there is no overhead).

2. Otherwise:

   a) If the lower block is included in Subtree(j), there is no gain possible (i.e. any transfer would generate an overhead). We call such a situation a “locked block”.

   b) Otherwise, there are nodes of the lower block outside Subtree(j), but they are not isolated. Hence, nodes of another branch will be split. In this case, the node given in return is the highest node of the lower block Subtree(j), so that the overhead generated is minimized. This overhead has to be compared with the gain obtained by transferring i to determine if the whole process is interesting.

An extension: multiple exchanges

Theorem 4.7 shows that transferring only one node at a time is very restrictive, because this a node cannot be transferred if there is another node of its block Subtree(j). Thus, a natural idea is to transfer more than one node at the same time: for instance, transferring a node along with all the nodes of its block which belong to Subtree(j) seems to be good way to circumvent the previous problem. We state the following result:

Theorem 4.9
A transfer from one block to the other induces a positive gain if and only if all the nodes of this block which belong to Subtree(j) are transferred at the same time.

Then, |Subtree(i)| nodes have to be given in return, and the previous rules can be extended.

These ideas describe the main components of this heuristics (a sequence of local improvements). Finally, what has to be noticed is the similarity between this heuristics and the constructive algorithm: if the initial guess of our heuristics is a depth-first search of the tree, then both algorithms look very close, because they try to gather nodes with a descent relation. The heuristics presented here is more general and seems more robust than the constructive algorithm, because it performs checks before exchanging nodes (and guarantees local gains). Further research has to be done, but the idea of local improvement seems promising, since a few exchanges are sometimes sufficient to improve heavily a given partition.

4.5 Perspectives

We have presented different approaches to tackle our combinatorial problem, with various points of view. A few questions remain open and several extensions could be considered:

- In-core case: as said above, the metrics associated to the in-core case are completely different, and it could be interesting to address this problem and to see how both problems compare.

- Multiple entries per column: if a column of the right-hand contains multiple entries, then this column is associated to several requested entries, but these entries are *necessarily processed at the same time*. Therefore, one cannot only settle for adding these entries in the previous models, since this constraint would not be taken into account (the entries could be spread into different blocks).

- Hypergraph implementation: the hypergraph representation presented above is interesting since it makes it possible to reduce our application to a rather well-known problem, with

robust software packages able to handle it. Unfortunately, our hypergraph is large (with many nets and connections), and partitioning could be intractable for large problems. The fact that our hypergraph has a very peculiar structure (nets are included in one another) could be profitable.

- Compressed representations of the problem: the necessary and sufficient condition presented above shows how a good partitioning of the right-hand sides should look-like, and emphasizes the fact that it is interesting to have partitions which form small subtrees. This information could be injected in the hypergraph model: entries forming small subtrees of the good size could be merely removed from the model, and a group smaller than the block size could be collapsed into a "super-entry".

- Flexible block size: in our study, the block size is given; this corresponds to a regular way of using direct solvers, but this is a significant constraint for our problem. Some examples show that sometimes, adding a last node to a block is "the straw that broke the camel's back": sometimes, a single node of a block can generate a large overhead because this node shares pieces of paths to the root with requested nodes of other blocks, and transferring this node to another block could cancel this effect. Therefore, allowing some flexibility for the block size could be interesting.

- Parallel execution: in parallel environment, one has to give work to each processor. A simple interleaving algorithm, based on the subtrees built during the mapping phase of the analysis, is proposed in [29], and could be improved or extended.

Part II

Experimental study

Chapter 5

Context of the experimental study

In this section, we focus on the impact of our work on the performance of the application developed by the SPI team at CESR. We recall the main and most costly steps of the application:

1: Compute $A = B^T B$
2: **for** i=1 to 10 **do**
3: Solve $z = Ax$
4: Update B
5: Update A
6: **end for**
7: Compute $diag(A^{-1})$

We first provide some preliminary results which were useful to determine the cost of each step of the global application. We then highlight some problems, suggest some corrections and show that computing the diagonal of the inverse of A is indeed the most expensive part of the application, hence justifying the need for our study.

5.1 First experiments

Preliminary results

We first present some preliminary results provided by the SPI team which illustrate the cost of the different steps of the computation. These tests were run on the "Integral-5" machine at CESR (4 dual-core Opteron processors at 2.4 GHz, with 32 GB of memory[1]). Matrix B has 688,201 rows, 66,522 columns, and 43,940,234 non-zero elements; $A = B^T B$ has 66,522 rows and columns, and 5,904,797 non-zero elements (see the introduction for the definition of matrices A and B). The update phase modifies the first 5,782 columns of B at each iteration. The computation of $diag(A^{-1})$ is done with `MUMPS`, by solving every $Ax = e_i, i \in [\![1, n]\!]$ and selecting the requested entries. These preliminary results are reported in Table 5.1.

Operation	Time (s)
Computation of $A = B^T B$	25,000
Solution of $Ax = b$	1,300
Update of A	30
Computation of $diag(A^{-1})$	6,000
Total time	45,000

Table 5.1: Preliminary tests on Integral-5 computer.

[1] Only one processor is used.

Observations

These results were startling for several reasons: first, the time for computing $B^T B$ is surprisingly high: even if B is quite large, it is also very sparse (its density[2] is approximately 0.001), so that we think that sparsity was not correctly exploited in this computation. Secondly, the solution time with MUMPS is surprisingly high as well. Finally, the computation of the diagonal of the inverse of the matrix represents a minor part of the overall time while it is the most computationally intensive part of the application from far; one should thus expect that it takes an important proportion of the computation time.

We show in the next section that using more efficient algorithms and tuning the codes, one can obtain significant improvements leading to timings in agreement with the complexity of the computations.

5.2 Simple improvements

Improvement of the computation of $B^T B$

The algorithm of computation of $B^T B$ (as well as the algorithm to update $B^T B$, which is a simple restriction of the first one) was a homemade algorithm developed by the SPI team. We decided to replace it with an adaptation of an algorithm from the MA49 package: this package by C. Puglisi [28] implements a multifrontal QR factorization.

MA49 relies on a property which states that, for any matrix, say A, the Cholesky factor L^T of $A^T A$ and the R factor of the QR factorization of A are the same[3]; this property is used for the symbolic factorization (computation of the structure of R), and therefore MA49 provides a routine to compute the structure of $A^T A$ from the structure of A. We have extended this routine so that it also computes the numerical values of $A^T A$.

We show some test results (the tests were carried on the "Tolosa" machine at IRIT (one 1.8 GHz core used), for both the computation of $B^T B$ and its update (after the modification of 5000 columns of B).

Algorithm	Time (s)
Initial algorithm	$\sim 60,000$
Modified MA49 routine	165

Table 5.2: Improvement of the computation of $A = B^T B$, on Tolosa computer.

Algorithm	Time (s)
Initial algorithm	~ 600
Modified MA49 routine	1.5

Table 5.3: Improvement of the update of $A = B^T B$, on Tolosa computer.

The gain is significant (improvement by a factor of 100). It shows the interest of using specialized libraries (dedicated to sparse matrix computations in this case).

Tuning of MUMPS

The bad results obtained during the solution phase (based on MUMPS solver) were due to a mistake in time measurement, and to a misuse of the solver:

[2]The density of an $m \times n$ matrix is defined by the ratio $\frac{nnz}{mn}$.

[3]If A has the *strong Hall property*.

- The analysis phase was done in all the iterations: since the structure of the matrix always remains the same, a single analysis (during the first iteration) could have been enough.
- Error analysis was enabled: this features provides numerical information about the initial matrix (estimated conditions numbers, infinite norm, ...) and the computed solution (residual, backward error, ...) [26]. This feature is very useful but expensive, and has to be disabled when performance is critical; furthermore the problem we address is not very tough numerically, thus it is not useful to request this analysis.
- Iterative refinement: this feature is designed to improve the accuracy of the computed solution by applying a few steps of an iterative procedure, starting from an initial solution given by the direct solver ([9]). In our case, this option was enabled (with 3 iterations requested) but not necessary (the stopping criterion was satisfied before starting the refinement, but this criterion is quite expensive to evaluate).

With a better timing of the software, the time for a solution with `MUMPS` decreased from 1300 seconds to approximately 10 seconds.

Chapter 6

Computation of the diagonal of the inverse

6.1 Experiments

In this section, we present a series of experiments related to the computation of the diagonal of the inverse of a matrix. We focus on matrices from the CESR: we show results for a small problem (of size 9000) and for the largest problem the CESR had to deal with (of size 148000).

We compare different methods and solvers:

- A "strict" traditional solution phase without pruning: here we simply solve every system $Ax = e_i$ without the improvements presented in 3.3 (pruning of the elimination tree). Two packages are compared: `MUMPS` and `MA57` [18] (multifrontal method for symmetric systems). With `MUMPS`, two block sizes of the right-hand sides have been tested (1 and 20).
- The approach relying on an $L\,D\,L^T$ factorization presented in 3.1. The package `LDSPARSE` [14] is used to compute the factorization. More details about the computation of the diagonal elements of the inverse are given in B
- `MUMPS` A^{-1} functionality, which uses a traditional solution phase with tree pruning as presented in 3.3.

6.2 Results

The results are presented in Table 6.1 and Table 6.2. "MUMPS (20 RHS)" and "MUMPS (1 RHS)" refer to a strict traditional solution (no pruning) using `MUMPS` with a block size of 20 and 1 respectively. "MA57 (20 RHS)" refers to a strict traditional solution phase using `MA57` with a block size of 20. "LDL" refers to the approach relying on an $L\,D\,L^T$ factorization using `LDSPARSE`. Finally, "MUMPS A^{-1}" refers of `MUMPS` A^{-1} functionality.

Solver	Time (s)
MUMPS (20 RHS)	43
MUMPS (1 RHS)	95
LDL	71
MUMPS A^{-1}	6

Table 6.1: Computation of the diagonal of a matrix of size 9000 from the CESR.

Note that on the largest problem, the factorization could not be completed by `LDSPARSE` because of numerical problems (this packages does not provide numerical pivoting). These results show very good performance of the approach presented in 3.3 over a strict traditional solution phase: pruning the elimination tree leads to clear gains. Note that the gain is even larger for the largest problem.

Solver	Time (s)
MUMPS (20 RHS)	11,000
MUMPS (1 RHS)	23,000
MA57 (20RHS)	13,000
MUMPS A^{-1}	760

Table 6.2: Computation of the diagonal of a matrix of size 148000 from the CESR.

The approach relying on an $L\,D\,L^T$ factorization shows reasonable performance for small problems, but could not be tested with larger matrices. Therefore, this approach could be interesting and usable if a package providing an $L\,D\,L^T$ with pivoting was used, but cannot really compete with a traditional solution phase with pruning anyway. Furthermore, as mentioned before, it is hard to generalize this approach for non diagonal computations or unsymmetric matrices.

Chapter 7

Influence of right-hand sides permutations

Here we present results related to the combinatorial problem of finding an optimal grouping of right-hand sides. We first report some experiments with topological orderings which motivated our study. We then compare the different solutions proposed: hypergraph model and constructive heuristics.

7.1 Topological orderings

The first experiments with topological orderings have been carried out in [29] in the context of the collaboration with the CESR. These experiments are presented in Table 7.1: randomly chosen 10% of the diagonal of A^{-1} are to be computed with 16 right-hand sides per block; the execution is sequential and MeTiS is used for ordering the matrix). This table presents the ratio between the size of the loaded factors and the lower bound mentioned above. The "no ES" column corresponds to the case where the sparsity of the right-hand sides is not exploited (i.e. using a strict traditional solution, without pruning). The next three columns ("Nat" , "Po", and "Pr") correspond to the partitioning of the right-hand sides by first sorting the requested entries with, respectively, the natural order, a post-order, and a pre-order (reverse of the order used for "Po"), and then by dividing the sorted subscripts into blocks of size 16. One can notice the clear speed-up obtained when sparsity is exploited, and the better performance of the three partitioning alternatives compared to the one based on natural order.

Matrix size	Ratio: factors loaded/minimum			
	no ES	with ES		
		Nat	Po	Pr
21,572	8.91	3.24	1.01	1.02
46,799	11.65	1.32	1.10	1.10
72,358	191.81	3.22	1.11	1.11
148,286	260.81	3.69	1.09	1.09

Table 7.1: Tests of topological orderings: computing random 10% of the diagonal entries of the inverse of a set of test matrices.

The experiments described above show that the two topological orderings (pre/post-order) lead to near optimal results on this set of matrices (ratios are around 1.1) but one can wonder if this is always the case. We show in Table 7.2 some experiments on a set of general-purpose matrices, from various applications (these matrices are described in Table 7.5, at the end of this chapter): first, computation of random 10% entries in the diagonal of A^{-1}; second, random 10% entries (diagonal or off-diagonal) in A^{-1}. The experimental setting is the same as before. We notice that on most matrices, the topological orderings lead to poorer performance than reported in Table 7.1.

Matrix	10% diagonal	10% off-diag
CESR(46799)	1.01	1.28
af23560	1.02	2.09
boyd1	1.03	1.92
ecl32	1.01	2.31
gre1107	1.17	1.89
saylr4	1.06	1.92
sherman3	1.04	2.51
grund/bayer07	1.05	1.96
mathworks/pd	1.09	2.10
stokes64	1.05	2.35

Table 7.2: Efficiency of post-order based partitioning of the right-hand sides (ratio of factors loaded/lower bound) on some matrices from University of Florida collection.

These results show that a permutation based on a post-order traversal of the tree provides interesting result in the diagonal case; however, important factors (between 1.5 and 3 for most problems) can be potentially gained in the general case by improving the way the right-hand sides are grouped. This observation has motivated our study.

7.2 Hypergraph model

Here we report some experiments carried out in [29], where the hypergraph model is tested with the matrices from the CESR, where random 10% entries of the diagonal of the inverse are computed. The partitioner used is PaToH [13]. In Table 7.3, the hypergraph model (column "HG") is compared to the post-order (column "Po").

Matrix size	Ratio: factors loaded/ minimum	
	Po	HG
21,572	1.01	1.01
46,799	1.10	1.01
72,358	1.11	1.00
148,286	1.09	1.01

Table 7.3: Tests of the hypergraph model: computing random 10% of the diagonal entries of the inverse of a set of test matrices.

One can notice the good performances of the hypergraph model, which provides a better permutation than the post-order, and nearly reaches the lower bound for all the problems. Even if, as said before, this particular set of problems from the CESR is rather "easy" (since topological orders perform quite well), these results are promising. Further research and experiments have to be made to improve the representation and implementation of this hypergraph model, especially in the general (non-diagonal) case.

7.3 Constructive heuristics

The constructive heuristics described in 4.3 has been implemented so that it provides `MUMPS` with a partition of the right-hand sides used in the solve phase, and has been tested with the set of

matrices described in Table 7.5, where random 10% diagonal entries of the inverse are requested. The results obtained are reported in Table 7.4.

Matrix	Efficiency
CESR(46799)	1.01
af2356	1.01
boyd1	1.03
ecl32	1.01
gre1107	1.09
saylr4	1.08
sherman3	1.05
grund/bayer07	1.07
mathworks/pd	1.06
stokes64	1.04

Table 7.4: Efficiency of heuristics based partitioning (ratio of factors loaded/lower bound) on some matrices from University of Florida collection. Random 10% diagonal entries are requested.

One can notice that this heuristic algorithm provides very good results for this set of problems, since the numbers of accesses to the factors never exceeds 10% of the lower bound; however, a permutation based on a post-order provides nearly the same results, but is less expensive to obtain (since a post-order of the elimination tree is already constructed during the analysis phase of the solver). Therefore, our main objective is to extend this approach to the diagonal, by modifying the algorithm so that it works with information on the rows and columns of the right-hand sides at the same time. Furthermore, work has to be done to evaluate precisely the cost of this heuristics and propose efficient implementations.

Matrix	Size	Number of non-zeros	Symmetry	Field of application
CESR(46799)	46,799	3,582,484	Symmetric	Astrophysics
af2356	23,560	484,256	Unsymmetric	Fluid Dynamics
boyd1	93,279	1,211,231	Symmetric	Numerical Optimization
ecl32	51,993	380,415	Unsymmetric	Semiconductor Simulation
gre1107	1,107	5,664	Unsymmetric	Computer Systems
saylr4	3,564	22,316	Unsymmetric	Petroleum Engineering
sherman3	5,005	20,033	Unsymmetric	Reservoir Simulation
grund/bayer07	3,268	20,963	Unsymmetric	Chemical Simulation
mathworks/pd	8,036	13,036	Unsymmetric	Linear Algebra
stokes64	12,546	140,034	Symmetric	Fluid Dynamics

Table 7.5: Set of matrices used for the experiments.

Conclusion

A motivation for this study was the collaboration with the SPI team at CESR: the aim was to understand their application globally and to improve the computation of the diagonal entries of the inverse of their large, sparse matrices. This objective has been fully reached, since gains of a factor of 15 have been obtained on the largest problem; furthermore, the whole application has been significantly improved as well. The solutions proposed ensure improved robustness, especially from a numerical point of view, and thanks to the fact that they rely on well-known and actively maintained packages, such as `MUMPS` and `MA49` for instance.

Our first contribution was to compare the behaviour of the different solutions for computing a set of entries of the inverse of a large, sparse matrix, and to experiment them in a real production code from an application in astrophysics. During this study, the problem of finding an optimal permutation of right-hand sides has been tackled. This new combinatorial problem is an interesting extension of previous works like [29], and we have shown that significant gains can be expected. Even if much work still has to be done (see below), the theoretical work proposed here is already valuable and has led to practical algorithms.

Many questions still remain open, and many trails can be explored in order to extend this study. These potential improvements have been suggested in the previous sections (especially 4.5), and we remind the most significant ones:

- In-core case: study the influence of right-hand sides groupings in this context, and study the difference with the out-of-core case.
- Parallel execution: extend what has been proposed in [29].
- Extend the different models and heuristics proposed in this study, and especially compact representations of these models.
- In the case of the computation of the diagonal elements of the inverse of a sparse, symmetric matrix, spare a solution with L by solving $Lx = e_i$ instead of $L\,D\,L^T x = e_i$ (theoretically, this leads to gaining a factor of 2).

Appendix A

More details about elimination and assembly trees

A.1 Construction of the elimination tree

In this section, we provide information about the construction of elimination trees. Several points of view, which highlight different properties of elimination trees (dependencies, structure of the factors, compactedness,...), can be adopted. All this information can be found in a survey of Liu [25], and we report only a few important aspects, which are critical to understand the role and expressivity of elimination trees in direct methods.

A first definition

We consider the case where A is positive definite; thus, a Cholesky factorization can be applied: $A = LL^T$. We consider $F = L + L^T$ the *filled matrix* of A, and its undirected graph $G(F)$.

Definition A.1 - *Elimination tree.*
Consider the L factor of A; for all $j < n$, remove all the nonzeros in this column except the first nonzero below the diagonal[1]. Let L_t be the resulting matrix and $F_t = L_t + L_t^T$. $G(F_t)$ has a tree structure, and is called the *elimination tree* of A, noted $T(A)$.

Properties:

- $T(A)$ has the same node-set as $G(A)$ and is a spanning tree of $G(F)$.
- x_n is the root of $T(A)$.

Column dependencies

The definition above does not seem very constructive at first sight, but is actually motivated by column dependencies, thanks the following proposition:

Lemma A.1
For $i > j$, the numerical values of column $L_{:,i}$ depend on column $L_{:,j}$ if and only if $l_{i,j} \neq 0$.

This result means that the filled graph $G(F)$ expresses the dependencies between the columns of the Cholesky factor L. One can notice that the graph contains redundant information: if there is a path of length greater than one between a node x_i and a node x_j, and an edge between them as well, this edge does not provide any information, since we already know that $L_{:,j}$ will depend

[1] Since A is positive definite, this is always possible.

on $L_{:,i}$. The graph can be compressed by suppressing all this redundant information. This process is known as *transitive reduction*, and one can show that the transitive reduction of $G(F)$ is $T(A)$.

Other constructions can be expressed; for example, $T(A)$ can be seen as a depth-first search tree of $G(F)$.

Structure of the factors

One of the main roles of the elimination tree is to determine the structure of the factors. We quote an important result:

Theorem A.1 - *Structure of the factors (theorem 3.5 in [25]).*
$l_{i,j} \neq 0$ if and only if the node x_j is an ancestor of some node x_k in the elimination tree, where $a_{i,k} \neq 0$.

Figure A.1 shows a simple illustration of theorem A.1: node 4 is an ancestor of 2 (actually its father), $a_{6,2} \neq 0$, hence $l_{6,4} \neq 0$.

$$\begin{pmatrix} 1 & & & & & \\ & 2 & & X & & X \\ & & 3 & & & \\ & X & & 4 & & \\ & & & & 5 & F \\ & X & & F & & 6 \end{pmatrix}$$

Figure A.1: Illustration of theorem A.1.

Thanks to this result, several algorithms can be developed to build the row or column structure of the factors, as well as the elimination tree itself.

A.2 Assembly trees

Motivation

Elimination trees are very efficient at expressing independent tasks, but these tasks are not efficient *per se*, because of their very small granularity. A major improvement is to create *blocked* computations, which are far more efficient: roughly speaking, processing a $n \times n$ dense matrix is more efficient than processing n^2 1×1 matrices, because it enables the use of dense kernels, such as BLAS [17], which are efficient, among other things, at taking advantage of cache memory. This blocking can be achieved thanks to the creation of *supernodes*, by grouping nodes of the elimination tree.

Supernodes and amalgamation

Supernodes (or *supervariables*) represent a group of columns of the factor L which share the same pattern (i.e. $\text{Struct}(L_{:,k}) = \text{Struct}(L_{:,k+1}) \cup \{k+1\}$). The corresponding nodes of the elimination tree are merged to create a so-called supernode. Processing this supernode consists in factorizing a dense matrix, and one can take advantage of high performance dense kernels. The tree obtained after the detection of supernodes in the elimination tree is called an assembly tree.

The previous constraint can be relaxed so that nodes that "nearly" share the same structure are amalgamated. This may generate some fill-in, and the number of operations and memory consumption are increased, but the whole process benefits from fast dense computations, because the size of the frontal matrices to factorize is increased.

A.3 Unsymmetric case

Here we briefly describe two ways of handling the case where the matrix is not symmetric: the first approach generalizes the concept of elimination tree, and the second one relies on a symmetrization of the matrix the factorize.

Extension of the symmetric model

Gilbert and Liu have extended their work to the unsymmetric case in [22]. In this case, the elimination tree, which was a transitive reduction of the dag of L^T, is replaced by two *elimination dags* (e-dags), which are the transitive reductions of the dags of L^T and U. Gilbert and Liu show how these two e-dags can be built simultaneously from one another and from the initial graph, and thus how to perform the symbolic factorization of the initial unsymmetric matrix. Some solvers like `SuperLU_DIST` [24], and `UMFPACK`[16] rely on the e-dag concept and properties.

Symmetrization

Some solvers, `MUMPS` in particular, choose to use a symmetric pattern to handle unsymmetric matrices: they work on the structure of the symmetrized matrix $A + A^T$ (each zero element of A which is nonzero in $A + A^T$ is replaced with a numerical zero value). Thus they can rely on the elimination tree of the symmetrized matrix. This has been generalized by Amestoy and Puglisi [7] to handle unsymmetric matrices introducing unsymmetric frontal matrices, but still relying on the “standard” elimination tree.

Appendix B

Computation of the diagonal of the inverse thanks to a factorization

Here we provide some details about the implementation of the computation of the diagonal of the inverse of a symmetric matrix A, based on an $L\,D\,L^T$ factorization. We first present two basic computational schemes, called *left-looking* and *right-looking* approaches, then we show how they can be applied to our problem, and finally we compare the different approaches (both in terms of complexity and practical performance).

B.1 Left-looking vs. right-looking approaches

Left-looking and *right-looking* are two computational schemes which can be applied to many algorithms; we apply them to the solution of a triangular system since it is directly related to our application. We then show how to apply such schemes to the factorization phase.

Generic scheme

We consider a group of objects (e.g. nodes of a dependency tree) which have computations to perform, with dependency relations between the computations. Left and right-looking schemes define the way the objects perform their work and communicate.

A generic left-looking scheme can be summarized as follows:

1. Each object gathers information from all the objects on which it depends,
2. Then it performs its computation.

Such an algorithm is also referred as "demand-driven", because the information needed to process a computation are gathered only when needed [23]. On the opposite, a right-looking scheme processes as follows:

1. Each object already has the information it needs and performs its computation,
2. Then it sends its contribution to the objects which depend on him.

This computational scheme is also referred as "data-driven", since each data is used as soon as it is computed to modify all the objects it affects.

Solution of a triangular system

Say we want to solve a lower triangular system $Lx = b$. We apply a forward substitution (the variables are computed from x(1)=b(1)/L(1,1) to x(n)). Algorithm B.1 and B.2 show the left and right-looking solutions respectively.

Algorithm B.1 Left-looking triangular solution $Lx = b$.

```
1: for i=1 to n do
2:    x(i)=( b(i) − Σ_{j=1}^{i−1} L(i,j)x(j) ) {x(i) gathers updates from previous variables}
3:    x(i)=x(i)/L(i,i)^{−1} {x(i) is then computed}
4: end for
```

Algorithm B.2 Right-looking triangular solution $Lx = b$.

```
1: for j=1 to n do
2:    x(j)=b(j)/L(j,j) {Computation of x_j}
3:    for i=j+1 to n do {x(j) contributes to the following variables}
4:       b(i)=b(i)-L(i,j)*x(j) {Right-looking update of the RHS}
5:    end for
6: end for
```

One can notice that the left-looking version accesses matrix L by rows, whereas the right-looking version accesses it by columns. Quite often, matrices are stored using a format which represent the matrix column by column; hence, accessing such a format by row is difficult, and performance generally collapses. However, in our application, both solutions are possible (see below).

Factorization phase

These schemes can also be applied to the factorization phase, and we compare them to the multifrontal method (we underline the different levels of communication in the elimination tree).

- *Left-looking scheme*: first, a node receives contributions from its descendants[1], then it processes its own computation.
- *Right-looking scheme*: a node processes its own computation, then it sends its contribution to its ancestors.
- *Multifrontal method*: first, a node of the tree receives contributions from its children, then it processes its own computation, and finally it send its contribution to its father.

Figure B.1 sums up these different approaches (for the matrix of Figure 2.1).

B.2 Implementation

We recall that the diagonal entries of the inverse matrix are computed as:

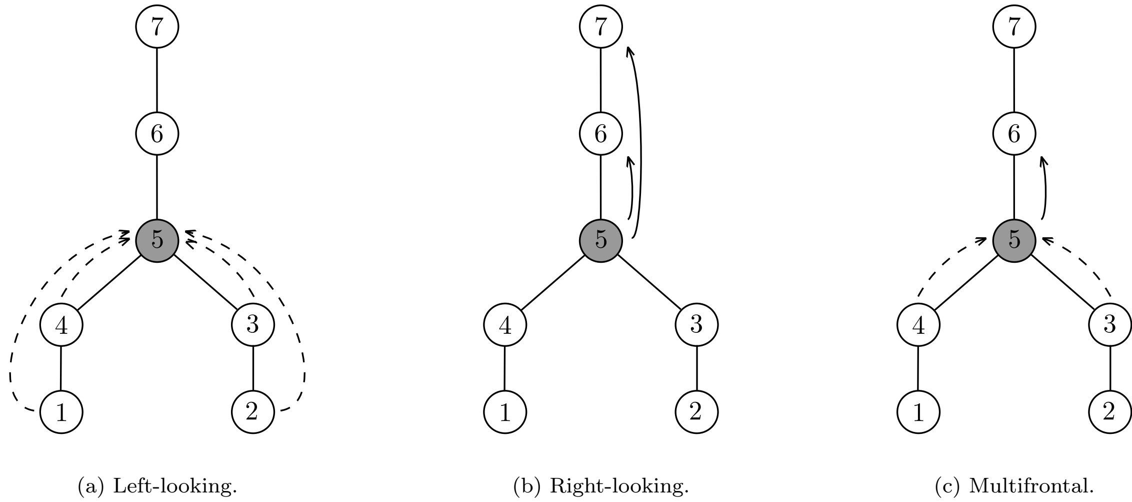
$$\forall i \in [\![1, n]\!], a_{i,i}^{-1} = \sum_{k=1}^{n} \frac{1}{d_{k,k}} {l_{k,i}^{-1}}^2$$

L^{-1} is computed thanks to a classic algorithm (which computes each column by solving a system $Lx = e_i$), which is described in [30] (and can be seen included in the next algorithm). In our application, L is stored with a column-oriented scheme; hence, when solving every system $Lx = b$, using a right-looking solution seems to be better suited.

Actually, a left-looking scheme can be used as well, by simply using the fact that $l_{i,j}^{-1} = l_{j,i}^{-T}$:

$$\forall i \in [\![1, n]\!], a_{i,i}^{-1} = \sum_{k=1}^{n} \frac{1}{d_{k,k}} {l_{i,k}^{-T}}^2$$

[1]Not every descendant though, only the ones which have an explicit dependency to the node.



(a) Left-looking. (b) Right-looking. (c) Multifrontal.

Figure B.1: Illustration of the different factorization schemes; node 5 is processed; dashed arrows represent contributions sent, and plain arrows contributions received.

Similarly, L^{-T} is computed by an algorithm which solves every system $L^T x = e_i$, but this time, the best choice is a left-looking approach: such an approach accesses the rows of L^T, hence its accesses L by columns, and therefore fits the storage scheme. However, this approach suffers a drawback: when solving $L^T x = e_i$, each component x_j is computed as $x_j = \sum_{k=1}^{i} l^T_{j,k} x_k$: only the i first components of the j-th line of L^T need to be accessed (this is illustrated in Figure B.2). If the lines of L^T (columns of L) are stored sorted, it is easy to"break" the loop which scans the lines so that it stops at $l^T_{j,i}$, but this cannot be assumed all the time [14], and in a case where the columns of L are not sorted, one has to access the whole lines of L^T. Even if the computation is still correct (these useless entries are multiplied by zero), it is easy to show that the numbers of floating-point operations is multiplied by 2 (see below).

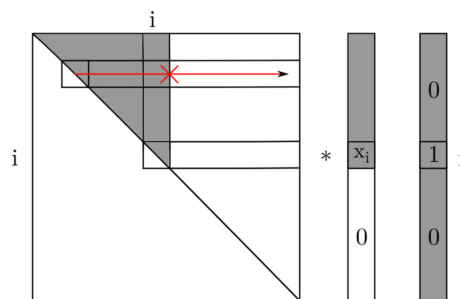
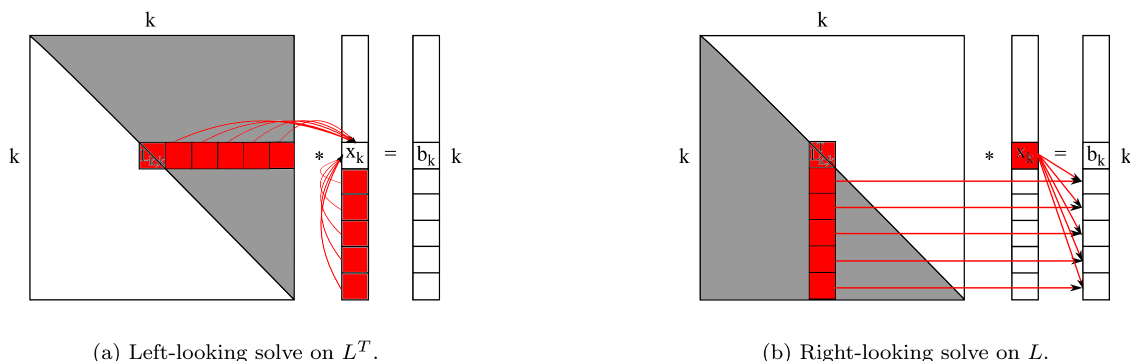


Figure B.2: Solution of $L^T x = e_i$.

Figure B.3(a) illustrates the left-looking solve performed on L^T and B.3(b) shows a right-looking solve performed on L. Algorithm B.3 and B.4 show the left and the right looking variants respectively. The storage format for L is the *compressed column form*: column j is stored in `Lx[Lp[j]...Lp[j+1]-1]`, and the corresponding row indices are `Li[Lp[j]...Lp[j+1]-1]`. One has to notice that the computation of the diagonal elements is different from one version to another: in the left-looking approach, all the elements are computed at the same time, by accumulating contributions of the columns of L^{-T}; in the right-looking approach, each element is computed in

one shot, and this can be useful if only a few entries of the diagonal are needed.



(a) Left-looking solve on L^T.

(b) Right-looking solve on L.

Figure B.3: Variants of implementation for the computation of the diagonal.

Algorithm B.3 Computation of the diagonal of A^{-1} (left-looking version).

```
1:  covar = 0
2:  for i=1 to n do {Computation of the i-th column of L^{-T} }
3:    x=0
4:    x(i)=1
5:    for j=i-1 to 1 by -1 do {Backward computation of the x_j}
6:      for k=Lp(j) to Lp(j+1)-1 do {x_j = ∑_k l^T_{j,k} x_k}
7:        if (Li(k)>i) then {To avoid to access the line to far}
8:          break
9:        end if
10:       x(j)=Y(j)-Lx(k)*x(Li(k))
11:     end for
12:   end for
13:   for j=1 to i do {We add the contribution of x on every a^{-1}_{j,j}}
14:     covar(j)=covar(j)+x(j)*x(j)/D(j)
15:   end for
16: end for
```

Algorithm B.4 Computation of the diagonal of A^{-1} (right-looking version).

```
1:  covar = 0
2:  for k=1 to n do {Compute the k-th column of L^{-1} solving Lx = e_k}
3:    x(k)=1
4:    x(k+1:n)=0
5:    for j=k to n do {Compute the x_j}
6:      x(j)=x(j)/Lx(j)
7:      for i=Lp(j) to Lp(j+1)-1 do {x(j) contributes to the following variables}
8:        x(Li(i))=x(Li(i))-Lx(i)*x(j) {Right-looking update of the RHS}
9:      end for
10:     covar(k)=covar(k)+x(j)*x(j)/D(j)
11:   end for
12: end for
```

B.3 Comparison

First, we show that both left and right-looking versions of the computation of the whole diagonal have the same complexity: the number of floating-points operations is proportional to the number of elements accessed in L (we ignore the computation of the covariances which requires the same number of operations in both algorithms):

- Left-looking version: it solves every system $L^T x = e_i$ by a backward substitution (from i=n to 1); for each one of these systems, the algorithm accesses rows i to 1 of L^T, and scanning each row j, leads to i-j accesses[2]. Hence, the number of accesses becomes:
$$\sum_{i=1}^{n}\sum_{j=1}^{i-1}(i-j) = \frac{1}{6}(n-1)n(n+1) = \Theta\left(\frac{n^3}{6}\right).$$
- Right-looking version: it solves every system $Lx = e_i$ by a forward substitution (from j=1 to n); for each one of these systems, the algorithm accesses columns i to n of L, and scanning each row, say j, leads to n-j accesses. Hence, the number of accesses is:
$$\sum_{i=1}^{n}\sum_{j=i}^{n}(n-j) = \frac{1}{6}(n-1)n(n+1) = \Theta\left(\frac{n^3}{6}\right).$$

Both versions have the same theoretical complexity; however, experiments show that they behave very differently. We show some tests performed on “Tolosa”, with a matrix of size 8999 from the CESR; Table B.1 shows the number of floating-point operations and the computation time for the different versions of the algorithm (“LL” stands for a left-looking version where the loop cannot be broken, “LL+stop” stands for a left-looking version where the loop is broken, and finally “RL” stands for a right-looking version).

Algorithm	Operations (G flops)	Time (s)
LL	6.29	2,919
LL+stop	2.34	2,928
RL	19.80	3,788

Table B.1: Tests of the different versions.

The “stopped” version of the left-looking algorithm performs well, but one can wonder why the right-looking algorithm performs so many operations, while we were expecting a decrease. Actually, the right-looking version suffers from the sparse structure of L: Figure B.4 shows the structure of A before and after reordering (based on AMD [3]), and the structure of the factor L. L contains a full block ($\sim 1400 \times 1400$), which slows down dramatically the right-looking algorithm. The left-looking version accesses this block (or subparts of this block) only 1400 times, whereas the right-looking variant accesses it n times (more precisely, it accesses $n-1400$ times the whole block, and 1400 times subparts of this block; cf. Figure B.5, which shows the sequences of accesses to the elements of L performed by the left and right-looking algorithms respectively). This behaviour explains the huge increase in the number of floating-point operations.

A few tests were also performed with Tim Davis' `LDSPARSE` package which provides an other implementation of the factorization and different settings for AMD. Figure B.6 shows the structures obtained with this new ordering. One can notice that the dense block has disappeared (this seems to be related to the numerical behaviour of Davis' LDL^T code). Table B.2 shows the results obtained with this new ordering; one can see that in this case, the right-looking behaviour is improved thanks to the reduction of the density of the last block, as explained before. However, the right-looking version is still slow, and the left-looking version (where the loop is stopped) is the method of choice.

[2] If the loop cannot be “broken”, there are n-j accesses, which finally implies a factor of 2.

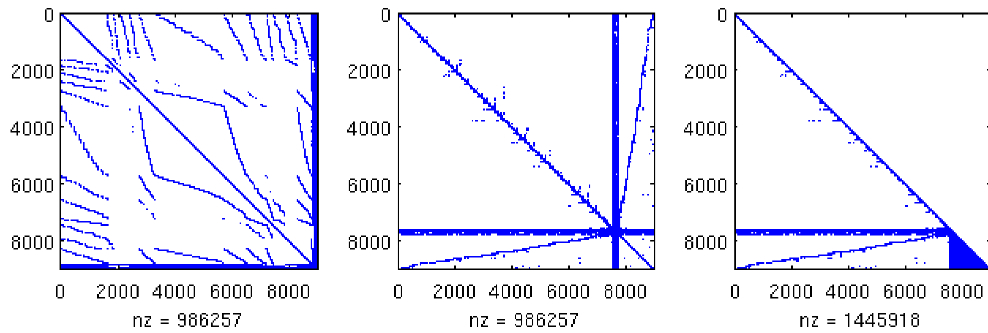
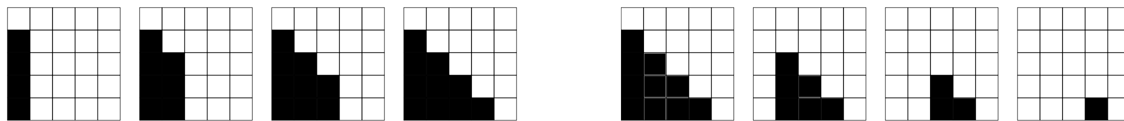


Figure B.4: Structures of A, A with reordering, and L.



(a) Accesses to L with the left-looking algorithm.

(b) Accesses to L with the right-looking algorithm.

Figure B.5: Comparison of the accesses performed by the different implementations.

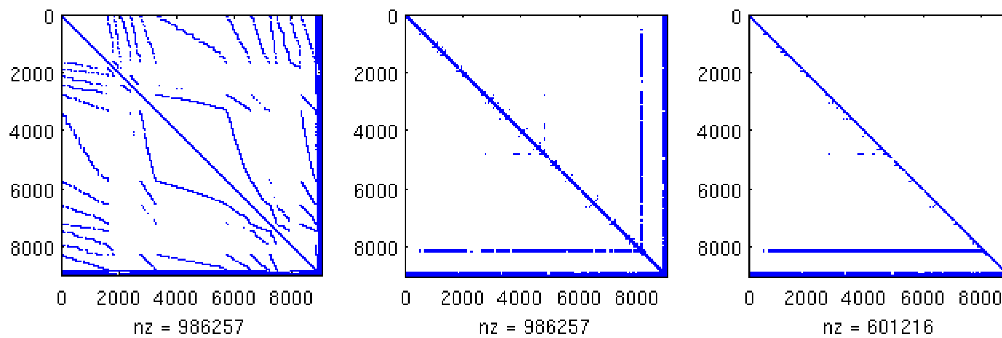


Figure B.6: Structures of A, A with new reordering, and L.

Algorithm	Operations (G flops)	Time (s)
LL	5.49	2,909
LL+stop	0.47	2,855
RL	5.57	3,788

Table B.2: Tests of the different versions with another ordering.

Bibliography

[1] E. Agullo, *On the out-of-core factorization of large sparse matrices*, PhD thesis, Ecole Normale Supérieure de Lyon, 2008.

[2] P. R. Amestoy, A. Buttari, Ph. Combes, A. Guermouche, J.-Y. L'Excellent, Tz. Slavova, and B. Uçar, *Overview of MUMPS (A MUltifrontal Massively Parallel Solver)*, June 2008. 2nd French-Japanese workshop on Petascale Applications, Algorithms and Programming (PAAP'08).

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

[4] P. R. Amestoy, I. Duff, Tz.. Slavova, and B. Uçar, *Out-of-core solution for singleton right-hand side vectors.* Presentation at SIAM Conference on Computational Science and Engineering (CSE09), March 2009.

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

[6] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent, *Multifrontal parallel distributed symmetric and unsymmetric solvers*, Computer Methods in Applied Mechanics and Engineering, 184 (2000), pp. 501–520.

[7] P. R. Amestoy and C. Puglisi, *An unsymmetrized multifrontal LU factorization*, SIAM Journal on Matrix Analysis and Applications, 24 (2003), pp. 553–569.

[8] P. R. Amestoy, F.-H. Rouet, and B. Uçar, *On computing arbitrary entries of the inverse of a matrix.* Presentation at SIAM Conference on Combinatorial Scientific Computing, 2009.

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

[10] Å. Björck, *Numerical methods for least squares problems*, Society for Industrial Mathematics, 1996.

[11] L. Bouchet, J.-P. Roques, P. Mandrou, A. Strong, R. Diehl, F. Lebrun, and R. Terrier, *INTEGRAL SPI observation of the galactic central radian: contribution of discrete sources and implication for the diffuse emission 1*, The Astrophysical Journal, 635 (2005), pp. 1103–1115.

[12] Y. E. Campbell and T. A. Davis, *Computing the sparse inverse subset: an inverse multifrontal approach*, Tech. Rep. TR-95-021, CIS Dept., Univ. of Florida, 1995.

[13] UV. Çatalyurek and C. Aykanat, *PaToH: partitioning tool for hypergraphs*, User's guide, (1999).

[14] T. A. Davis, *User Guide for LDL, a concise sparse Cholesky package*, tech. rep., Technical Report, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA, 2005, 2007.

[15] ——, *The "factorize object" for solving linear systems.* University of Florida, 2009.

[16] T. A. Davis and I. S. Duff, *An unsymmetric-pattern multifrontal method for sparse LU factorization*, Citeseer, 1993.

[17] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, *An extended set of FORTRAN basic linear algebra subprograms*, ACM Transactions on Mathematical Software, 14 (1988), pp. 1–17.

[18] I. S. Duff, *MA57 - A code for the solution of sparse symmetric definite and indefinite systems*, ACM Transactions on Mathematical Software, 30 (2004), pp. 118–144.

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

[20] ——, *The multifrontal solution of unsymmetric sets of linear systems*, SIAM Journal on Scientific and Statistical Computing, 5 (1984), pp. 633–641.

[21] A. M. Erisman and W. F. Tinney, *On computing certain elements of the inverse of a sparse matrix*, Comm. ACM, 18 (1975), pp. 177–179.

[22] J. R. Gilbert and J. W. H. Liu, *Elimination structures for unsymmetric sparse LU factors*, SIAM J. Matrix Analysis and Applications, (1993).

[23] M. T. Heath, E. Ng, and B. W. Peyton, *Parallel algorithms for sparse linear systems*, SIAM review, (1991), pp. 420–460.

[24] X. S. Li and J. W. Demmel, *SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Trans. Mathematical Software, 29 (2003), pp. 110–140.

[25] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), p. 134.

[26] MUMPS, *MUltifrontal Massively Parallel Solver (MUMPS version 4.9), user's guide*, 2009.

[27] A. Pothen, *Graph partitioning algorithms with applications to scientific computing*, ICASE LaRC Interdisciplinary Series in Science and Engineering, 4 (1997), pp. 323–368.

[28] C. Puglisi, *QR factorization of large sparse overdetermined and square matrices with the multifrontal method in a multiprocessing environment*, PhD thesis, Institut National Polytechnique de Toulouse, Toulouse, France, 1993.

[29] Tz. Slavova, *Parallel triangular solution in an out-of-core multifrontal approach for solving large sparse linear systems*, PhD thesis, Institut National Polytechnique de Toulouse, Toulouse, France, 2009.

[30] G. W. Stewart, *Matrix algorithms*, Society for Industrial and Applied Mathematics, 1998.

[31] K. Takahashi, J. Fagan, and M. S. Chen, *Formation of a sparse bus impedance matrix and its application to short circuit study*, in Power Industry Computer Applications Conference, 1973, pp. 63–69.

[32] B. Uçar and C. Aykanat, *Revisiting hypergraph models for sparse matrix partitioning*, SIAM Review, 49 (2007), pp. 595–603.