

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *INP Toulouse*
Discipline ou spécialité : *INFORMATIQUE*

Présentée et soutenue par *Tzvetomila Slavova*
Le *28 Avril 2009*

Titre : *Résolution triangulaire de systèmes linéaires creux de grande taille dans un contexte parallèle multifrontal et hors-mémoire.*

Parallel triangular solution in the out-of-core multifrontal approach for solving large sparse linear systems.

JURY

Amestoy P.R	*Professeur, INPT,Toulouse*	*Directeur de thèse*
Duff,I.	*Directeur de recherche, CERFACS, Toulouse*	*Co-encadrant*
Guermouche, A.	*LaBRI, Univ. Bordeaux 1 / INRIA Futurs*	*Co-encadrant*
L'Excellent,J-Y.	*Chargé de recherche, INRIA-LIP, Lyon*	*Membre*
Ng, E. G.	*Directeur de recherche, Lawrence Berkeley Lab.*	*Rapporteur*
Trystram, D	*Professeur, INPG, Grenoble*	*Rapporteur*
Ucar, B.	*Chargé de recherche CNRS, LIP, Lyon*	*Membre*

Ecole doctorale : *Mathématiques, Informatique et Télécommunications de Toulouse*
Unité de recherche : *CERFACS*
Directeur de Thèse : *Amestoy, P. R.*

THÈSE

présentée pour obtenir

LE TITRE DE DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

Spécialité: INFORMATIQUE

par

Tzvetomila Slavova
CERFACS

Résolution triangulaire de systèmes linéaires creux de grande taille dans un contexte parallèle multifrontal et hors-mémoire.

Parallel triangular solution in the out-of-core multifrontal approach for solving large sparse linear systems

---

Thèse présentée le 28 Avril 2009 devant le jury composé de:

Amestoy, P. R.	Professeur, INPT, Toulouse	Directeur de thèse
Duff , I.	Directeur de recherche, CERFACS, Toulouse	Co-encadrant
Guermouche, A.	LaBRI, Univ. Bordeaux 1 / INRIA Futurs	Co-encadrant
L'Excellent, J-Y.	Chargé de recherche, INRIA-LIP, Lyon	Membre
Ng, E. G.	Directeur de recherche, Lawrence Berkeley Lab.	Rapporteur
Trystram, D.	Professeur, INPG, Grenoble	Rapporteur
Ucar, B.	Chargé de recherche CNRS, LIP, Lyon	Membre

Thèse préparée au CERFACS, CERFACS Report Ref: TH-PA-09-59

Résolution triangulaire de systèmes linéaires creux de grande taille dans un contexte parallèle multifrontal et hors-mémoire.

Parallel triangular solution in the out-of-core multifrontal approach for solving large sparse linear systems

Abstract

We consider the solution of very large systems of linear equations with direct multifrontal methods. In this context the size of the factors is an important limitation for the use of sparse direct solvers. We will thus assume that the factors have been written on the local disks of our target multiprocessor machine during parallel factorization. Our main focus is the study and the design of efficient approaches for the forward and backward substitution phases after a sparse multifrontal factorization. These phases involve sparse triangular solution and have often been neglected in previous works on sparse direct factorization. In many applications, however, the time for the solution can be the main bottleneck for the performance.

This thesis consists of two parts. The focus of the first part is on optimizing the out-of-core performance of the solution phase. The focus of the second part is to further improve the performance by exploiting the sparsity of the right-hand side vectors.

In the first part, we describe and compare two approaches to access data from the hard disk. We then show that in a parallel environment the task scheduling can strongly influence the performance. We prove that a constraint ordering of the tasks is possible; it does not introduce any deadlock and it improves the performance. Experiments on large real test problems (more than 8 million unknowns) using an out-of-core version of a sparse multifrontal code called `MUMPS` (MUltifrontal Massively Parallel Solver) are used to analyse the behaviour of our algorithms.

In the second part, we are interested in applications with sparse multiple right-hand sides, particularly those with single nonzero entries. The motivating applications arise in electromagnetism and data assimilation. In such applications, we need either to compute the null space of a highly rank deficient matrix or to compute entries in the inverse of a matrix associated with the normal equations of linear least-squares problems. We cast both of these problems as linear systems with multiple right-hand side vectors, each containing a single nonzero entry. We describe, implement and comment on efficient algorithms to reduce the input-output cost during an out-of-core execution. We show how the sparsity of the right-hand side can be exploited to limit both the number of operations and the amount of data accessed.

The work presented in this thesis has been partially supported by SOLSTICE ANR project (ANR-06-CIS6-010).

Keyword: Gaussian elimination, multifrontal method, Distributed computing, parallel computing, sparse matrices, tasks scheduling, multiple right-hand side vectors.

Résumé

Nous nous intéressons à la résolution de systèmes linéaires creux de très grande taille par des méthodes directes de factorisation. Dans ce contexte, la taille de la matrice des facteurs constitue un des facteurs limitants principaux pour l'utilisation de méthodes directes de résolution. Nous supposons donc que la matrice des facteurs est de trop grande taille pour être rangée dans la mémoire principale du multiprocesseur et qu'elle a donc été écrite sur les disques locaux (hors-mémoire : OOC) d'une machine multiprocesseurs durant l'étape de factorisation. Nous nous intéressons à l'étude et au développement de techniques efficaces pour la phase de résolution après une factorization multifrontale creuse. La phase de résolution, souvent négligée dans les travaux sur les méthodes directes de résolution directe creuse, constitue alors un point critique de la performance de nombreuses applications scientifiques, souvent même plus critique que l'étape de factorisation.

Cette thèse se compose de deux parties. Dans la première partie nous nous proposons des algorithmes pour améliorer la performance de la résolution hors-mémoire. Dans la deuxième partie nous pousuivons ce travail en montrant comment exploiter la nature creuse des seconds membres pour réduire le volume de données accédées en mémoire.

Dans la première partie de cette thèse nous introduisons deux approches de lecture des données sur le disque dur. Nous montrons ensuite que dans un environnement parallèle le séquencement des tâches peut fortement influencer la performance. Nous prouvons qu'un ordonnancement contraint des tâches peut être introduit; qu'il n'introduit pas d'interblocage entre processus et qu'il permet d'améliorer les performances. Nous conduisons nos expériences sur des problèmes industriels de grande taille (plus de 8 Millions d'inconnues) et utilisons une version hors-mémoire d'un code multifrontal creux appelé MUMPS (solveur multifrontal parallèle).

Dans la deuxième partie de ce travail nous nous intéressons au cas de seconds membres creux multiples. Ce problème apparaît dans des applications en electromagnétisme et en assimilation de données et résulte du besoin de calculer l'espace propre d'une matrice fortement déficiente, du calcul d'éléments de l'inverse de la matrice associée aux équations normales pour les moindres carrés linéaires ou encore du traitement de matrices fortement réductibles en programmation linéaire. Nous décrivons un algorithme efficace de réduction du volume d'Entrées/Sorties sur le disque lors d'une résolution hors-mémoire. Plus généralement nous montrons comment le caractère creux des seconds -membres peut être exploité pour réduire le nombre d'opérations et le nombre d'accès à la mémoire lors de l'étape de résolution.

Le travail présenté dans cette thèse a été partiellement financé par le projet SOLSTICE de l'ANR (ANR-06-CIS6-010).

Mots-clés: calcul distribué, calcul parallèle, élimination de Gauss, matrices creuses, méthode multifrontale, séquencement des tâches, seconds membres multiples

Contents

Abstract i
Résumé iii

1 General introduction 1
1.1 Context of our study 8
1.2 General background 10
Graphs 10
Direct methods 13
Least-square solution 15
1.3 Test environment 17

I Analysis of the Solution Phase of a Parallel Multifrontal Approach 21

2 Introduction 27

3 Main in-core parallel features of the solver 29
3.1 Introduction 29
3.2 In-core parallel factorization phase 29
3.2.1 Parallelism during the factorization phase 31
3.3 In-core parallel solve phase 33
3.3.1 Some notation 33
3.3.2 Algorithm for management of tasks and messages 33
3.3.3 Algorithm for forward substitution 35
3.3.4 Detailed illustration of the forward substitution 38
3.3.5 Algorithm for backward substitution 40
3.3.6 Detailed illustration of the backward substitution 41

4 Out-of-Core (OOC) main features 45
4.1 Introduction 45
4.2 OOC factorization phase 45
4.3 OOC solve phase 46
4.4 System based demand driven approach 47

5 DIRECT_IO based method 49
5.1 Introduction . . . 49
5.2 User defined buffer . . . 49
5.3 States of a node . . . 50
5.4 Comparison of SYSTEM_BASED and DIRECT_IO methods . . . 51
5.4.1 Sequential case . . . 51
5.4.2 Influence of parallelism on the performance . . . 52
5.5 Influence of scheduling . . . 53
5.5.1 Sequential performance . . . 53
5.5.2 Parallel performance with LIFO scheduler . . . 55
5.5.3 Illustration of the high number of emergency calls with LIFO . . . 56

6 Scheduling to improve performance 59
6.1 NNS scheduler . . . 59
6.1.1 Description of the algorithm . . . 59
6.1.2 Experiments with LIFO and NNS strategies . . . 63
6.2 BPN scheduler . . . 66
6.2.1 Description of the algorithm . . . 66
6.2.2 Experiments with BPN strategy . . . 70

II Exploit Sparsity of Sparse Right-Hand Sides in OOC Environment 73

7 Introduction 79

8 Exploiting sparsity of the right-hand sides: Context and applications 81
8.1 Context of our study . . . 81
8.1.1 Relationship between matrix graph and structure of the solution . . . 81
8.1.2 Background on computing entries in the inverse of a matrix . . . 84
8.2 Sparsity of the right hand-sides and applications . . . 91
8.2.1 Sparse right-hand sides / reducible matrices . . . 91
8.2.2 Null-space computations . . . 92
8.2.3 Computing entries in A^{-1} . . . 96
8.2.4 Pruning and concluding remarks . . . 98

9 Algorithms to exploit sparsity 101
9.1 Introduction . . . 101
9.2 Pruning algorithms . . . 101
9.2.1 'Branch detection' . . . 101
9.2.2 Subtree detection . . . 103
9.3 Topologically-based permutations . . . 104

9.3.1 Post-order permutation of the right-hand sides 106
9.3.2 Pre-order permutation of the right-hand sides 106
9.4 Permuting columns of the right-hand sides to address parallelism 107

10 Hypergraph models to exploit the sparsity 109
10.1 Introduction 110
10.2 Model for entries in A^{-1} 110
10.3 Model for null-space computations 114
10.4 Conclusions 115

11 Results and performance analysis 117
11.1 Introduction 117
11.2 Null-space computations 117
11.2.1 Sequential execution 117
11.2.2 Parallel execution 120
11.3 Computing elements in A^{-1} 122
11.3.1 Sequential execution 122
11.3.2 Parallel execution and permutations 124

12 General conclusion and future work 129

Bibliography 133

Chapter 1

General introduction

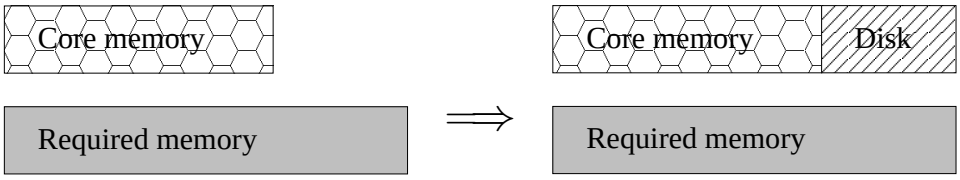
Introduction générale

Contexte de l'étude

Nous nous intéressons à la résolution de grands systèmes linéaires

$$Ax = b \tag{1}$$

avec une méthode directe multifrontale, dans un environnement parallèle hors-mémoire (dans un environnement hors-mémoire le disque dur est utilisé comme extension de la mémoire centrale, voir Figure 1).



a) Mémoire nécessaire insuffisante

b) Utilisation du disque dur pour compléter la mémoire nécessaire

Figure 1: Limitation mémoire résolue par l'utilisation de la mémoire du disque dur.

A est une matrice carrée creuse de très grande taille, et x et b sont des vecteurs colonnes. La matrice originale A est factorisée en un produit de matrices dites matrices de facteurs. Selon que la structure de la matrice est symétrique ou non on effectuera respectivement une factorisation $A = LDL^T$ ou $A = LU$. Les matrices L et U sont respectivement des matrices triangulaires inférieures et supérieures, et D est une matrice diagonale ou bloc-diagonale avec des blocs 1×1 et 2×2. Les matrices de facteurs sont ensuite utilisées pour résoudre le système initial via une séquence de résolution élémentaires,

$$LDL^T x = b \quad ou \quad LUx = b \tag{2}$$

selon que la matrice est symétrique ou non.

Le nombre d'entrées dans les facteurs (pour des problèmes tridimensionnels de grande taille) peut être beaucoup plus important (10 à 100 fois plus grand) que la taille de la matrice originale. Ainsi la mémoire utilisée pour stocker ces facteurs peut constituer un obstacle dans l'utilisation d'approches directes de résolution. Pour autant les méthodes directes, de par leur robustesse numérique sont souvent préférées aux méthodes itératives [35, 66] pour beaucoup d'applications.

Implémenter efficacement les méthodes directes reste un travail délicat dans le cas séquentiel comme dans le cas parallèle. Prévoir le remplissage dans les matrices de facteurs, répartir dynamiquement les tâches pour équilibrer la mémoire en fonction des processeurs utilisés, et beaucoup d'autres subtilités algorithmiques tout aussi critiques pour la performance demandent une expérience forte et un investissement important en temps de développement.

La phase de résolution a été souvent négligée dans les travaux précédents sur la factorisation directe creuse [44, 65, 73, 74, 102, 104, 105]. Pourtant, dans beaucoup d'applications, le temps de résolution peut constituer le problème principal. Dans un contexte hors-mémoire où les facteurs sont stockés sur disque dur, c'est encore plus

critique car le temps de la phase de résolution peut être dominé par l'accès au disque local (et non pas par le temps de calcul comme c'est normalement le cas). Il faut noter qu'il y a alors peu d'espoir pour recouvrir, même partiellement, les calculs avec des entrées/sorties (E/S). Ceci explique la forte influence de l'environnement hors-mémoire sur le temps total de résolution.

Notre principal objectif dans cette thèse a été l'étude et le développement d'approches efficaces pour la phase de résolution dans un environnement parallèle à mémoire distribuée [9, 10, 11] et dans un contexte hors-mémoire. Nous proposons dans cette thèse des algorithmes pour améliorer la performance de la résolution directe multifrontale hors mémoire. Notre travail diffère et étend le travail d'autres applications en environnement hors-mémoire (voir [1, 103, 104, 105] et [114]) selon trois axes : d'abord, comme décrit dans [1] nous considérons un contexte parallèle. Deuxièmement, nous nous concentrons sur la performance de la phase de résolution. Troisièmement, nous mettons en oeuvre des algorithmes pour exploiter le caractère creux (" sparsité") des seconds membres (quand b dans l'Équation (2) devient une matrice creuse).

Avant l'introduction des notions de base, nous décrivons dans le paragraphe suivant la structure de la thèse. Dans la première partie de ce travail, nous étudions et comparons deux mécanismes d'entrées-sorties pour accéder aux données du disque dur. Une couche logicielle a été écrite en C pour cacher tous les mécanismes d'E/S de bas niveau (gestion du buffeur, pré-chargement des données, synchronisation). Nous avons remarqué que la performance de la phase de résolution est fortement liée à la façon dont on accède aux données sur le disque dur et au nombre et à la régularité de ces accès. Nous avons aussi démontré qu'en parallèle l'ordre avec lequel les tâches sont exécutées influence d'une manière importante la performance de la phase de résolution. Notre travail sur l'ordonnancement des tâches nous a permis de développer une nouvelle approche efficace aussi bien en séquentiel qu'en parallèle. Les expériences sur de nombreuses matrices, dont certaines de plus de 8 millions d'inconnues, montrent le bon comportement des approches proposées en utilisant la version parallèle out-of-core du solveur direct multifrontal MUMPS.

Dans la deuxième partie de la thèse nous nous intéressons à la sparsité (nature creuse) du second membre. Nous étudions différentes techniques qui préservent la sparsité des calculs grâce à l'exploitation de la nature creuse des seconds membres. Des applications à plusieurs seconds membres issues des domaines applicatifs tels que l'électromagnétisme et l'assimilation de données sont utilisées pour illustrer les performances des approches proposées. Par ailleurs, lorsque le nombre de seconds membres est important (dans certains cas plusieurs dizaines de milliers de seconds membres) nous avons étudié, implémenté et décrit des techniques efficaces pour réduire le volume d'Entrées/Sorties. Nous démontrons que l'ordre de traitement des seconds membres peut être utilisé pour réduire aussi bien le nombre d'opérations que la taille totale des données à précharger du disque dur. Nous proposons des permutations de seconds membres permettant d'améliorer l'utilisation de la mémoire et d'optimiser les préchargements du disque dur.

Notions de base et définitions

- Graphes

Un graphe est un ensemble fini de noeuds et d'arêtes. Une arête est définie par une paire non-ordonnée de sommets. Un graphe est connexe s'il est possible, à partir de n'importe quel sommet, de rejoindre tout autre sommet en parcourant les arêtes du graphe. En donnant un sens aux arêtes d'un graphe, on obtient un graphe orienté. Un graphe orienté sans cycle est dit acyclique (**dag**). On utilise les dags pour représenter la structure de la matrice (voir Figure 2).

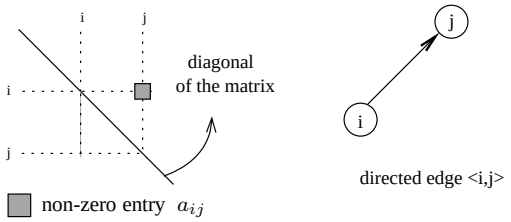


Figure 2: L'entrée non-nulle $a_{i,j}$ correspond à l'arête orientée $< i, j >$ dans la representation-graphe.

Propriété 1. *Toute matrice triangulaire (supérieure ou inférieure) peut être représentée par un graphe orienté sans cycle (dag).*

La connectivité entre noeuds d'un graphe orienté peut être représentée de façon efficace grâce au graphe obtenu par réduction transitive. Si la matrice est symétrique, la réduction transitive du graphe associé à la matrice des facteurs (L telle que $A = LDL^T$) est un arbre appelé **l'arbre d'élimination** (voir par exemple Gilbert et Liu [63]). Si la matrice est non-symétrique alors la réduction transitive du graphe associé à chacune des matrices de facteurs (L et U telles que $A = LU$) est un graphe orienté acyclique particulier appelé **e-dag** par Gilbert et Liu en [63].

Les hypergraphes généralisent la notion de graphe dans la mesure où les arêtes ne relient plus un ou deux sommets, mais un nombre quelconque de sommets. Un hypergraphe (défini comme un ensemble de noeuds et un ensemble de "nets") a la particularité que chaque net est aussi un ensemble de noeuds. Les noeuds qui ont certaines propriétés communes sont mis ensemble sous forme de nets. Un noeud peut faire partie de plusieurs nets (voir Figure 3, le noeud 4 fait partie des nets 1 et 2).

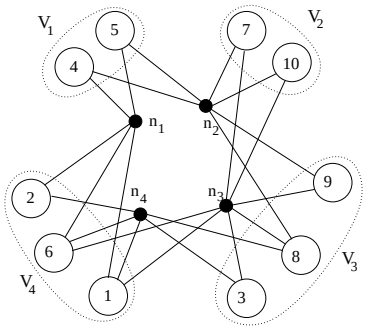


Figure 3: Exemple d'un hypergraphe contenant 10 noeuds (représentés par des cercles) et 4 nets (représenté par des points). L'hypergraphe est partitioné en 4 parties, representées par des ellipses.

Les hypergraphes sont manipulés dans tous les domaines où l'on utilise la théorie des graphes : résolution de problèmes de satisfaction de contraintes, traitement d'images, optimisation d'architectures réseaux, modélisation, etc.

- Méthodes directes

Les méthodes directes de résolution de systèmes linéaires creux se déroulent en trois phases : une phase d'analyse, une phase de factorisation, et une phase de résolution. Une fois la factorisation réalisée ($A = LU$ ou $A = LDL^T$ dans le cas d'un matrice symétrique), le système $Ax = b$ se résout en deux étapes : résolution du système $Ly = b$ (phase dite de 'descente'), puis du système $Ux = y$ (phase de 'remontée').

La dépendance des calculs est représentée par le graphe d'élimination (e-dag) qui est un arbre (l'arbre d'élimination) dans le cas symétrique. Dans notre approche directe multifrontale, nous utilisons la matrice symétrisée $A+A^T$, ce qui conduit à la substitution de l'e-dag par un arbre d'élimination. Une particularité sur laquelle repose l'efficacité des méthodes directes est que les colonnes de la matrice qui ont une structure similaire sont groupées dans des supervariables appelées **supernodes** [51, 94, 98], qui sont ensuite éliminées simultanément. Les méthodes multifrontales diffèrent d'autres méthodes directes (voir [78]), telles que les approches dites left-looking et right-looking, qui sont caractérisées par la façon dont les mises à jour sont faites. Dans une approche right-looking, les modifications résultant du calcul courant sont immédiatement répercutées sur le reste des données concernées. Dans une approche left-looking, ce n'est qu'au moment où l'on travaille sur une donnée que l'on va prendre en compte toutes les modifications résultant des étapes précédentes. Il faut noter que les mises à jour correspondent à des messages dans le cas d'exécution parallèle sur architectures à mémoire distribuée. Dans ce contexte, la structure des communications dépend fortement de la méthode choisie. Le volume et le nombre de messages dépendent aussi de la répartition ("mapping") des noeuds sur des processeurs.

- Environnement et matrices de test

Nos tests ont été effectués sur le calculateur parallèle à mémoire partagée Cray XD1 situé au CERFACS (58 noeuds, 2 processeurs par noeud, 4 Go par noeud, 2 Go par processus MPI , système de fichier `reiserfs`, et bande passante pour la lecture des données de 16 Mo/s au maximum), en utilisant un seul processus MPI par noeud.

Le tableau 1 décrit nos matrices de tests, ordonnées par rapport à la taille de leurs facteurs. Nous avons aussi fait des expériences sur des matrices de plus petite taille dont la structure particulière est orientée vers des applications de seconds membres creux multiples. Le tableau 2 représente les matrices utilisées pour le calcul du noyau des matrices déficientes. Dans le tableau 3 nous décrivons les matrices correspondant à l'étude de problèmes de moindres carrés. Plus précisément, l'étude de la variance et de la covariance conduit à calculer certains éléments de l'inverse de la matrice des équations normales $A^T A$. Certaines de ces matrices sont issues d'une collaboration avec le Centre d'Etude Spatiale du Rayonnement (CESR) de Toulouse et correspondent à des problèmes d'astrophysique. Cette application requiert un fort volume de calcul (14528 secondes) et nous montrerons que l'exploitation de la structure creuse des seconds membres permet de réduire ce temps de calcul de façon tout a fait significative.

Nom de la matrice	Ordre	Entrées (Millions)	Facteurs (MB)	Nb Noeuds dans l'arbre	Description (origine)
QIMONDA07*	8 613 291	66.9	2 534	3 083 998	Simulation de circuit (Qimonda AG)
CAS4R-L15	2 423 135	19.5	4 832	864 447	Electromagnétisme 3D (EADS)
CONESHL *	1 262 212	43.0	5 908	113 513	Eléments finis 3D (SAMTECH)
NICE20MC *	715 923	28.1	9 263	68 134	Traitement sismque (BRGM)
AUDI *	943 695	39.3	12 202	113 119	Modélisation d'un vilebrequin
GRID3.5M	3 500 000	37.8	15 720	1 535 044	Discrétisation 11 points d'un Laplacien 3D
GRID5M	5 000 000	53.8	17 798	2 203 434	Discrétisation 11 points d'un Laplacien 3D
COR5HZ *	2 233 031	90.2	21 622	268 798	Traitement sismique (BRGM)
AMANDE	6 994 683	58.5	55 295	871 621	Electromagnétisme 3D (CEA-CESTA)
NICE9HZ *	5 140 838	215.5	64 848	603 495	Traitement sismique (BRGM)

Table 1: Matrices de tests: taille et origine. Les matrices marquée d'une * sont publiques.

Nom de la matrice	Ordre	Nb entrées	Nb Noeuds dans l'arbre	Globale Def.	Racine Def.	Pivots nuls
boxcav_8_5_3	619	3 471	319	56	7	49
boxcav_16x10x3	2 675	15 953	1 311	270	10	260
boxcav_20x13x3	4 419	26 129	2 121	456	10	446
boxcav_30x20x4	14 454	89 185	5 758	1 653	103	1 550
boxcav_40x27x5	33 627	212 883	12 948	4 056	185	3 871

Table 2: Matrices de tests pour le calcul de la base du noyau de matrices déficientes: taille et déficience.

Nom de la matrice	Ordre	Nb entrées	Nb Noeuds
a-1_08M	8 999	497 628	1 186
a-1_21M	21 532	855 866	5 207
d-11_25M	25 000	249 720	12 091
a-1_46M	46 799	1 791 242	12 419
a-1_72M	72 358	3 549 284	7 941
a-1_148M	148 286	7 388 031	12 734

Table 3: Matrices de tests pour calculer des entrées dans $(A^T A)^{-1}$.

1.1 Context of our study

We are interested in solving large sparse linear systems of the form

$$Ax = b \tag{3}$$

with direct methods [45, 47, 59] in a parallel limited-memory environment. Here A is a large, square, sparse matrix and b and x are column vectors. We are first interested in case where A is nonsingular matrix. The case of a singular matrix A is discussed in Chapter 8.2.1 where the specific structure of the matrix is exploited for the sparsity of the computations during the solution phase. In the direct solution of this linear system, the matrix A is first factorized into the factors LDL^T (when A is symmetric) or LU (when A is unsymmetric), where L and U are triangular matrices and D is a diagonal or block diagonal matrix with 1×1 and 2×2 blocks. These factors are then used to solve the system through the forward and backward substitution steps

$$\left[\ LDy = b \ \text{ and } \ L^T x = y\ \right] \quad \text{or} \quad \left[\ Ly = b \ \text{ and } \ Ux = y\ \right], \tag{4}$$

depending on whether the matrix is symmetric or not.

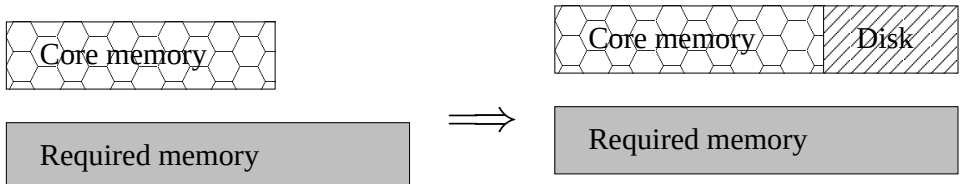
In this context, the number of entries in the factors can be an important limitation for using sparse direct solvers. Indeed, the number of entries in the factors (on large 3-dimensional problems) can be much larger (10 to 100 times larger) than the size of the original matrix. This is one reason for users to choose iterative methods (see for example [35, 66]). Time for solution can be another reason. The direct solution of sparse linear systems using Gaussian elimination [59]has a clear advantage over iterative methods in terms of numerical robustness, and it remains the method-of-choice for many applications. However, it is very challenging to implement such methods efficiently on a single processor. This is even more complicated on multiprocessor machines. One of the main reasons is because of fill-in created during the matrix factorization. Moreover, if numerical pivoting is necessary this involves dynamically tracking the fill-ins that are generated in a somewhat unpredictable way. Handling highly irregular data access and computation is further compounded by sophisticated computer architectures with several layers of memory hierarchy. Therefore, unlike many iterative algorithms that users can often implement reasonably well and quickly by themselves, direct solvers require much more expertise and a longer time to develop.

Working out-of-core (using the storage disks to extend the main memory), we can overcome the memory limitation of direct methods [44, 65, 73, 74, 102, 104, 105] and handle large matrices whose factors do not fit within the main memory of the computer. If the memory required for solving a matrix is larger than the available core memory (as shown in Figure 4-a), a natural possibility to overcome this problem is to use the hard disk memory (as shown in Figure 4-b).

In general, direct methods proceed in the following three phases.

- Analysis phase: The matrix is preprocessed to limit the fill-in and to improve its numerical behaviour. The symbolic factorization is performed and the computational dependency graph is computed.

- Factorisation phase: The factors are computed ($A = LU$ or, in the symmetric case $A = LDL^T$).



a) Memory crash

b) Use of disk to complete the required memory

Figure 4: Memory constraint is solved by extending the main memory with the memory on disk.

- Solution phase: Forward and backward substitutions (respectively $Ly = b$ and $Ux = y$, in general, or $DL^T x = y$ in the symmetric case).

For an unsymmetric matrix, we compute its LU factorization; if the matrix is symmetric, its LDL^T factorization is computed. Because of numerical stability, pivoting is required in these cases in contrast to symmetric positive definite sparse systems where pivoting can be avoided.

The solution phase involves sparse triangular solutions and has often been neglected in previous work on sparse direct factorization. In many applications, the time for solution is even the main bottleneck for the performance. In an out-of-core context (factors stored on local disk), this is even more critical since the time for the solution phase can be dominated by the time for memory access and not by the time to perform the arithmetic operations. It is interesting to notice that running out-of-core does not significantly affect the time performance of the factorization (see for example [1, 3, 102, 105]). This can be explained by the fact that the time spent doing computation during the factorization phase is generally much larger than the time to perform input/output (I/O) on disks. I/O access can then be 'hidden' by overlapping I/O with computation. On the other hand, the number of operations during the solution phase is of the order of the size of the factors (in the case of a single right-hand-side), which is equal to the volume of I/O. Thus, there is very little scope for overlapping computation with I/O, which explains the strong influence of the out-of-core environment on the time for solution.

Our main focus in this thesis has been the study and design of efficient approaches for the forward and backward substitution phases of a distributed parallel sparse multifrontal solver [9, 10, 11] in an out-of-core context. Our work differs and extends the work of other out-of-core applications (see [1, 103, 104, 105] and [114]) in three aspects. First, as done in [1] we consider a parallel out-of-core context. Second, we focus on the performance of the solution phase. Third, we design algorithms to exploit the sparsity of multiple right-hand side vectors (when b in Equation (4) is a sparse matrix).

Before providing in the general background section some basic ideas and theory, we describe in the following the outline of the thesis.

In the first part of this work, we describe and compare two I/O approaches to access data from the hard disk. An input/output (I/O) software layer written in C has been designed to hide all the low level I/O mechanisms (small buffer management, prefetch and post-store mechanism, synchronisation). Using this software layer we have been able to work at an algorithmic level on the algorithms to design an efficient solution phase in an out-of-core (OOC) context. We have observed that the performance of the solution phase is strongly related to the way data on disk is accessed and to the number and the regularity

of the accesses. We have then shown that in a parallel environment task scheduling can also strongly influence the performance. We have proved that a constrained ordering of the tasks is possible – it does not introduce any deadlock and it improves the performance. Experiments on large real test problems (more than 8 million unknowns) using an out-of-core version of a sparse multifrontal code called `MUMPS` (MUltifrontal Massively Parallel Solver) have shown the good behaviour of our algorithms.

In the second part of the thesis, we are interested in applications with sparse multiple right-hand sides. Applications in electromagnetism and data assimilation have been used to illustrate our discussion. In such applications we need either to compute the null-space of a highly deficient matrix or to compute entries in the inverse of a matrix associated with the normal equations of linear least-squares problems. We have described, implemented and discussed efficient algorithms to reduce I/O data when solving with OOC execution. We have shown how the sparsity of the right-hand sides can be exploited to limit both the number of operations and the amount of data accessed.

1.2 General background

Graphs

A given square matrix A can be structurally represented by its associated graph – $G(A)$. A **graph** $G = (V, E)$ is a set of **nodes** or vertices V connected by a set of **edges** E. Nodes correspond to rows (columns) of the matrix and edges correspond to nonzero entries. Any nonzero position in A $(a_{ij} \neq 0)$ corresponds to an edge from node i to node j in the graph $G(A)$ which we write as $< i, j >$ (see Figure 5).

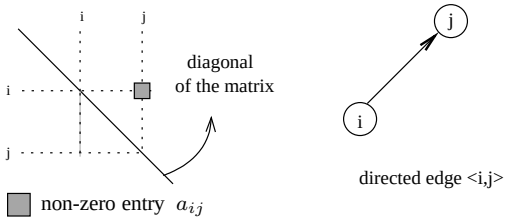


Figure 5: The non-zero entry $a_{i,j}$ corresponds to a directed edge $< i, j >$ in the graph representation.

A one-way edge is called a **directed edge** and has the property:

$$a_{i,j} \neq 0 \Longleftrightarrow \exists\, directed\, edge\ < i, j >$$

Note that as we differentiate entries $a_{i,j}$ and $a_{j,i}$, we also distinguish their corresponding directed edges $< j, i >$ and $< i, j >$. We say that, there is a **path** from node i to node k in the graph, if we can follow directed edges from node i to node k in the graph. In such a case we say that node k is **reachable** from node i.

A graph is said to be **connected**, in the sense of a topological space, if there is a path from any vertex to any other vertex in the graph. Any irreducible matrix A can be represented by a connected graph $G(A)$.

A graph with directed edges and no **cycle** is called a **directed acyclic graph (or dag)**.

Property 1. *Any lower triangular matrix or upper triangular matrix can be represented by a directed acyclic graph (dag).*

An economical way to represent path information for a directed graph is by its transitive reduction ([4]). An arbitrary graph may have many transitive reductions, but Aho, Garey and Ullman [4] show that a dag has only one. For a given unsymmetric matrix A which can be factored as $A = LU$, Gilbert and Liu in [63] define an **edag** of L (respectively U) as the unique transitive reduction of $G(L)$ (respectively $G(U)$).

Example 1. *Elimination dag*
We consider the L pattern of a given matrix, as shown in Figure 6. Its associated directed graph $G(L)$ is acyclic, as stated in property 1. The edges corresponding to redundant paths are removed (edge from node 5 to node 1) and the reduced elimination dag or edag is build.

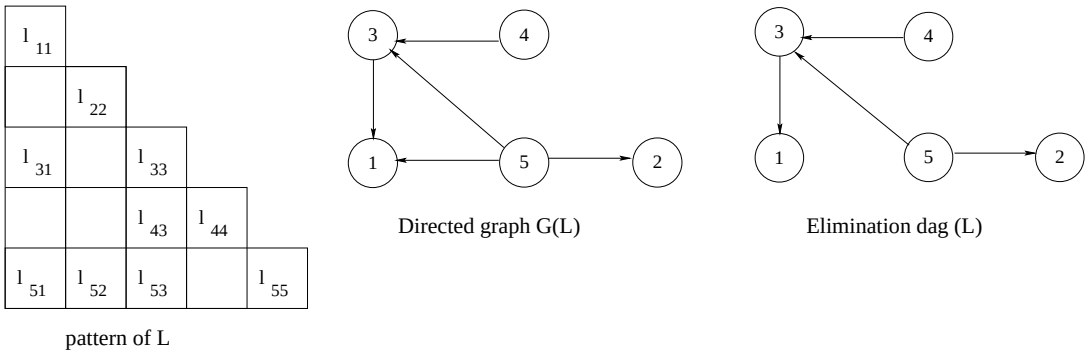


Figure 6: Example of L pattern with the associated dag and the reduced elimination dag (edag) of L.

Theorem 1 (Gilbert and Liu [63])**.** *For a symmetric matrix, the edag(L) is a tree, the so called* **elimination tree**.

Note that for our example in Figure 6 the edag of L is not a tree, since node 3 has two father nodes – 4 and 5. From Theorem 1, the original matrix A was thus not symmetric. Indeed entry u_{34} of the U factors of the factorization of A must be zero to have $l_{54} = 0$. On our test example, symmetrizing the matrix such that the U entry u_{34} becomes non-zero is enough to make our edag a tree ($u_{34} \neq 0$ implies $l_{54} \neq 0$) as shown in Figure 7.

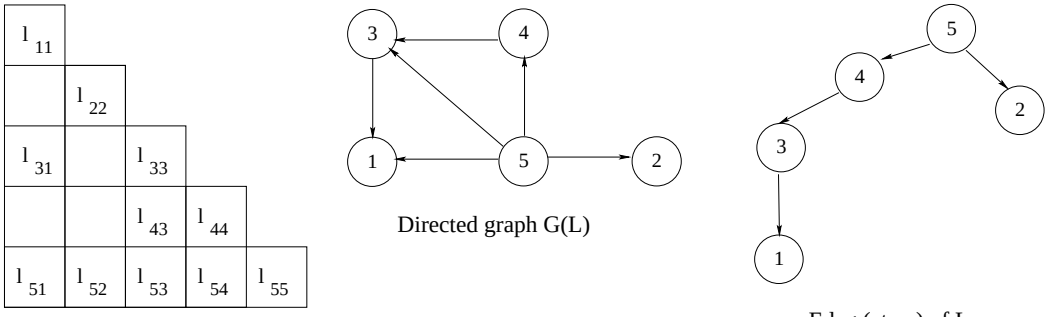


Figure 7: Modification of the patten of L from Figure 6 such that the associated edag is a tree.

Hypergraphs

We give a brief definition of hypergraphs, which will be used in Section 10 for the hypergraph based permutation of the right-hand sides. Hypergraph use and construction will be described in Chapter 1.2.

A hypergraph $H = (V, N)$ is defined as a set of vertices V and a set of nets N. Every net is a subset of vertices. The size of a net n_i is equal to the number of its vertices, i.e., $|n_i|$. The set of nets that contain vertex v_j is denoted by $Nets(v_j)$.

Example 2. Hypergraph model: *Figure 8 shows a hypergraph with* 10 *vertices, represented by circles, and* 4 *nets, represented by points. The net* n_1 *contains* 5 *vertices:* v_4, v_5, v_1, v_2 *and* v_6*, thus its size is* 5 *(*$|n_1| = 5$*).*

Weights and costs can be associated with vertices and nets, respectively. We use $w(j)$ to denote the weight of the vertex v_j, and $c(i)$ to denote the cost of the net n_i.

$\Pi = \{V_1, \ldots, V_s\}$ is a s-way vertex partition of $H = (V, N)$ if each part is nonempty, the parts are pairwise disjoint, and the union of parts equals V. In Π, a net is said to *connect* a part if it has at least one vertex in that part. The *connectivity set* $\Lambda(i)$ of a net n_i is the set of parts connected by n_i. The *connectivity* $\lambda(i) = |\Lambda(i)|$ of a net n_i is the number of parts connected by n_i. In Π, the weight of a part is the sum of the weights of vertices in that part.

In the hypergraph partitioning problem, the objective is to minimize

$$cutsize(\Pi) = \sum_{n_i \in N} c(i).(\lambda(i) - 1) \ . \tag{5}$$

Example 3. Net's cost and partitioning: *In Figure 8, there are four disjoint parts* $\{V_1, \ldots, V_4\}$ *and their union by definition gives* V*. The connectivity of net* n_1 *is* 2 *(*$\lambda(1) = 2$*), because* n_1 *is connected to parts* V_1 *and* V_4*.*

Let suppose that the cost of each net in Figure 8 is 1 *(*$c(i) = 1$*). Thus:*

$$\begin{aligned} cutsize(\Pi) &= \sum_{i=1}^{4} c(i).(\lambda(i) - 1) = \\ &= 1.(2-1) + 1.(3-1) + 1.(3-1) + 1.(2-1) = 6 \end{aligned}$$

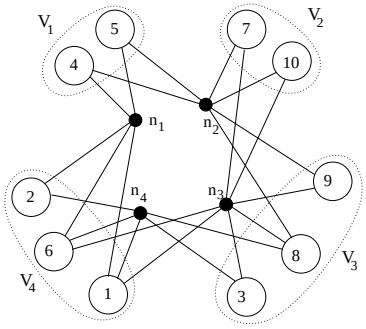


Figure 8: Example of hypergraph containing 10 vertices (represented by circles) and 4 nets (represented by points). The hypergraph is partitioned into 4 parts, represented by ellipses.

Minimizing the $cutsize$ function is widely used in the VLSI (Very-Large-Scale Integration) community [89] and in the scientific computing community [17, 27, 115, 116], and it is referred to as the $connectivity\ -1$ cutsize metric. The partitioning objective is to satisfy a balancing constraint on part weights:

$$\frac{W_{max} - W_{avg}}{W_{avg}} \leq \varepsilon \ . \tag{6}$$

Here W_{max} is the largest part weight, W_{avg} is the average part weight, and ε is an allowable imbalance ratio. The problem is NP-hard [89].

Direct methods

As we focus on the multifrontal method, we will comment on some of its main properties with respect to other methods. For an overview of the multifrontal method (although we describe in the next chapter), we refer the reader to [47, 51, 78, 93]; for the discussion of other direct approaches, we refer the reader to [39, 77, 81]. The multifrontal method was initially developed for indefinite sparse symmetric linear systems [51] and was then extended to unsymmetric matrices [52]. It belongs to the class of approaches which separates the factorization into two phases. The symbolic factorization phase is not concerned with numerical values. It looks for a permutation of the matrix that will reduce the number of operations and memory requirements in the subsequent phase, and then computes a dependency graph associated with the factorization. Finally, in an implementation for parallel computers, this phase partially maps the graph onto the target multiprocessor computer. The numerical factorization phase computes the matrix factors that will then be used during the solution phase to compute a solution. The experimental part and the development performed in this thesis are based on the `MUMPS`, a MUltifrontal Massively Parallel Solver [8, 10].

Note that on unsymmetric matrices, the computational dependency graph is the so-called elimination dag (or edag). This edag is used in the unsymmetric multifrontal approaches `UMFPACK` [34] and `WSMP` [75, 76]. In our multifrontal approach, the pattern of the symmetrized matrix $A + A^T$ will be used so that the edag is in fact an elimination tree. The elimination tree represents the task dependency of the computations, it gives a partial order in which the columns can be eliminated. For example the elimination tree on Figure 9, associated with factors on Figure 10 expresses dependency: column 5 must

wait for the elimination of columns 3 and 4. Node 5 of the elimination tree is said to be the father of nodes 3 and 4. The elimination tree also provides parallelism; column 3 and 4 can be processed in parallel. Note that in a general case the elimination tree is a forest (if the matrix is reducible). For the sake of clarity we will continue to use the term elimination tree in the rest of the thesis even when the matrix is reducible.

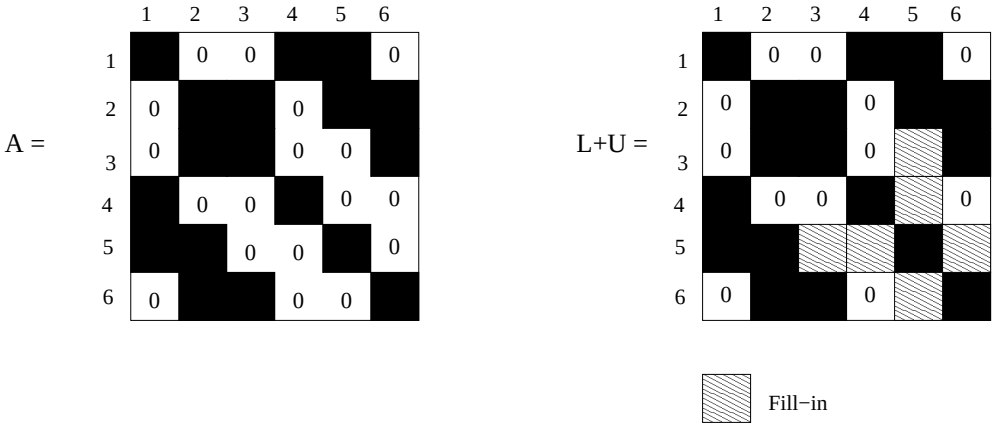


Figure 9: Pattern of a structurally symmetric matrix and fill-in in its factors.

An important issue for efficiency is that columns with similar sparsity pattern are grouped into large supernodes [51, 94, 98]. The resulting tree will be referred to as **the assembly tree**. In Figure 9, columns 2 and 3 of the L factors have the same structure and are processed as a unique node in the assembly tree compatible with the elimination tree, shown in Figure 10.

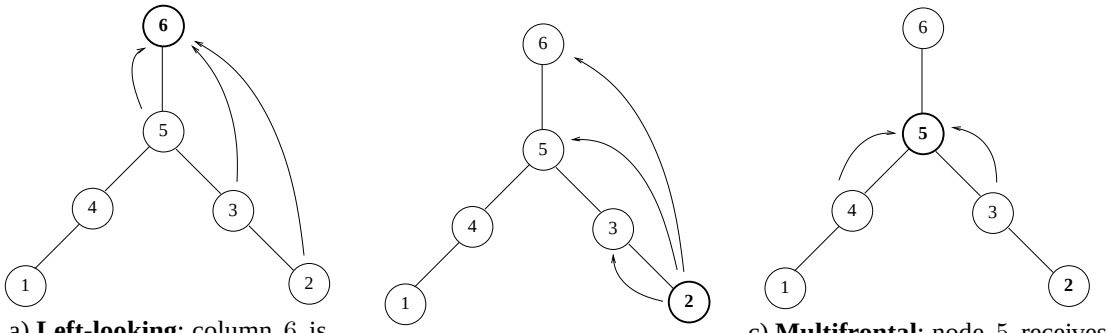


Figure 10: Elimination tree and assembly tree associated with the matrix of Figure 9.

In practice, supernodes are naturally used in direct solvers whatever the method is (left-looking, right-looking or multifrontal), as for example in `SuperLU` [38, 40], `PaStiX` [80], `UMFPACK` [34], `TAUCS` [113], `Oblio` [42, 43], `PARDISO` [106, 107], `PSPASES` [77], `HSL library` [82], `SPOOLES` [15], `WSMP` [75, 76], `MUMPS` [9, 10, 11], and others. Some of these solvers have been designed for distributed memory computers (see for example `PaStiX`, `SuperLU_DIST`, `PSPASES` and `MUMPS`. Because of the difficulty of handling dynamic data structures efficiently, most distributed memory approaches do not perform numerical pivoting during the factorization phase. Instead, they are based on a static mapping of the tasks and data and do not allow task migration during numerical factorization. In this context one unique and original feature of `MUMPS` solver is that it enables standard numerical pivoting. Dynamic task creation, scheduling and data mapping are used to handle numerical issues and to provide a very adaptive approach. Numerical pivoting can clearly be avoided for symmetric positive definite matrices. For unsymmetric matrices, Duff and Koster [48, 49] have designed algorithms to permute large entries onto the diagonal and have shown that this can significantly

reduce numerical pivoting. Demmel and Li [90] have shown that, if one preprocesses the matrix using the code of Duff and Koster, static pivoting (with possibly modified diagonal values) followed by iterative refinement can normally provide reasonably accurate solutions. They have observed that this preprocessing, in combination with an appropriate scaling of the input matrix, is a key issue for the numerical stability of their approach.

One main difference between multifrontal and other direct approaches (see [78]) such as left-looking and right-looking, is in the way of doing the updates for each node in the elimination tree. In the left-looking approach the updates to a node are done just before the node is factorized. This is also known as a fan-in method [14, 79]. In the right-looking approach, the updates to each node are sent just after the factorization. This approach is also known as a fan-out method. From this point of view, the multifrontal method [13, 52, 93] can be seen as a combination of left-looking and right looking approaches, where all updates are sent after the factorization of the current node but only to its father. To do the update the father must be capable of storing all contributions from all its descendants. One can show that a full square matrix (so called frontal matrix) of order the number of nonzero entries in the column of L is enough to store all contributions. More that one branch of the tree and multiple associated frontal matrices can be processed simultaneously so that the method has been named the multifrontal approach.



a) **Left-looking**: column 6 is updated with contributions of nodes 2, 3 and 5 just before processing node 6.

b) **Right-looking**: after processing node 2 the updates to node 3, 5 and 6 are done.

c) **Multifrontal**: node 5 receives updates only from its direct children – nodes 3 and 4, then starts to factorize node 5.

Figure 11: Updates in left-looking, right-looking and multifrontal approaches. The bold nodes represent the current node and the arrows refer to updates.

Note that each update corresponds to a communication message in a parallel distributed memory environment so that each approach will have a different communication pattern. The volume and number of messages will then strongly depend on the mapping of the nodes of the elimination tree onto the processors (see [78]).

Least-square solution

Linear least-squares problems arise in many important fields of science and engineering, such as econometry, geodesy, statistics, structural analysis, fluid dynamics, etc. The linear least-squares problem [21, 61, 95, 96, 97, 99] is a computational problem that originally arose from the needs to fit a linear mathematical model to given observations. To reduce the influence of errors in the observations a great number of measurements are taken. Thus the resulting problem to solve is an overdetermined linear

system of equations. In matrix terms, given a vector $\mathbf{b} \in \mathcal{R}^m$ and a matrix $\mathbf{A} \in \mathcal{R}^{m \times n}, m > n$, we want to find a vector $\mathbf{x} \in \mathcal{R}^n$, such that Ax is the 'best' approximation to b. There are many possibilities of defining this 'best' approximation. Often for statistical reasons, but also to provide a simple computational problem x is chosen to minimize the Euclidean vector norm:

$$min_x ||Ax - b||_2 \quad , \qquad where \quad A \in \mathcal{R}^{m \times n}, \quad b \in \mathcal{R}^m \tag{7}$$

This is known as the linear least-squares problem. Vector x is the linear least-squares solution of the system $Ax = b$. Let $\mathbf{r}$ be the residual vector, $r = b - Ax$. Thus to solve the linear least-squares problem we must minimize $||r||_2^2$ which is the sum of the squared residuals: $||r||_2^2 = \sum_{i=1}^{m} r_i^2$. If the rank of matrix A is smaller than n ($\mathbf{rank}(A) < n$), the solution x of equation (7) is not unique. However, among all least-squares solutions there is an unique solution which minimizes $||x||_2$ (see Chapters 1 and 2 of Björck, Numerical Methods for Least-Square Problems [21]).

In linear statistical models the vector b of observations is related to the unknown vector x by the linear relation:

$$Ax = b + \epsilon \tag{8}$$

where ϵ is a vector of random errors. Let $rank(A) = n$ and ϵ has zero mean, $\mathcal{E}(\epsilon) = 0$. Let also the variance-covariance matrix be $\nu(\epsilon) = \sigma^2 I$. Then by the Gauss-Markov theorem, the least-squares estimate $\hat{\mathbf{x}}$ is the linear unbiased estimator of x (an estimator for which there is no difference between an estimator's expected value and the true value of the parameter, $\hat{x} = x$) with minimum variance equal to

$$V_x = \sigma^2 C_x \quad , \qquad C_x = (A^T A)^{-1} = R^{-1} R^{-T} \tag{9}$$

where $\mathbf{R}$ is the Cholesky factor of the so called **normal equations** $A^T A$. An unbiased estimate of σ^2 is given by :

$$s^2 = ||\hat{r}||_2^2 / (m - n) \quad , \qquad \hat{r} = b - A\hat{x}.$$

In order to assess the accuracy of the computed estimate of x it is often required to compute the minimum variance matrix V_x or part of it. In particular, the variance of the component $\hat{x}_i$ is given by the diagonal entries v_{ii} in V_x [101].

1.3 Test environment

Except where stated otherwise, all our runs have been performed on the multiprocessor Cray XD1 located at CERFACS (58 nodes with 2 processors per node; and 4 GB per node, 2 GB per MPI process). Each node is equipped with an `IDE` disk managed by the `reiserfs` file system of maximum bandwidth for a read operation close to 16 MB/sec with one MPI process per node.

Performance of the solution phase

Our set of test matrices used for the experiments in the first part of the thesis – solution phase performance is described in Table 4, sorted by factor size. The size of the factors is obtained using a `Metis` reordering [83] of the original matrix. All test matrices are real symmetric except CAS4R-L15 and AMANDE which are complex symmetric.

Matrix name	Order	Entries (Millions)	Factors (MB)	Nb Nodes in the tree	Description (origin)
QIMONDA07*	8 613 291	66.9	2 534	3 083 998	Circuit simulation (Qimonda AG company)
CAS4R-L15	2 423 135	19.5	4 832	864 447	3D Electromagnetism (EADS)
CONESHL *	1 262 212	43.0	5 908	113 513	3D finite element from SAMTECH
NICE20MC *	715 923	28.1	9 263	68 134	Seismic processing (BRGM Lab.)
AUDI *	943 695	39.3	12 202	113 119	Automotive crankshaft model
GRID3.5M	3 500 000	37.8	15 720	1 535 044	3D 11pt-discretization of Laplacian operator
GRID5M	5 000 000	53.8	17 798	2 203 434	3D 11pt-discretization of Laplacian operator
COR5HZ *	2 233 031	90.2	21 622	268 798	Seismic processing (BRGM Lab.)
AMANDE	6 994 683	58.5	55 295	871 621	3D Electromagnetism (CEA-CESTA)
NICE9HZ *	5 140 838	215.5	64 848	603 495	Seismic processing (BRGM Lab.)

Table 4: Test matrices: size and origin. Matrices marked by * are publicly available.

Matrix AUDI from the PARASOL Collection [1] or the matrices from our applications partners that are publicly available can be found on the `gridtlse.org` web site. COR5HZ matrix corresponds to a dynamic analysis of the Cornioglio (Italy) earthquake (1994) with maximum signal frequency of 5 Hz. NICE20MC and NICE9HZ correspond to dynamic analysis of the Nice earthquake (2001) with maximum signal frequency of 1.5 Hz and 9 Hz respectively. AMANDE and CAS4R-L15 are problems from electromagnetism. CONESHL corresponds to 3D computations from structural engineering and QIMONDA07 to circuit simulation.

We show in Table 4 the order and the number of entries for each matrix which give us an estimation about the size and the sparsity of the matrix. The factor size denotes the amount of LU factors stored on disk during the factorization phase and read during the solution phase. The time performance of the solution phase in an out-of-core environment is strongly related to this amount of data. We show also the number of nodes in the elimination tree which is very useful to estimate the impact of the scheduling strategy. The more nodes that are in the tree, the more important will be the influence of the scheduling.

We note the difficulty in getting very large problems from industry. It is also necessary that the integer description (symbolic representation of the matrix) will fit on a single processor in order for us to complete the analysis and construct the data structures for subsequent numerical factorization and solution.

[1] www.parallab.uib.no/projects/parasol/data

During factorization **all** factors are written to files (local to each MPI process) on disks. In our experimental context (one MPI process per node) all I/O files of each MPI process are thus associated with local disks. Our approach will however naturally work when disks are shared by more than one MPI process but with a reduction in the average I/O bandwidth. Furthermore, factors are not kept in memory at the beginning of the solution phase *and* between the forward and backward steps. So we have no intended reuse of data, which will help to better understand the behaviour of each step.

With these assumptions, we will thus have to load all of the factors during the solve phase. Note that QIMONDA07 is a large and very sparse matrix with more than 3 million nodes in the assembly tree. Indeed, it is the matrix with the largest number of nodes in our set. I/O access might occur for each node of the elimination tree and thus it is an interesting matrix to illustrate the behaviour of our algorithms. We thus use this example extensively in our detailed analysis but show relevant results on all our test problems later in the thesis.

Exploit the sparsity of the right-hand side vectors

In the second part of the thesis we are interested in applications with sparse multiple right-hand sides. An application in electromagnetism leads to computing the null-space basis of a matrix with a large deficiency. Another application in astrophysics requires the computation of the diagonal entries of the inverse of a matrix.

For null-space basis computations our test matrices (given in Table 5) come from 3D applications in electromagnetism when computing resonance modes in box cavities discretization.

Matrix name	Order	Nb entries	Nb Nodes in the tree	Global Def.	Root Def.	Null Pivots
boxcav_8_5_3	619	3 471	319	56	7	49
boxcav_16x10x3	2 675	15 953	1 311	270	10	260
boxcav_20x13x3	4 419	26 129	2 121	456	10	446
boxcav_30x20x4	14 454	89 185	5 758	1 653	103	1 550
boxcav_40x27x5	33 627	212 883	12 948	4 056	185	3 871

Table 5: Test matrices for null-space basis computations: size and deficiency.

The matrices are not as large as the matrices for analysing the performance of the parallel out-of-core solution. However the main issue with these matrices is the relatively large deficiency (rank of the null-space basis of the matrix) with respect to the order of the matrix (compare column 5 (Global Def) with column 2 (Order)). As shown in Section 8.2.2 of Part2, computing the null-space basis will require a large number of backward solutions with highly sparse right-hand-side since as many solution steps as the size of the deficiency must be performed. In columns 6 and 7 we indicate how the deficiency was detected during the factorization. As explained in Section 8.2.2 one part of the deficiency can be detected on the fly of a “quasi-normal" factorization phase (column Null Pivots) with modified pivoting strategies; another part can be detected while processing the root of the elimination tree with a rank revealing algorithm (column Root Def.). We will show in Section 8.2.2 that the locality of the deficient rows in the elimination tree influences the performance of our algorithms.

To illustrate the behaviour of our algorithms for computing entries in A^{-1}, our set of matrices is based on applications in astrophysics and results from a collaboration

with SPI/INTEGRAL team at CESR (Centre d'Etude Spatiale des Rayonnements in Toulouse). In the context of the INTEGRAL (INTErnational Gamma-Ray Astrophysics Laboratory [119]) mission of ESA (European Space Agency) a spatial observatory with high resolution (both in terms of angle and energy) hardware technology has been launched on October 2002. SPI [118] is one of the main instrument onboard INTEGRAL, a spectrometer with high energy resolution and indirect imaging capabilities. To obtain a complete sky survey with SPI/INTEGRAL, the processing of a very large amount of data acquired by the INTEGRAL observatory is needed [23]. For example, to estimate the total point-source emission contributions, a linear least-squares problem of about 1 million equations and 100000 unknowns must be solved. As already explained in this chapter (see previous Section 1.2 for least-square solution), one might want in this case to compute part of the inverse of the variance (see Equation 9). To do so one must then compute part of the inverse of the normal equation matrix A^TA where A is the matrix associated with the original linear least-squares problem. A few test matrices associated with the normal equations built from our application are shown in Table 6.

Matrix name	Order	Nb entries	Nb Nodes
a-1_08M	8 999	497 628	1 186
a-1_21M	21 532	855 866	5 207
d-11_25M	25 000	249 720	12 091
a-1_46M	46 799	1 791 242	12 419
a-1_72M	72 358	3 549 284	7 941
a-1_148M	148 286	7 388 031	12 734

Table 6: Test matrices to compute entries in $(A^TA)^{-1}$.

This application is computationally intensive because, in the context of the sky survey, all diagonal entries of the inverse of the normal equation matrix are required. For example, on the largest matrix in the test set, solving the complete problem requires about 7 seconds for the analysis phase, 1.4 seconds to factor the matrix and $14\,528$ seconds to compute all diagonal entries of the inverse of the matrix (results obtained at CESR with an incore factorization based on MUMPS solver on an Opteron 2.8 GHz with 16 Gbytes of main memory). We will show in Part 2 of the thesis how we can exploit the sparsity better in order to reduce the solution time and limit the memory used.

Part I

Analysis of the Solution Phase of a Parallel Multifrontal Approach

Résumé de la Partie I : Analyse de la phase de résolution parallèle dans une approche multifrontale

Le système linéaire de grande taille $Ax = b$ est résolu en utilisant une méthode directe de factorisation basée sur une approche multifrontale dans un environnement parallèle out-of-core (hors-mémoire). Les méthodes directes sont souvent composées de trois phases : une phase de prétraitement et d'analyse, une phase de factorisation ($A = LU$ ou $A = LDL$ si A est symmétrique) et une phase de résolution. La phase de résolution se décompose en une étape de descente $Ly = b$ suivie d'une étape de remontée $Ux = y$ dans le cas non-symétrique. Dans ce travail nous nous intéressons à la phase de résolution et aux possibilités de l'optimiser, surtout dans un contexte hors-mémoire où le temps de résolution est dominé par le temps d'accès et de lecture du disque dur.

Avant d'étudier et présenter des performances dans un contexte hors-mémoire, nous allons présenter certaines propriétés générales de la phase de résolution.

Dans un environnement en mémoire (in-core)

Les méthodes directes utilisent l'arbre d'élimination pour représenter la dépendance des calculs. Pour gérer l'ordre dans lequel les tâches de calcul sont effectuées nous utilisons une structure de données appelé POOL. Elle représente toutes les tâches prêtes à être exécutées à tout moment de la résolution. Au début de l'étape de descente (forward substitution, résolution de $Ly = b$) toutes les tâches associées aux feuilles de l'arbre de l'élimination sont stockées dans le POOL en respectant un post-ordre de parcours de l'arbre. Au début de l'étape de remontée (backward substitution, résolution de $Ux = b$ dans le cas d'une matrice symétrique), la seule tâche prête à être exécutée correspond au noeud associé à la racine de l'arbre. Dans les deux étapes (la descente et la remontée) les tâches mises dans le POOL sont extraites en utilisant l'ordonnancement LIFO, ce qui dans le cas séquentiel correspond à une traversée optimale de l'arbre - post-ordre des tâches. Dans le cas parallèle, l'extraction des noeuds du POOL est influencée par le mapping des noeuds sur les processeurs. Dans ce cas, le post-ordre ne peut plus être respecté et on parle d'ordonnancement topologique des tâches (chaque noeud père ne peut être activé qu'après avoir traité tous ses enfants).

Implémenter efficacement les méthodes directes multifrontales dans un environnement parallèle reste un travail difficile au niveau des communications entre les processeurs et la synchronisation des tâches à exécuter. Pour la première fois, une description détaillée des algorithmes parallèles utilisés dans la phase de résolution de la méthode multifrontale sera faite dans cette thèse. Des particularités importantes en parallèle seront illustrées (Propriétés 3.1, 3.2, 3.3 et 3.4) tout en prouvant leur efficacité pour la parallélisation massive de la méthode.

Dans un environnement hors-mémoire (OOC)

Il faut insister sur le fait que le temps de toute la phase de résolution est dominé par le temps d'accès et de préchargement du disque dur. Il devient alors primondial d'optimiser le processus de préchargement des données. Dans ce contexte, l' objectif de notre travail a été de diminuer le nombre d'accès au disque dur tout en rendant la lecture des données

la plus ‘régulière’ si possible. Une implémentation simple et efficace, dans le cas où la mémoire n’est pas critique, est d’utiliser le cache du système pour le préchargement des données. Dans le cas où la mémoire pour résoudre le système devient critique, les mécanismes du cache ne sont plus efficaces, comme indiqué dans le Tableau 1.7 - en diminuant le nombre de processeurs utilisés on observe un réduction du débit d’accès aux facteurs sur le disque (du 92.6 MB/s avec 8 processeurs à 9.2 MB/s en utilisant un seul processeur.)

Nprocs	Taille des facteurs (per proc) MB	Solution Parallèle Fwd (sec)	Bwd (sec)	Débit d’accès aux facteurs (MB/s)
In core				
8	317.5	0.9	0.9	—
OOC (Out-Of-Core)				
8	317.5	3.6	4.5	92.6
4	635.0	45.9	15.1	83.3
2	1 270.1	129.4	93.1	22.8
1	2 534.3	269.4	282.9	9.2

Table 1.7: Influence de la mémoire utilisée par noeud sur le Cray XD1 pour la performance en parallèle de la phase de résolution sur la matrice QIMONDA07. Cette approche OOC est basée sur la simple utilisation de mécanisme cache (SYSTEM_BASED approche).

On propose donc une autre méthode pour lire les données sur le disque (méthode appelée Direct I/O), plus contraignante pour le développeur, mais beaucoup plus efficace du point de vue du temps d’accès et de la gestion de la mémoire. Des buffeurs internes au programme sont destinés à précharger les données du disque. Le grand avantage de cette approche est que leur taille est indépendante de la taille du problème et qu’elle peut être fixée par l’utilisateur. Le buffeur est divisé en deux parties - une partie pour un préchargement d’un grand nombre de données en utilisant des méthodes sophistiquées d’optimisation; et une partie de lecture sur un seul bloc de données en urgence (lecture en mode bloquant) (voir Figure 1.12).

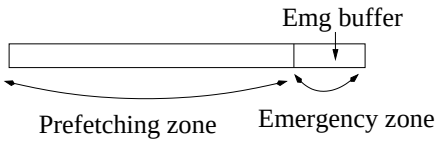


Figure 1.12: Buffeur dont la taille est prédéfinie par l’utilisateur.

Une comparaison entre les deux méthodes en terme de temps de calculs et de nombre d’accès au disque dur est donnée dans le Tableau 1.8.

Méthode	Fwd (sec)	Bwd (sec)	Nb_Req Fwd Prefetch	Nb_Req Fwd Emg zone	Nb_Req Bwd Prefetch	Nb_Req Bwd Emg zone
DIRECT_IO (Emg+Prefetch)	171.5	176.8	541	0	496	0
SYSTEM_BASED	269.4	282.9	—	—	—	—

Table 1.8: Influence du nombre des buffeurs sur la rśolution sequentiel de QIMONDA07. Fwd=forward phase. Bwd=backward phase. Emg zone: 1 MB; Prefetch buffer: 10 MB.

Après une comparaison exhaustive, nous avons démontré l’efficacité de la méthode DIRECT_IO sur l’ensemble de nos matrices.

Ordonnancement (Scheduling)

Comme nous l'avons déjà dit précédemment, l'accès régulier au disque est extrêmement important pour le temps de calcul de la phase de résolution. Par accès régulier, on sous-entend le préchargement des données de grande taille d'une manière contiguë sur le disque dur. Dans ce contexte, l'ordonnancement efficace des différentes tâches prêtes à être exécutées devient très important. Dans le cas séquentiel, l'ordre optimal pour parcourir l'arbre d'élimination et traiter les tâches correspond à un post-ordre. Dans le cas parallèle, les choses se compliquent en introduisant le mapping des tâches sur les différents processeurs, et donc le post-ordre ne peut plus être respecté. Les stratégies connues jusqu'à présent, LIFO et FIFO, ne sont pas adaptées non plus à la résolution parallèle du système à cause du grand nombre d'appels irréguliers au préchargement des données du disque. (Plus de $400\,000$ appels dans l'étape de backward substitution avec 3 et 4 processeurs).

Stratégie	Nb of Procs	Fwd (sec)	Bwd (sec)	Nb Max requêtes par step			
				Fwd $^{(*)}$		Bwd $^{(*)}$	
				Prefetch	Emg zone	Prefetch	Emg zone
LIFO	1	171.5	176.8	541	0	496	0
LIFO	3	64.9	262.1	190	3	169	422 497
LIFO	6	38.0	186.7	102	6	86	422 498
LIFO	8	24.9	137.6	70	0	64	321 871
LIFO	16	13.2	94.4	39	2	32	214 245
LIFO	24	10.9	48.5	42	5	38	119 792
LIFO	32	9.1	53.1	25	1	30	116 209

Table 1.9: Influence de l'ordonnancement LIFO sur la matrice QIMONDA07. Emg=emergency buffer:1 MB; Prefetch buffer:10MB par processeur; $^{(*)}$: Max par processeur.

Nous proposons une nouvelle stratégie d'ordonnancement des tâches, NNS, basée sur le stockage des tâches sur le disque dur. Elle prend en compte la répartition des tâches sur les disques locaux de chaque processeur. En ordonnançant les tâches maîtres afin de respecter la séquence de chaque processeur, on arrive à reproduire la séquence d'écriture des facteurs lors de la phase de factorisation. Ainsi on obtient un accès beaucoup plus régulièr en lecture aux données du disque dur et un temps de la résolution fortement réduit.

Stratégie	Nb de Procs	T_min (sec)	Bwd (sec)	Nb_Req$^{(*)}$ Bwd	
				Prefetch	Emg
NNS	1	158.4	177.2	496	0
NNS	3	57.9	65.5	174	1
NNS	6	31.5	37.9	93	0
NNS	8	21.8	45.2	57	0
NNS	16	11.9	13.8	36	0
NNS	24	9.0	13.2	38	0
NNS	32	8.2	10.7	34	0

Table 1.10: Influence de l'ordonnancement NNS sur la matrice QIMONDA07. Emg=buffer d'urgence:1 Mo; Prefetch buffer:10Mo par processeur; $^{(*)}$: Max per processor.

Les tableaux 1.9 et 1.10 montrent les performances des deux ordonnancements LIFO et NNS. Dans les deux cas, on compare le temps obtenu (Fwd et Bwd) avec le temps minimum pour charger les facteurs du disque dur (T_min). On montre aussi le nombre des préchargements du disque. Comme le montre le Tableau 1.10 sur la matrice

QIMONDA07 mais aussi sur l'ensemble de nos matrices, l'ordonnancement NNS s'est montré plus efficace à réduire le nombre de préchargements du disque et le temps global de la phase de résolution.

Chapter 2

Introduction

We are interested in solving large sparse linear systems $Ax = b$ with direct methods [45, 47, 78], in a parallel limited-memory environment. We suppose that the original matrix A is first factorized into the factors LDL^T (when A is symmetric) or LU (when A is unsymmetric), where L and U are triangular matrices and D is diagonal (or block diagonal with blocks of order 1 or 2 in the case of numerical pivoting for indefinite systems). Note that in our factorization expressions, we have omitted, for the sake of clarity, the permutations performed to preserve sparsity and to implement numerical pivoting. These factors are then used to solve the system through the forward and backward substitution steps

$$\left[\ LDy = b \ \text{ and } \ L^T x = y\ \right] \quad \text{or} \quad \left[\ Ly = b \ \text{ and } \ Ux = y\ \right], \tag{2.1}$$

depending on whether the matrix is symmetric or not. In this work, we are concerned with the case when the matrix A is large and sparse [47, 59]. The main limitation in the use of sparse direct methods comes from the need to store the factors that often have many (10 to 100 times) more entries than the original matrix.

Usually the most time consuming part of the solution process is in the initial matrix factorization and it is this step that most previous work has addressed. However, in many applications, the substitution phases can be performed very many times for each factorization so that the accumulated time for these phases dominates. This is true, for example, in some algorithms for nonlinear optimization and for applications where solutions with many different right-hand sides are required (for example, in electromagnetic or seismic modelling). Furthermore, when solving systems in parallel or when working out-of-core, the substitution times can be greatly increased. We believe this is the first in depth study of the substitution phases in a parallel and out-of core environment. Our work differs and extends the work of [103, 104, 105] and [114] because firstly we consider a parallel out-of-core context, and secondly we focus on the performance of the solve phase. In this context, an out-of-core **(OOC)** multifrontal [51, 52] approach is considered. Here the factors are written to disk during the factorization phase, as a sequence of blocks (that we call **factor blocks**). Overlapping communications and I/O with computations during the factorization phase is an important issue (see [2]), but is not the scope of this work. During the subsequent forward and backward solve phases, that we call **solve phase**, we have to load the factor blocks from the local disks of the computer to the main memory. In this context, the cost of the solve phase can become the dominant phase of the complete solution process. When the solve

phase has to be performed for many right-hand sides (simultaneously or not) then it is even more critical.

We first discuss in Chapter 3 the main aspects of the in-core distributed memory solve phase: mono-processor and multi-processor case. Although details of our solver MUMPS have described in previous publications [7, 10, 12] this is the first time we have considered the solve phase in detail. We explain why our parallel solve phase does not follow the standard dependency structure of the factorization phase and prove the correctness of our approach. We then explain how our algorithms have been adapted to the out-of-core context in Chapter 4. We show the limitations of a simple demand driven approach, that we call SYSTEM_BASED, based on automatic system I/O caching mechanisms. In Chapter 5 we show how user buffers can be introduced to improve the behaviour of the solve phase and then describe an approach where the memory used is completely controlled, which we call the DIRECT_IO. We show that a naive implementation of the DIRECT_IO based approach is not suitable for parallel implementation and introduce a new scheduling scheme that constrains the ordering of the tasks. We first prove that the new algorithm is correct. We then illustrate in Chapter 6 the gain in performance obtained on a set of large real problems.

Chapter 3

Main in-core parallel features of the solver

3.1 Introduction

Direct solvers try to preserve the zero pattern and to exploit the independence of some computations in parallel environments. So called three-phase approaches have become very popular:

- The **analysis** phase considers only the pattern of the matrix and builds the necessary data structures for numerical computations.
- The **factorization** phase tries to follow the decision of the analysis and builds the sparse factors (LU for unsymmetric case, or LDL^T for the symmetric case).
- The **solve** phase performs forward and backward substitution phases and optionally performs iterative refinement to improve the solution.

We will start by introducing the factorization phase (Section 3.2) and basic notions, used later during the solve phase (Section 3.3). We assume in this Chapter that our matrix has a symmetric structure and thus even when the matrix is unsymmetric $Struct(L) = Struct(U)$.

3.2 In-core parallel factorization phase

Multifrontal methods use an **elimination tree** [92] to represent the dependencies of the computations. Based on the structure of the L factors, we define the elimination tree as follows: node i is the father of node j if and only if i is the first non-zero entry in column j of L. Each node of this tree is associated with a **frontal matrix** that is assembled (summed) based on contributions from the children and the entries from the original matrix. In practice, nodes of the elimination tree are amalgamated so that more that one variable can be eliminated at each node of the tree. The resulting amalgamated tree is referred to as the **assembly tree**. The work associated with an individual node of the assembly tree corresponds to the factorization of the frontal matrix. Frontal matrices are always considered as dense matrices (see Figure 3.1).

Once all **eliminations** for a node have been performed, the Schur complement matrix $F_{22} - F_{21}F_{11}^{-1}F_{12}$ is computed. It is used to update later rows and columns of the overall matrix which are associated with the parent nodes. We call this Schur complement matrix the **contribution block** (**CB**) of the node, see Figure 3.1.

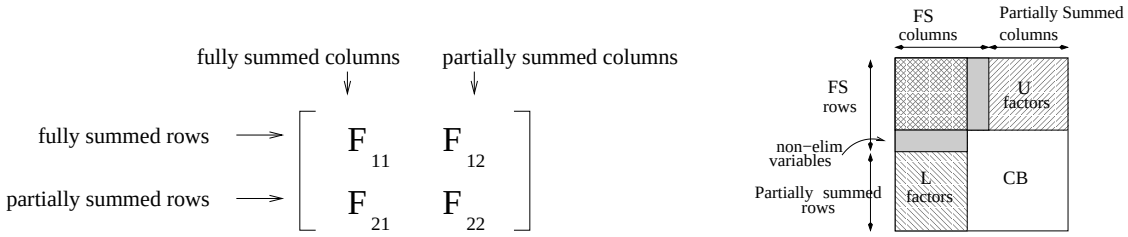
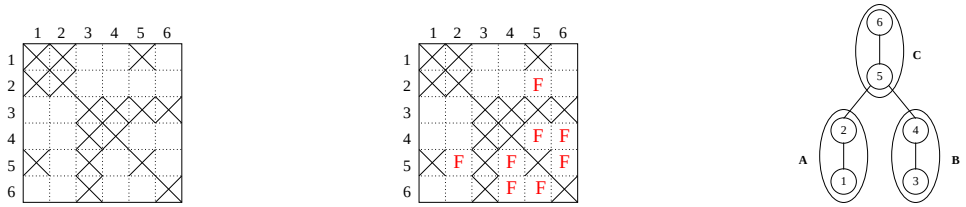


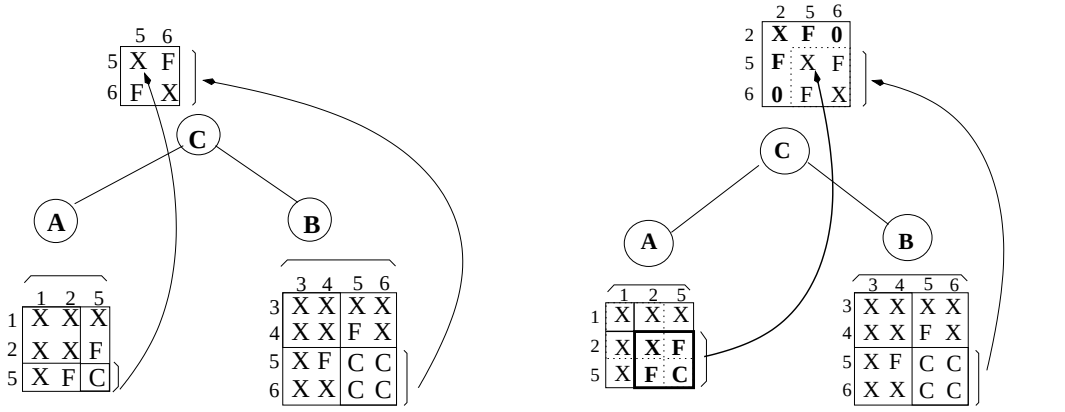
Figure 3.1: Frontal matrix : general structure

If some variables are not eliminated because of numerical issues, they are included in the contribution block and their elimination is postponed to the parent node or later to an ascendant node (Figure 3.2). These **non-eliminated variables** (delayed pivots) increase the fill-in the factors, the number of the operations and the factorization time, but can be critical to the accuracy of the solution.

We show in Figure 3.2 the difference of the factorization in both cases: with and without non-eliminated variables. We first show in Figure 3.2-a) a symmetric matrix pattern, its associated factors and the corresponding assembly tree. Then the frontal matrices of all nodes in the assembly tree are presented. Figure 3.2-b) shows the case when there is no delayed pivots. Nodes A and B are completely factorized and the contribution of column 5 and 6 are sent to node C. Figure 3.2-c) shows the case when there column 2 is not factorized and thus becomes a delayed pivot. The contribution block of node A is thus extended and sent to the parent node C which frontal matrix size is also updated.



a) Original matrix, LU structure and corresponding assembly tree



b) Case of non-delayed pivots *c) Case of delayed pivot 2 from node A to node C*

Figure 3.2: Data structure of the frontal matrix and the eliminations performed between two children and a father node. X = non-zero positions of the original matrix; F = fill-in; C = Contribution Block entry.

During the factorization phase factors associated with each node of the elimination tree are written on disk, as they will be accessed again only in the solution phase. Thus the amount of needed/active memory for factorizing the nodes is lower (as shown in Figure 3.3) and we can benefit of better efficiency. We use the term active memory to reference the memory needed to store the current frontal matrix and the contribution blocks computed to the moment.

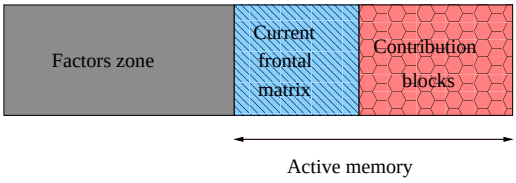


Figure 3.3: In-core memory needed to factorize a matrix is freed from factors.

3.2.1 Parallelism during the factorization phase

Note that, in a sequential environment, we choose to process the nodes of the elimination tree, using a post-ordering (nodes belonging to any subtree are numbered consecutively). In a sequential environment with a post-ordering it can be shown than a simple stack mechanism can be used to manage the working space associated to the contribution blocks, as shown in Figure 3.4

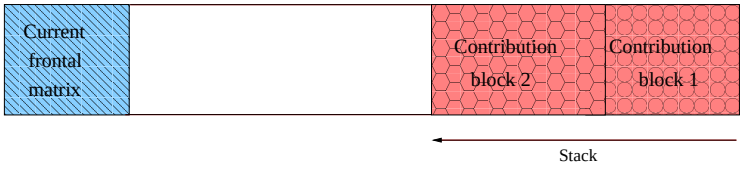


Figure 3.4: In-core memory with the current frontal matrix and a stack with two contribution blocks.

In a parallel environment, the tree nodes are distributed onto processors and only a topological ordering is followed (nodes of any subtree are numbered consecutively, but are not always processed consecutively. Nevertheless, a parent node can be process only after all its children).

The multifrontal MUMPS solver [7, 10, 12] provides three different types of parallelism for both factorization and solve phases. The reason for the three types is to balance the total work and the memory on each processor.

We use the assembly tree, representing the order in which the matrix will be factorized, to distribute the nodes over the processors. Depending on the size of the node and on which level of the assembly tree the node is situated, we have :

Type1 node: **sequential processing of a node** — essentially for the low levels of the tree (near the leaves), where the tree parallelism is sufficient.

Type2 node: **irregular 1D decomposition of the node** — for the intermediate levels when the node is large enough: the contribution blocks are partitioned and each partition assigned to a different processor with respect to the total amount of data mapped on each processor. The so called **master** process is in charge of factorizing the block of fully summed variables and of deciding how many **slave** processes will be used to process this node. Data equilibration is thus done among the processes [72, 73].

Type3 node: **block cyclic 2D distribution [31] of the frontal matrix** — reserved only for the root node, if it is large enough. In this case, ScaLAPACK [22] is used on the node.

In Figure 3.5 the pattern of symmetric factors with a post-ordering of the nodes in the elimination tree is shown. We show the mapping of the nodes on four processors. Each node in the elimination tree has a master process, indicated in a box. If a node is of type 2 or type 3, slave processes are also associated, using dynamic decision during the factorization phase to equilibrate factors among processors. We show also the distribution of the L factors of a frontal matrix depending on the type of the node. For type 1 node, the whole frontal matrix is mapped to one processor, as shown for node 5. If a node is of type 2, as node 3, the frontal matrix and the contribution block are divided between the master and the slave processes in irregular *1D* decomposition. The frontal matrix on the root, node 7, is divided in block cyclic *2D* decomposition.

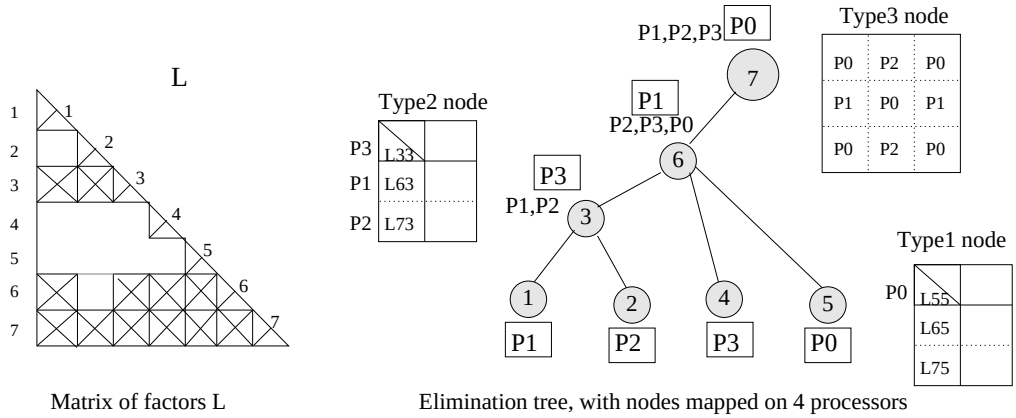


Figure 3.5: Illustration of node types and mapping of the processors on the nodes in the elimination tree. In box is indicated the master process associated with each node.

3.3 In-core parallel solve phase

Our algorithms handle both symmetric and unsymmetric matrices. For the unsymmetric or indefinite matrices the algorithms incorporate numerical pivoting (threshold partial pivoting and two-by-two pivots). Note that although the pivoting is a part of the factorization phase, the permuted row/column factors should have to be taken into account in the solution phase. For the sake of clarity we will focus in the following on symmetric matrices and will not consider pivoting for numerical stability in the description of our algorithms. Our solve phase follows a multifrontal factorization and uses the assembly tree [92] to represent the dependencies of the computations during the solution phase.

3.3.1 Some notation

During the solve phase, each node of the elimination tree holds the L factor block computed during factorization. The **forward substitution** is a bottom-top traversal of the tree (post-ordering for the sequential case and topological ordering for the parallel case). The **backward substitution** traverses the tree in the reverse order. The factor block can be partitioned into **factored** variables and **unfactored** variables as shown in Figure 3.6.

In our parallel context, the distribution of the L factors depends on the type and the mapping of the nodes onto the processors as explained in the previous section.

We first comment on data structure used for task scheduling. Then we describe, separately, the forward and the backward algorithms, identifying the critical issues in each case.

3.3.2 Algorithm for management of tasks and messages

To handle the task dependency graph, both the forward and backward algorithms make use of a distributed pool of tasks, that we call the POOL. This pool contains a list of all ready tasks to be executed and is used to schedule work in both the sequential and

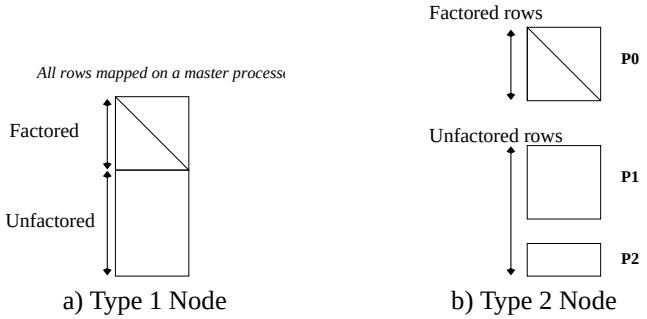


Figure 3.6: Partitioning and distribution of the factor blocks depends on the type of the node. On the Type 2 node, P0 is the master process in charge of factored row variables, and P1 and P2 are slave processes in charge of a partition of the unfactored row variables.

the parallel cases. At the beginning of each step, we initialize the distributed pool with all tasks ready on each process using a post-order (see Figure 3.7 for a description of the situation on one process). Tasks are then extracted from the end of the pool (LIFO strategy).

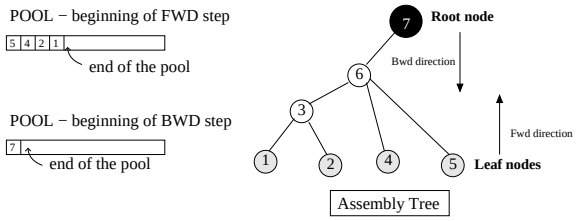


Figure 3.7: The POOL of tasks at the beginning of the forward and backward sequential solve steps.

For the forward step, the pool is initialized with the leaf nodes of the assembly tree. A node will then be placed at the end of the pool as soon as all of its children are processed. Note that in a sequential context this lead to a post-order traversal of the tree. At the beginning of the backward step the pool is initialized only with the root nodes. At the end of a node process, we add to the end of the pool all of its children. Furthermore, for both the forward and the backward steps, when a node is distributed over more than one process (Type 2 or Type 3 nodes) only the master task is added to its local pool. The slave tasks are processed on the fly. The algorithm for extracting nodes from the pool is described in Algorithm 3.1.

Note that priority is given to the reception of messages - to a blocking or non-blocking receive. We look at the pool for work only when no message need be processed. The algorithm for the forward case finishes when all root nodes have been treated. The backward algorithm finishes when all leaf nodes have been processed.

We first describe in Algorithm 3.2 the parallel forward substitution ($LDy = b$) and later present the algorithm for the backward step ($L^T x = y$) in Algorithm 3.3.

Algorithm 3.1 : Algorithm for extracting a node from POOL (LIFO strategy)

$Myid$ - process number; *Inode* - the current node mapped on process $Myid$;

```
1:  Step = Fwd or Bwd
2:  if ( Fwd ) Initialise POOL with the leaf nodes mapped on Myid
3:  if ( Bwd ) Initialise POOL with root nodes mapped on Myid
4:  while (Not finished) do
5:     if (POOL is not empty) then
6:        if a message is available Process_Message(message)        [See Algorithms 3.2 and 3.3]
7:     else
8:        Wait for a message and then Process_Message(message)      [See Algorithms 3.2 and 3.3]
9:     end if
10:    if (POOL is not empty and Process_Message not called) then
11:       Extract node, say Inode, from the end of POOL
12:       if ( Fwd ) Fwd_Process_node(Inode)   [See Algorithm 3.2]
13:       if ( Bwd ) Bwd_Process_node(Inode)   [See Algorithm 3.3]
14:    end if
15: end while
```

3.3.3 Algorithm for forward substitution

We first describe in Algorithm 3.2 the parallel forward substitution ($Ly = b$). We then show in Section 3.3.4 details of the algorithm used to process a node and add comments on how messages are processed.

Note that in Algorithm 3.2 (and in practice) the same working space can be used to store both y and b. We will keep two separate vectors in our algorithm only for the sake of simplicity.

To better understand the distributed memory version of our algorithms, we introduce a few properties related to the use of the elimination tree. For properties 3.1 and 3.2 note that the terms factored and unfactored variables were described in Figure 3.6. We show (Property 3.2) that our algorithm does not always follow the dependency paths of the assembly tree which explains why we must reset our working array Wb to zero.

For the sake of completeness references to BLAS (Basic Linear Algebra Subroutines) kernels (`GEMM/V` and `TRSM/V`) have been added to the description of the Algorithm 3.2 and Algorithm 3.3 (algorithm for the backward substitution). Notations 'step i' refer to Figure 3.9 in Section 3.3.4 where we illustrate in details the main steps of our algorithm.

Without loss of generality we will assume in the remainder of the first part of this thesis that we have only one right-hand side and thus one solution to compute since the extension to multiple right-hand sides is straightforward.

Property 3.1. *All updates to factored variables of a node, say Inode, come only from processes involved in the children of Inode (both master or slave processes).*

Proof: This property is clearly preserved by the algorithm, since in our algorithm only processes involved in the children send updates to the master of the father - message `ContVec` or direct update of Wb either during **Fwd_Process_Node** for Type 1 nodes or at the reception of message `MASTER2SLAVE` for Type 2 nodes. Furthermore updates to the factored variables of a node can only come from nodes involved in the subtree rooted at that node (main property of the assembly tree). This proves our property. $\diamond$

Algorithm 3.2 : Algorithm for the forward step ($LDy = b$)

$Myid$ - process number; *Inode* - the current node mapped on process $Myid$;
$Nb_children$ - the number of children of *Inode* and
Pfather - the process on which the master of *father(Inode)* is mapped.
Wb - a local working array, initialized to 0 and designed to accumulate modifications of the right-hand side b ;
`Use_factors` will be expanded in later discussion to cases when such use or access to them is non-trivial.

Fwd_Process_node(*Inode***)** {I am the master of node *Inode* }

1: For factored variables, update b with entries of Wb and `Use_factors` to compute the partial solution (TRSM/V) (step 1 and 2)
2: **if** (*Inode* is of Type 2) **then**
3: Send to each slave of *Inode* the computed solution and entries of Wb corresponding to variables mapped on this slave (message MASTER2SLAVE) and reset these entries of Wb to zero (*see Property 3.2*) (step 3)
4: **else if** (*Inode* is of Type 1) **then**
5: Update Wb for unfactored variables (GEMM/V)
6: **if** (*Myid* $\neq$ *Pfather*) **then**
7: Send updated entries of Wb to *Pfather* (message ContVec) and reset them to zero (*see Property 3.2*)
8: **else**
9: Increment updates for *Pfather* and if last update add $father(Inode)$ to the end of POOL
10: **end if**
11: **else**
12: Type 3 root node process based on ScaLAPACK for both forward and backward steps on all processes
13: **end if**

Process_Message(Message) {I am updating *Inode*}

1: **if** (Message = ContVec) **then**
2: Update Wb with contribution received; Increment number of updates
if last update, add *Inode* to the end of POOL
3: **else if** (Message = MASTER2SLAVE) **then**
4: Gather in a small *local array* entries of Wb just received
5: `Use_factors` and the solution sent by the master to update the *local array* (GEMM/V)
6: **if** (*Myid* $=$ *Pfather*) **then**
7: Scatter and add the *local array* in Wb
8: Increment number of updates and if last update add *Inode* to the end of POOL
9: **else**
10: Send *local array* to *Pfather* (message ContVec)
11: **end if**
12: **end if**

Property 3.2. *All updates of descendants of a node Inode, to unfactored variables of a node are not always sent to processes in charge of that node.*

Proof: Figure 3.8 will be used to prove our property. All nodes in Figure 3.8 are Type 1 nodes. Node 1 (mapped onto P1) sends to node 4 (mapped onto P0) updates to Wb (corresponding to entries of Wb on P1) and resets those entries to zero. Node 2 (mapped onto P0) updates Wb and sends its updates to P2 (corresponding to entries of Wb on P0) and resets those entries to zero. At this point, part of the updates of the sub-tree rooted at node 5 will circulate through node 6 on P2. This is the case if both node 1 and node 2 have a common row in the factor block of node 7. This update to Wb will then be sent to P4 by P2 during the processing of node 6. On our test matrix (see Figure 3.8) the

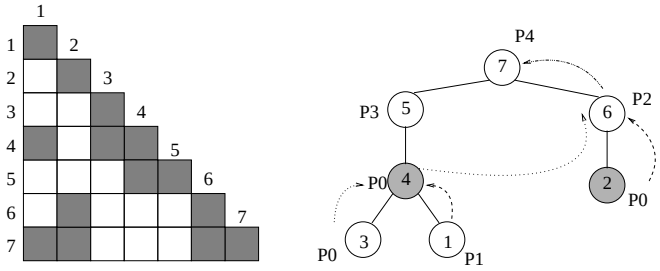


Figure 3.8: Example used to prove Property 3.2: part of the updates of node 1 are not sent to process P3 in charge of node 5.

contribution due to the zero entries l_{71} and l_{72} in row 7 available on processor $P0$ will circulate through node 6 on $P0$.

As a consequence during the processing of node 4, process P0 will not send to its father (node 5) all updates from node 1 to node 7. Instead Property 3.1 says that the common row updated by node 2 and node 4 could not be eliminated at node 5, but at node 7. $\diamond$

Note that Property 3.1 is one of the main properties of the elimination tree, exploited by the multifrontal approach and preserved, on each process, by the algorithm for the factored variables. However, contrary to what is exploited during multifrontal factorization, this elimination tree property is no longer respected on each process for unfactored variables (Property 3.2). Property 3.2 also explains the importance of resetting Wb to zero in Algorithm 3.2.

Property 3.3. *At any time a computed update is stored in the Wb array of only one process.*

Proof: We recall that Wb is designed to sum update vectors. Wb is first initialized to zero on each process at the beginning of the forward step. It corresponds to updates to the right-hand side b due to solution terms already computed. Each time part of Wb is sent to a process (message `ContVec` or `MASTER2SLAVE`) then the corresponding entries are reset to zero in the procedure **Fwd_Process_node**.

Let us now check that updates to Wb are never lost. First, during the function **Process_Message**(`MASTER2SLAVE`), each slave gathers in a local array contributions sent by its master. This local array is either used to update Wb locally, if the $process_id$ of the slave is equal to *Pfather*, or is forwarded (message `ContVec`) to process *Pfather* without updating Wb locally. $\diamond$

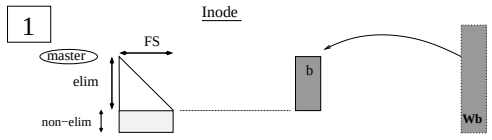
Property 3.4. *When starting to process a node (first line of procedure* **Fwd_Process_node***(Inode)) of Algorithm 3.2, b holds all contributions needed to compute the solution corresponding to the factored variables of the node.*

Proof: Results from Property 3.1 and 3.2. $\diamond$

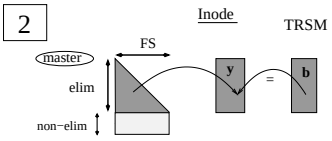
Corollary 3.1. *Property 3.4 recursively proves that Algorithm 3.2 computes the correct solution.*

3.3.4 Detailed illustration of the forward substitution

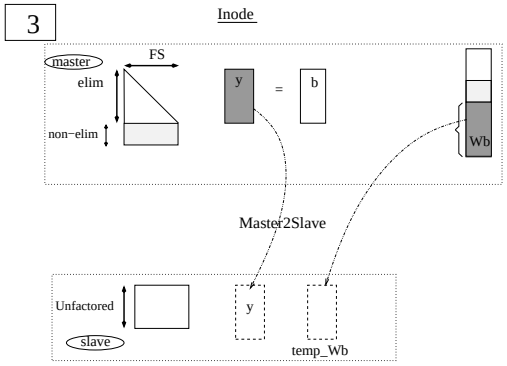
Figure 3.9 is used to graphically represent the main steps of the algorithm of the parallel forward substitution. A small example is then provided (Example 3.1) to further explain the algorithm.



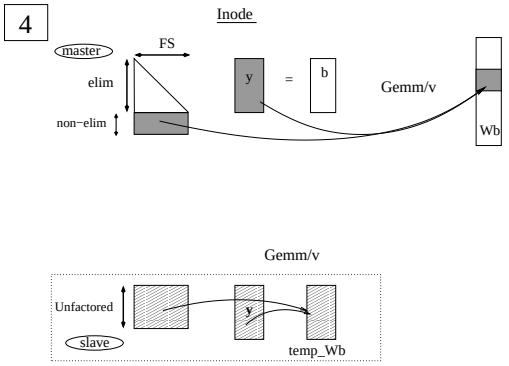
Updates b with entries of Wb



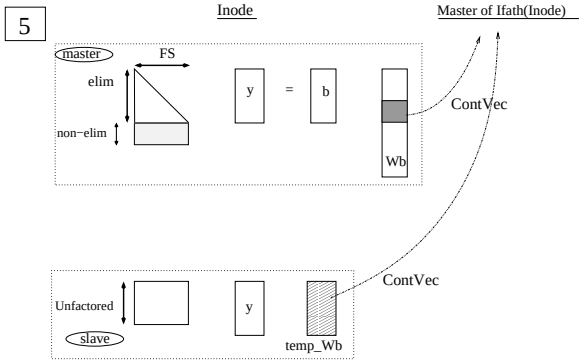
Computes the solution corresponding to the eliminated variables



Sends the solution and entries of Wb to the slave



Master and slaves compute an update to b



Send contributions to the Pfather

Figure 3.9: Communication pattern of procedure Fwd_Process_Node($Inode$) (see Algorithm 3.2) in the case of Type2 Node with a single slave. *temp_Wb* corresponds to the local array referred in Algorithm 3.2 when processing message MASTER2SLAVE.

In Figure 3.9 we present the main steps of processing a node of Type 2, mapped on two processes: a master with one slave. On the master process are mapped all factored (fully

summed FS) variables: eliminated (*elim*) and non-eliminated (*non-elim*) variables, as defined in Figure 3.1. All unfactored variables are mapped on the slave process.

In Figure 3.9 we reference each step of Algorithm 3.2. At each step $\boxed{\text{i}}$, the concerned parts are coloured in gray. At step $\boxed{1}$ node $Inode$ becomes ready and the master process updates b with entries of its vector of contributions Wb. At step $\boxed{2}$ the master process uses the eliminated variables and the updated b vector to compute the corresponding solution. At step $\boxed{3}$ the master sends the computed solution together with entries of the contribution vector Wb to the slaves (`MASTER2SLAVE` message). At $\boxed{4}$ each process (master or slave) computes an update to b. For the master process it corresponds to non-eliminated variables which are then stored in Wb. For the slave, it corresponds of unfactored rows, stored temporary in $temp_Wb$ At $\boxed{5}$ the updates are sent to the father's master process via `ContVec` messages. The father's node becomes ready when both contributions are received.

Note that there are two different ways to send the contribution vectors to the father of $Inode$. The first way (our current proposed scheme) is to have each slave of $Inode$ to sum the contributions and then send them directly to the father of $Inode$. In this case we have divided the large messages into many small ones - see Figure 3.10 - a). The second way is to send the contribution vector at once, directly from the master of $Inode$ to the master of its father. Note that in this case the contribution computed by each slave still need be send by each slave.

In both cases the volume of data transfered is identical. Furthermore the master has anyway to send the computed solution to each slave. This means that the number of messages for both schemes remains the same. The first scheme (chosen in our algorithm) should thus be more efficient because on can expect natural parallelisation of the data transfer of the known solution stored in Wb by the master ('one to many' compared with 'one to one' of the same total volume of data with the same number of messages).

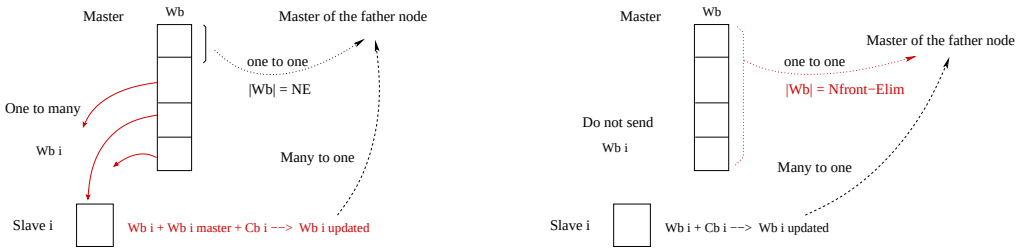


Figure 3.10: Different possibilities of sending contribution vectors to the master node.

Example 3.1. Node processing during the Fwd step ($Ly = b$) *(see Figure 3.11)*

Node 3 is a type 2 node that has received all contributions from its children and has been added to the pool.We assume that the master of node 3 is mapped on processor P3 and has slaves mapped on processors P1 and P2. P3 updates b with the contributions to b stored in Wb. P3 computes eliminated variables of y_3, and sends them together with part of Wb to each slave processor (stored locally in temporary array $temp_Wb$). Each slave updates $temp_Wb$. P2 sends the updated block ($temp_Wb$) to the master of node 6 (P1) via a `ContVec` *type message. The slave P1 mapped on node 3 can directly update Wb, because P1 is the master of the father node 6.*

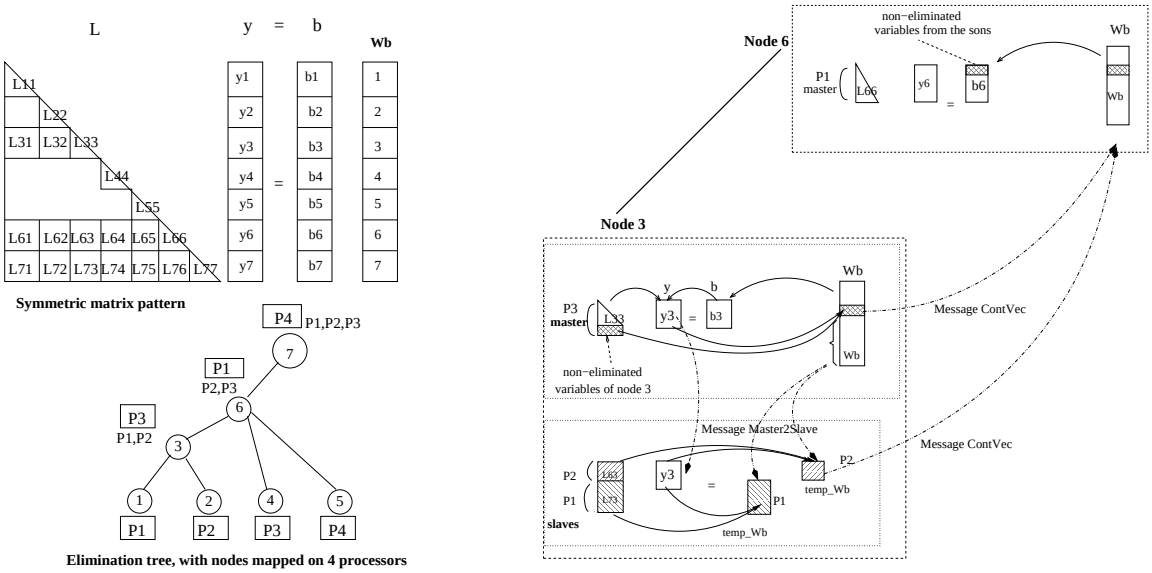


Figure 3.11: Communication pattern and data computation during the forward step on a node of Type2.

Node 6 The master of node 6 (P1) initialises the number of contribution vectors to $Nb_children(node\ 6) + \sum slaves(node\ 6)$. For each message of type `ContVec` *received, P1 updates Wb and decrements the number of contribution vectors. When all contributions are received, node 6 is added to the local pool of P1.*

3.3.5 Algorithm for backward substitution

The algorithm for the backward substitution ($L^T x = y$) is described in Algorithm 3.3. As for the forward step, priority is given to message reception. If no message is received, a node from the pool is extracted. The backward step manages three types of messages: `Bwd_MASTER2SLAVE` and `Bwd_ContVec` are similar to `MASTER2SLAVE` and `ContVec` of the forward case respectively; a new type of message `Bwd_Node` is used to control the activation of the children.

During the backward step, when a Type 2 node is processed, the slave processes are first involved in the updating of the right-hand side y (after reception of message `Bwd_Master2Slave` from the master process of that node). Once the master process has received all updates to the right-hand side computed by the slaves (message `Bwd_ContVec`), the solution associated with the factored variables is then computed. A message `Bwd_Node` is then sent to each process on which at least one master node of the children is mapped. Note that even if several nodes are mapped on the same process, messages `Bwd_Node` will be sent only once to this process.

Figure 3.12 will be used to illustrate in more details the main steps of our algorithm in Section 3.3.6.

Algorithm 3.3 : Algorithm for the backward step ($L^T x = y$)

Myid - process number; *Inode* - current node mapped on *Myid*;

Bwd_Process_Node($Inode$ **)**
if (*Inode* is of Type 2) **then**
  Master distributes already computed solution corresponding to factored variables between the slaves of *Inode* (message `Bwd_MASTER2SLAVE`, step 1)
else if (*Inode* is of Type 1) **then**
  `Use_factors` associated with unfactored variables to update y (`GEMM/V`) and with factored variables to compute solution (`TRSM/V`)
  for each child of *Inode* whose master process is mapped on *Myid*, add it to the end of POOL
  Send the solution corresponding to **all** variables of *Inode* to processes on which at least one master of a child node is mapped (message `Bwd_Node`)
end if

Process_Message(Message)
if (Message = `Bwd_Node`) **then**
  Update known solution and add to POOL *Inode* and all of its brothers whose master process is mapped on $Myid$
else if (Message = `Bwd_MASTER2SLAVE`) **then**
  `Use_factors` mapped on this slave process together with the received solution to compute a contribution to y (`GEMM/V`, step 2) and send it to the master of *Inode* (message `Bwd_ContVec`, step 3)
else if (Message = `Bwd_ContVec`) **then**
  Update y with message and increment number of updates received (step 3 and 4)
  if last update **then**
    `Use_factors` associated with factored variables to compute the solution (`TRSM/V`, step 5)
    for each child of *Inode* whose master process is mapped on *Myid*, add it to the end of POOL
    Send the solution corresponding to **all** variables of *Inode* to processes on which at least one master of a child node is mapped (message `Bwd_Node`, step 6)
  end if
end if

3.3.6 Detailed illustration of the backward substitution

Figure 3.12 is used to graphically represent the main steps of the algorithm of the parallel backward substitution. A small example is then provided in Figure 3.13 to further explain the algorithm.

As done for the forward substitution, Figure 3.12 is divided into steps related to the communication pattern of the processing of a Type 2 node. We assume that the node is mapped on 3 processes - a master and 2 slaves.

At the reception of message `Bwd_Node`, the master process holds the complete solution sent from its father. The node becomes ready and is added into the pool. Once extracted from the pool, at step 1 the master distributes the received solution x' between the slaves involved in the computations on this node. At step 2, after reception of message `Bwd_MASTER2SLAVE`, the slaves use the sent part of the solution to compute updates to y. Then, at step 3, the slaves send back the contribution to y to the master via message `Bwd_ContVec`. The master updates y with all received contributions, at step 4. At step 5, the master computes his contribution to y related to the non-eliminated variables. Finally, the master uses the eliminated variables and the updated right-hand side y to

compute a part of the solution vector x. The updated known solution is then sent at step 6 to all processes involved in the master processing of at least one son.

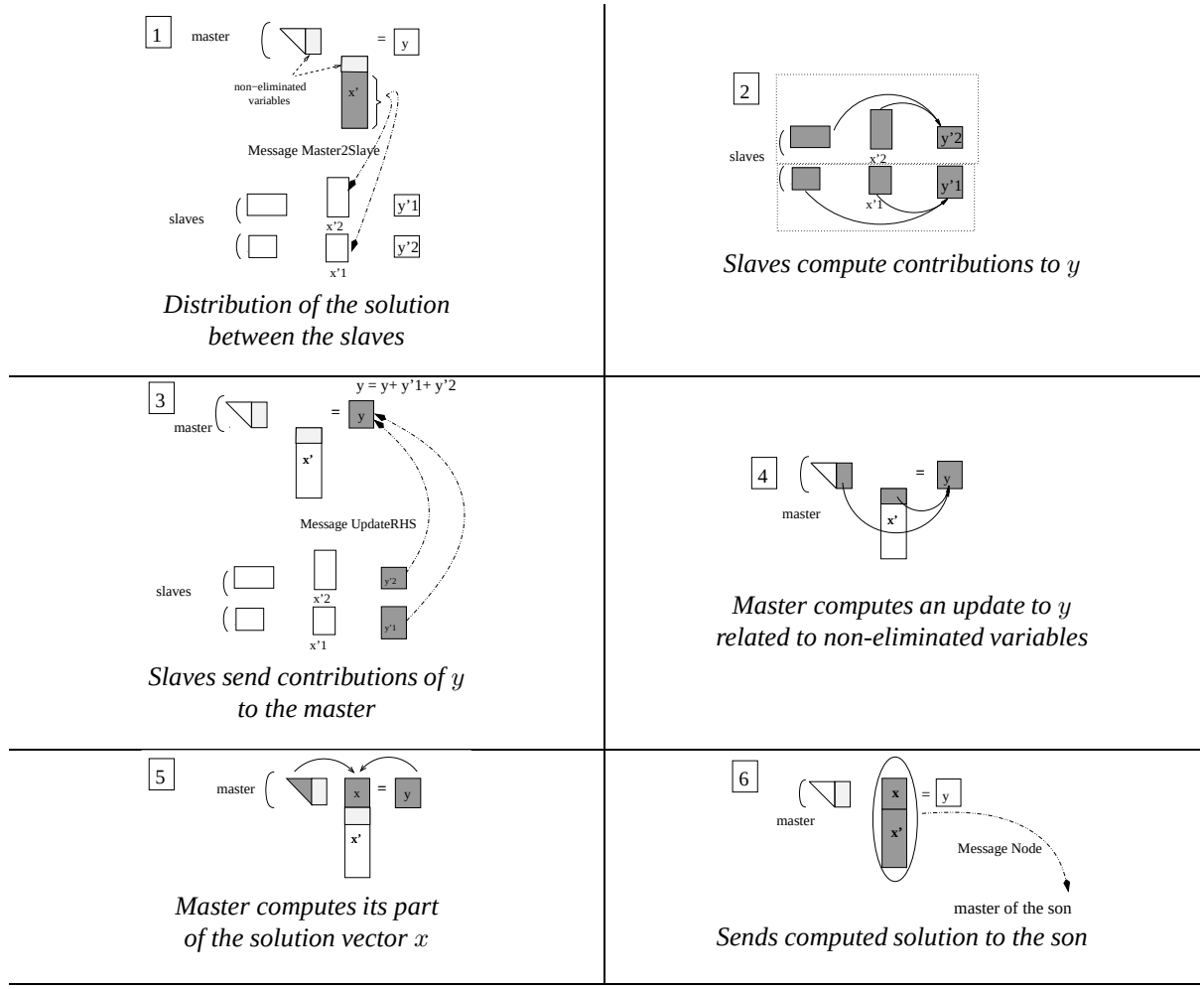


Figure 3.12: Communication pattern of procedure Bwd_Process_Node($Inode$) (see Algorithm 3.3) in the case of the Type2 Node with 2 slaves.

To help understanding our algorithm, we will describe in example 3.2 the main communication steps involved during the processing of a node.

Example 3.2. *Let us consider three consecutive nodes in the elimination tree (j =father(k) and i =father(j)), mapped on different processors, and focus on the processing of the central node j which is of type 2 (see Figure 3.13). Node j is added to the local pool of its master process after the reception of a message of type* `Bwd_Node` *from its father node i (step a in Figure 3.13). The master process of node j then sends information, relative to the solution sent by node i, to all of its slaves (step b). Each slave of node j then sends updates to the right-hand side to the master of node j (step c). Once all messages from the slaves have been received, the master can compute the solution (associated with the column indices of its frontal matrix). Then it sends*

message `Bwd_Node` *to the master of its son* k *(step d).*

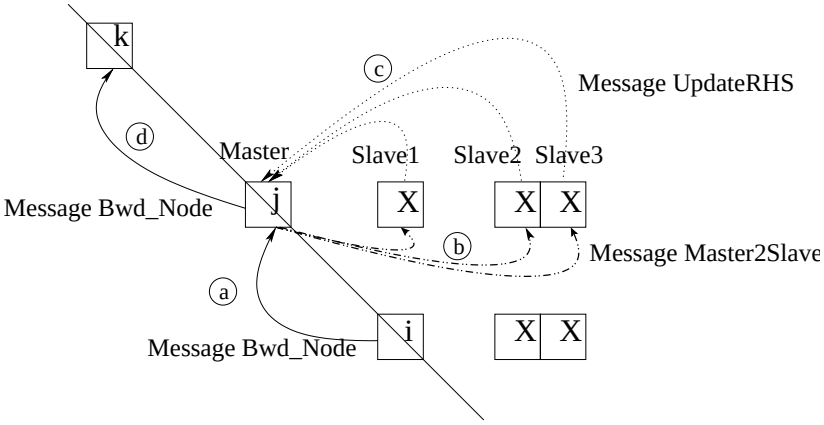


Figure 3.13: Main communication steps during the backward substitution

Chapter 4

Out-of-Core (OOC) main features

4.1 Introduction

The out-of-core behaviour of our forward and backward algorithms is very critical for large matrices when memory is limited. Our objective is to achieve good performance with respect to both run-time and memory effectively used in both sequential *and* parallel cases. The OOC run time is strongly related to the hard disk access time. The latency, the number of disk accesses, and the regularity of the reading pattern are issues that will have to be taken into consideration.

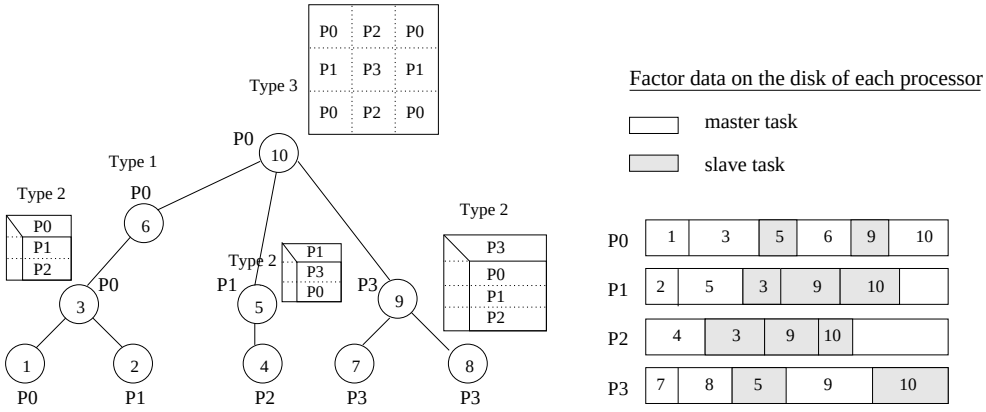
In this section, we describe the main OOC features of our algorithms.

4.2 OOC factorization phase

During the OOC execution, the computed factors are stored on the hard disk and are written in the order in which they have been computed. Results obtained by [2] show that this can be obtained with limited overhead with respect to the in-core factorization.

In a sequential environment, factors are written on the hard disk following a post-order traversal of the tree. For the parallel runs only a topological ordering, with unpredictable dynamic interleaving of slave and master tasks can be obtained. In Figure 4.1 we show such an interleaving. Figure 4.1-a) show the elimination tree is mapped on 4 processors and for each node the factor distribution is shown with respect to the type of the node. In Figure 4.1-b) the factors write sequence is shown where each block of factors is written in the order on which it is computed during the factorization phase. Post-ordering (processing the parent node just after its children) is no longer respected, and only topological ordering is applied. For example, in Figure 4.1-b) processor 0 has processed node 5 before processing node 6 which is the direct parent of nodes 3. Note also that the write sequence of factors on disk is local on each process. Type 2 tasks are distributed among several processes - one master and the other are slave with respect to the current task.

Although one could clearly take advantage of keeping part of the factors in-core at the end of the factorization, for the sake of clarity we will consider in the following that all factors data has been written to the disk at the end of the factorization phase.



a) elimination tree mapped on 4 processors *b) Hard disk of each processor*

Figure 4.1: Example of interleaving of master and slave tasks during the factorization and influence on the disk usage on each processor. We note that the sequence is not unique because of the non-deterministic nature of our asynchronous algorithm.

4.3 OOC solve phase

We use the factorization write sequence in order or in reverse order, to respectively prefetch factor blocks during the forward and the backward steps. Looking at the hard disk storage area, these two steps can be represented as directions for reading data. The forward step needs factors from the disk in a left-right direction. That is why, for the forward step, we prefetch data in the natural direction (the order in which data has been written) (see Figure 4.2). The backward step needs factors in the reverse order: right-left direction on the disk. Here, the inverse of the natural reading direction is used, so that one could expect the performance of the backward step to be slightly worse than the forward step.

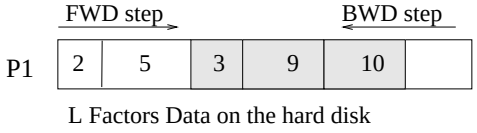


Figure 4.2: Reading direction on the disk in the solution step.

For this OOC implementation we use almost the same algorithms as for the in-core case. The only modification (see Algorithm 4.1) for the OOC execution is to load data from disk for each occurrence of the sequence '`Use factors`' in Algorithms 3.2 and 3.3.

Algorithm 4.1 : Modification of Fwd and Bwd algorithm for OOC execution

`Use_factors` of *Inode* ...	$\Longrightarrow$	**if** (OOC run) **then**
		Load data (*Inode*) from disk
		end if
		`Use_factors` of *Inode* ...

4.4 System based demand driven approach

A simple way to implement the OOC solve phase is then to use a demand driven approach. We do not use any explicit prefetching. We let the operating system handle intermediate caches when loading data.

To illustrate the potential and the limitations of a demand driven approach we report in Table 4.1 its behaviour on our test matrix QIMONDA07. We analyse the situation when the matrix fits in the main memory (parallel execution) and when the memory is critical (uni-processor execution) and also report, as a reference, the in-core solution time on 8 processors.

Nprocs	Factor Size (per proc) MB	Parallel Solve Fwd (sec)	Bwd (sec)	Factor access rate (MB/s)
In core				
8	317.5	0.9	0.9	—
OOC (Out-Of-Core)				
8	317.5	3.6	4.5	92.6
4	635.0	45.9	15.1	83.3
2	1 270.1	129.4	93.1	22.8
1	2 534.3	269.4	282.9	9.2

Table 4.1: Influence of memory used per node of the Cray XD1 on the performance of the parallel solve phase on matrix QIMONDA07. The OOC is based on a simple SYSTEM_BASED approach.

On 8 processors, we see that the extra time required in both forward and backward phases for the OOC execution corresponds to copying the factor data at a rate of 92.6 MB/s so that the copy is not all from the disk but from the system cache. Indeed the SYSTEM_BASED demand driven approach unpredictably affects the behaviour in an intrusive way. Even if the factors were written to the disk during the factorization, a significant part of them still remains in the system caches, so that the cost of accessing them during the solve phase is the cost of a main-memory access. The OOC execution allows us to decrease the number of processes used by increasing the local factor size per process. The fewer processes that are used, the fewer factors remain in the system caches and, as a consequence, the speed of access to the factors decreases. On QIMONDA07, the size of the total workspace for sequential in-core factorization (5 GB) is bigger than the available memory (4 GB). In OOC execution, a working space of size 2 GB is still needed during the factorization so that the system cannot keep all the factors in the system caches at the end of the factorization phase. Some factor blocks must then be loaded from the disk. In this case, increasing the number of disk accesses will increase the execution time. On one process, the disk access speed is really slow – 9.2 MB/s. Note that the peak speed of a memory read from the disk is 16 MB/s, so that the minimum time just to load all the factor blocks is 158 seconds.

We thus see that, when the memory is critical, the performance of the SYSTEM_BASED approach is far from the optimal. The reason is that the system I/O mechanism is in conflict with the automatic system swapping mechanisms [33, 69].

As shown in Table 4.1, the SYSTEM_BASED approach is inefficient on large matrices, when the volume of data on the disk is larger than the memory size. In this case, we observe the so called *swapping effect*: the system decides when and which data to swap to the disk. The decision is done by the system and is often based on a variant of a least recently used strategy. Note that the system has no knowledge of the data access

pattern of the algorithm. Furthermore, the fact that the system cache grows with each disk access (reading or writing data) is even more critical. It is impossible to control the actual memory used: either its size or the effective bandwidth for accessing the disk. So we do not know how much real memory is used. Moreover, the system cache management may lead to user space swaps - on our own or on other user's data, or even other system processes. Thus, if we consider that OOC is requested when the memory is limited, this unpredictable behaviour is likely to occur very often.

These drawbacks lead us to look for a new mechanism to load data from the hard disk.

Chapter 5

DIRECT_IO based method

5.1 Introduction

In this section we present a new approach based on direct access to the hard disk, that will be named DIRECT_IO. Using the DIRECT_IO access, the user has full knowledge and control of the memory used. This is a specific feature existing on many operating systems that can be specified while opening the files. Data must be aligned in memory when using DIRECT_IO mechanisms: the address and the size of the buffer must be a multiple of the memory page size. The use of this kind of I/O operation ensures that a requested I/O operation is effectively performed and that no caching is done by the operating system. Strategies can then be used to prefetch data. The inconvenience of this method is that the cache mechanism exploited by the SYSTEM_BASED approach is not available; it is thus more complex to implement and requires more algorithmic effort.

Finally for portability issues we have designed a software layer (written in C) to hide the complexity of low level direct I/O access such as the memory alignment of data. It is based on the use of a small (around 1MB) intermediate aligned buffer through which data is written (read) to (from) disk. Our code may thus be used on all operating systems.

5.2 User defined buffer

To solve large problems efficiently, which is the main target in designing an OOC solver, we propose to use small **user buffers** to explicitly control how much data is needed to prefetch from the disk.

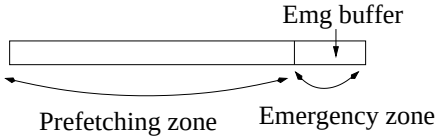


Figure 5.1: User defined buffers.

The buffer zone is divided into two areas: a prefetching zone and an emergency one - as we show in Figure 5.1. In the prefetching zone, all the space allocated to this is used to load data. We prefetch each time a large enough contiguous block in the prefetching

zone is free (1 MB in our experiments). The emergency zone is used when a block factor is not prefetched or not 'on the way' (part of a prefetch request - see Algorithm 5.1). It has to be as large as the largest factor block. In this zone we load only one factor block at a time and it is used only in so called emergency cases.

The size of the user buffers can influence the performance of the solve phase. The size of the emergency buffer $\texttt{EmgSize}(p)$ is defined as the largest block factor mapped on processor p. Let $\texttt{AvgEmgSize}$ denote the average of $\texttt{EmgSize}(p)$ over the processors. Let $\texttt{FactorSize}(p)$ be the size of the L factors per processor and let $\texttt{AvgFactorSize}$ be its average size. The prefetching buffer zone on each processor $\texttt{PrefetchBufferSize}(p)$ is then defined as

$$\texttt{PrefetchBufferSize(p)} = \max\left(\min\left(10 \times \texttt{AvgEmgSize}, \frac{\texttt{AvgFactorSize}}{4}, \ 500\ \texttt{MB}\right), \texttt{EmgSize}(p), 10\ \texttt{MB}\right) \tag{5.1}$$

The total size of buffers per processor is then

$$\texttt{Size of buffers}(p) = \texttt{PrefetchBufferSize}(p) + \texttt{EmgSize}(p) \tag{5.2}$$

In the context of our study we want to control the buffer size with respect to a fixed value (here 500 $\texttt{MB}$) and with respect to the volume of I/O per processor ($\texttt{AvgFactorSize}/4$). We thus reduce the buffer size when increasing the number of processors and limit the difference of the buffer sizes on the processors (upper bounds based on average distributions) and finally to enable some prefetching for our algorithms ($10 \times \texttt{AvgEmgSize}$). In the remainder of this thesis, equation (5.2) will be used to define the size of the buffer area for our experiments.

5.3 States of a node

The implemented algorithm reduces the disk access to the strict minimum - each item of data is loaded only once and kept in memory until it is used. To handle this, four states of the node are used to describe these transitions, (see Figure 5.2).
For every node the possible states are:
— **on disk only** - data is not available in the main memory
— **on the way** - data is not available, but it is being loaded
— **ready** - data is in the buffer and is ready to be processed
— **used** - data is in the buffer but has been already used. Corresponding space can be freed.

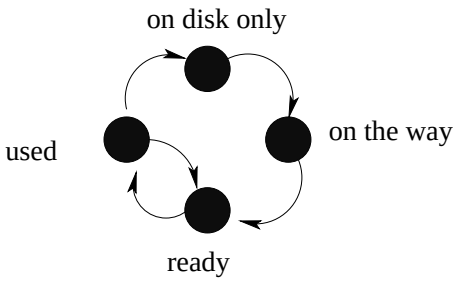


Figure 5.2: The 4 possible states of the node

The statement 'on disk only' means that the factors are not yet accessed. If we need to access data 'on disk only', we have to verify that there is enough free space in the buffer to load the data. The statement 'on the way' corresponds to data that is not yet in main memory, but we know that it is being loaded. So we may have to wait until the data is 'ready'. After the prefetching process, all loaded data in the user buffers is in the state 'ready'.

Here we use again the algorithms presented for the in-core execution (see Algorithms 3.2 and 3.3) with some additional functionalities (see Algorithm 5.1). Loading data is performed each time enough contiguous free space becomes available in the prefetching zone.

Before processing a node, we check whether it is 'ready' or 'on the way', or whether we need to load it in the emergency buffer. The verification of data availability is done each time we have '`Use_factors`' in the algorithms.

Algorithm 5.1 : OOC functionalities for the DIRECT_IO approach

```
1: if (OOC run) then
2:    if (factors of Inode are 'on disk only') then
3:       Load data from disk      (emergency loading of Inode)
4:    else if ( the factors of Inode are 'on the way') then
5:       wait until the end of the prefetch
6:    end if
7: end if
8: Use_factors to do ...
```

5.4 Comparison of SYSTEM_BASED and DIRECT_IO methods

5.4.1 Sequential case

To compare the behaviour of our out-of-core schemes with respect to an in-core execution, we report in Table 5.1 the sequential time needed for both phases of the solution step on the AUDI matrix. For this test we want an architecture with enough shared main memory. So, for this experiment only, we use an AMD Opteron based node equipped with 32 `GB` of memory and 4 high performance disks managed with a RAID0 scheme. We see that when all data (working arrays and factors) fits in main memory the SYSTEM_BASED approach is four times faster than the DIRECT_IO approach. Note also that the performance of the DIRECT_IO approach is limited by the disk bandwidth (time needed to read factors for a given phase is near to 55 `seconds`, corresponding to an access rate of 220 `MB/sec`).

Methods		Time in sec	
		Fwd	Bwd
in-core		3.8	3.8
out-of-core	SYSTEM_BASED	17.2	17.3
	DIRECT_IO	67.3	72.9

Table 5.1: Comparison of time (in seconds) needed for sequential solution step in both out-of-core and in-core for the AUDI matrix.

We then compare the performance of the SYSTEM_BASED and the DIRECT_IO approaches on the large matrix QIMONDA07 in a sequential environment and also analyse the behaviour of our algorithm when using the emergency buffer and/or the prefetch buffer. When only the emergency buffer (Emg) is used (`PrefetchBufferSize` set to zero in equation (5.2)), the total number of requests to the disks (Nb_Req Fwd and

Nb_Req Bwd) is high (equal to the number of nodes in the elimination tree) and incurs a very significant time overhead (see Table 5.2). Using a prefetch buffer of small size, our prefetching mechanism can anticipate and in this case suppress the use of the emergency buffer.

Methods	Fwd (sec)	Bwd (sec)	Nb_Req Fwd		Nb_Req Bwd	
			Prefetch	Emg zone	Prefetch	Emg zone
DIRECT_IO (Emg)	1160.6	1295.8	0	3 083 998	0	3 083 998
DIRECT_IO (Emg+Prefetch)	171.5	176.8	541	0	496	0
SYSTEM_BASED	269.4	282.9	—	—	—	—

Table 5.2: Influence of the number of buffers on the uni-processor performance on QIMONDA07. Fwd=forward phase. Bwd=backward phase. Emg zone: 1 MB; Prefetch buffer: 10 MB.

As the total size of the factors is in this case bigger than the available memory (2 GB), both the SYSTEM_BASED and the DIRECT_IO approaches *really* load factors from disk. Thus it becomes possible to compare their execution time on the solve phase. We see that the DIRECT_IO time is better for both forward and backward steps; there is, however, an even more major reason to favour this approach.

The main advantage is that the memory effectively used for buffers in the DIRECT_IO approach is 10 MB whereas the cache for the SYSTEM_BASED approach may be as large as 2.5 GB (the size of the factors). The performance of the solve is thus stabilized using the DIRECT_IO strategy, while controlling the size of the buffers.

5.4.2 Influence of parallelism on the performance

Matrix name	L factor size Avg (MB)	L factor size Max (MB)	Facto time (sec)	Procs	Methods	Workspace per proc (MB)	Fwd	Bwd
					sb	(*)	269.4	282.9
QIMONDA07	2534	2534	95.4	1	od	12	171.5	176.8
					sb	(*)	595.3	1061.2
CAS4R-L15	2416	2547	509.4	2	od	559	336.3	270.1
					sb	(*)	446.1	448.1
CONESHL	5908	5908	706.8	1	od	709	375.2	378.3
					sb	(*)	158.4	239.0
NICE20MC	1537	1689	418.2	6	od	491	148.7	225.2
					sb	(*)	298.6	573.5
AUDI	2741	2872	728.9	4	od	676	231.8	355.2
					sb	(*)	680.2	808.9
GRID3.5M	7860	7900	753.8	2	od	639	507.0	519.0
					sb	(*)	334.8	507.4
COR5HZ	2702	2970	797.7	8	od	660	397.1	476.5
					sb	(*)	512.4	1291.8
AMANDE	1404	1625	2874.5	20	od	425	725.9	964.8
					sb	(*)	596.8	1299.4
NICE9HZ	3208	3651	2030.5	20	od	893	685.9	1050.2
					sb	(*)	439.8	614.5
GRID5M	4259	4356	447.4	4	od	699	325.4	554.0

Table 5.3: Time performance of the DIRECT_IO (**od**) and the SYSTEM_BASED (**sb**) methods; Workspace holds the average working space used by the solve phase (including prefetching buffer defined in equation (5.2)). ((*) It cannot be estimated in the SYSTEM_BASED approach because of the system cache).

To illustrate the performance of the two approaches with respect to CPU time, we show in Table 5.3 the parallel behaviour of the solve phase on our complete set of test matrices.

We are interested in the case where factors are written to disk during the factorization phase because memory was limited. For the sake of clarity we thus assume that before each step (forward or backward) the system cache is flushed so that we are sure that both the SYSTEM_BASED and the DIRECT_IO approaches will have to read the L factors from disk. For each matrix, the minimum number of processors required to run the factorization phase was used (column `Procs` in Table 5.3). We show for comparison the factors size and factorization time for each matrix, as the average working space used by the solution phase. We then compare the performance of the forward and backward substitutions on the SYSTEM_BASED and the DIRECT_IO strategies. We see that, in parallel, the SYSTEM_BASED approach does not efficiently prefetch the L factors from the disk.

We note also that the backward step can be much slower then the forward step, especially on parallel runs. A possible explanation is that the backward substitution reads data from disk in a more irregular way. Since for an already processed parent node often there are more then one direct sons, choosing to process a son before another one will impact the order of needing and loading factors data on all processors. Thus irregular readings may occur. We remind, that the performance of the OOC solution phase is strongly related to the regularity of disk access.

5.5 Influence of scheduling

In the previous section, we have thus shown that the SYSTEM_BASED approach is not efficient in terms of both memory (no control of the effective memory used) and time (automatic system based prefetching is not adapted to a parallel execution). In this section, we analyse in more detail the parallel behaviour of the DIRECT_IO approach and focus on the influence of task scheduling on the performance. It is possible to use any scheduling algorithm to choose the order in which to process nodes in the pool of tasks. That is, we add nodes only at the end of the pool, but we can extract them in any order. A LIFO (Last In First Out) strategy was used in the initial Algorithm 3.1 because it is an optimal strategy for sequential execution (in terms of regularity of disk access to block factors).

5.5.1 Sequential performance

The order in which nodes are extracted from the pool can be very critical for the execution time because this will influence the order in which data is read from the disk. Indeed solving a matrix using irregular access to the hard disk could slow down the time for both forward and backward steps by a factor of more than 10 (see Table 5.4). Therefore an efficient scheduler has to be implemented to reduce the number of disk accesses and to improve the regularity of accesses.

Scheduling the order of a node's processing is possible in the pool of tasks. We add nodes only at the end of the pool, but we can extract them in any order. We show the differences between two strategies - FIFO and LIFO, in terms of disk access (Figures 5.3 and 5.4 respectively). We describe how the factor data are stored on the hard disk and how, by using the assembly tree, we add into the pool all the ready tasks at each step.

We use three data structures: the assembly tree (task dependency), the pool of tasks (only for the ready tasks) and the user buffers (to load data from the disk). The structure of the user defined buffers (prefetching and emergency zone) has been already described in Section 5.2 . In this example, we do not differentiate the states 'on the way' and 'ready'. All prefetched data are thus ready to be used. In our figures, the *arrows* point to the node to be processed. The numbers in grey with a diagonal line across represent already used data. Thus the space related to these entries in the prefetching zone can be used for further load of data. Each time we have to process a node that is not in memory, we load it to the emergency (Emg) buffer. In this example, prefetching is performed each time half of the prefetching zone is free (because the associated node factors are in the state 'used'). The second zone (Emg) is used only if the data needed is not already prefetched in the user buffer.

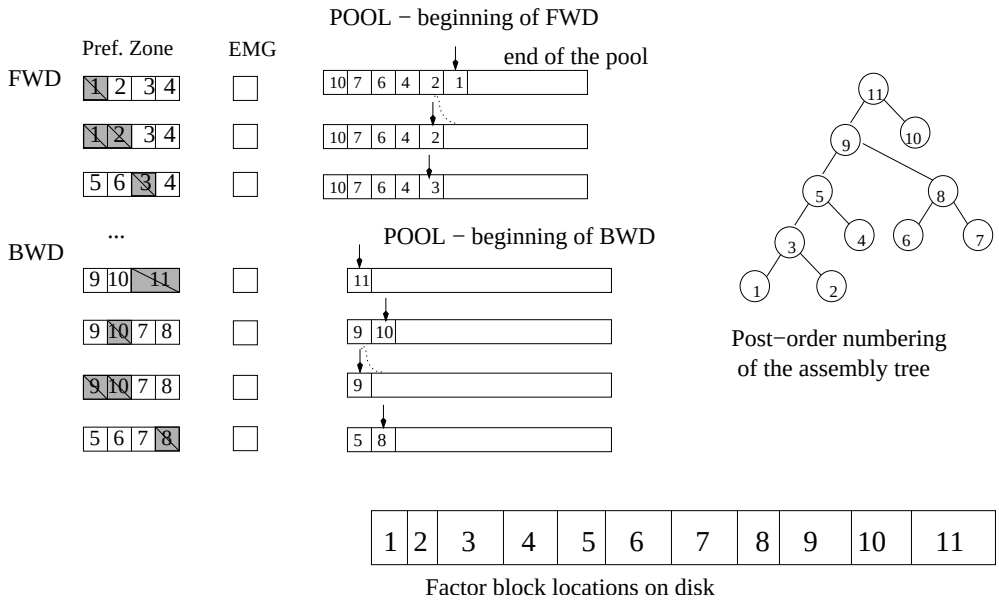


Figure 5.3: Algorithm with a LIFO processing of the tree in sequential mode

In Figure 5.3, we present the optimal (for sequential execution) LIFO (Last In First Out) strategy for extracting a node from the pool. Here we have no calls to the emergency zone during both forward and backward steps.

In Figure 5.4 we use a FIFO (First In First Out) strategy to extract the nodes from the pool. Starting with the forward step, the leaf nodes are added into the local pool so that the post-ordering is respected (from the end to the beginning of the pool). The prefetching zone loads data in the forward direction from the disk. Nodes 10, 7, and 6 are loaded through the emergency zone. Loading data in the Emg zone often leads to an irregular access (of relatively small size) to the data on the hard disk. This will influence the execution time of the whole phase.

For the backward step, the prefetching zone has been loaded data in the backward direction from the disk. Firstly, the root node is extracted from the pool and processed. Nodes 7 and 8 are, in our case, prefetched in place of the root factor block. This time we have less emergency calls and more regular access to the disk. Similar effects are observed on real matrices, which explains the relatively better behaviour of the backward step with the FIFO strategy (see Table 5.4).

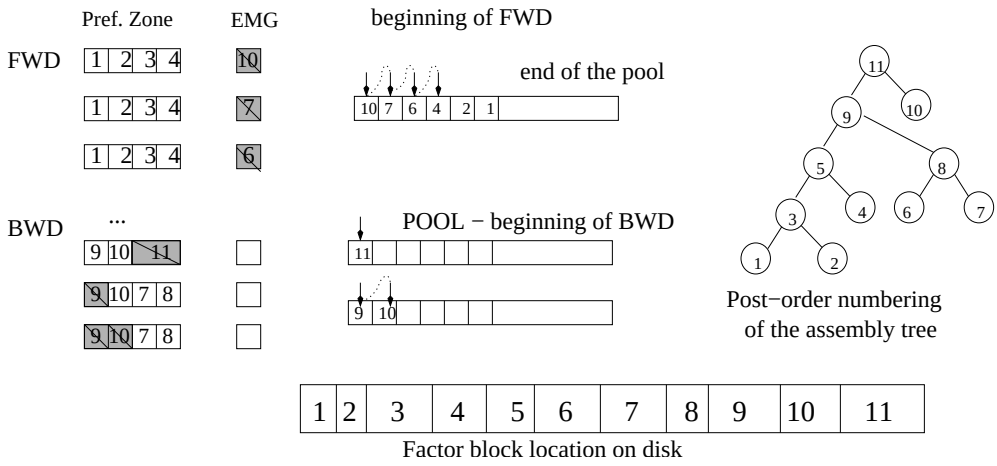


Figure 5.4: Algorithm with a FIFO processing of the tree in sequential mode

We present the results of the two strategies in Table 5.4. We compare the time for the forward and for the backward step with the minimum time needed if all factor data could be loaded at once (T_min). We compare, also, the number of requests per step for the prefetching zone and the emergency one (Nb_Req). Note that the FIFO strategy, which does not respect the node order on the disk, is significantly slower than the LIFO strategy. Furthermore, as expected, the forward step is even slower than the backward step in this case.

Strategy	T_min (sec)	T_Fwd (sec)	T_Bwd (sec)	Nb_Req Prefetch Fwd	Nb_Req Emg zone Fwd	Nb_Req Prefetch Bwd	Nb_Req Emg zone Bwd
LIFO	158.4	171.5	176.8	541	0	496	0
FIFO	158.4	2 360.9	1 480.1	338	3 054 580	30 053	2 877 695

Table 5.4: Influence of the scheduling of the tasks on QIMONDA07 in sequential. Emg buffer = 1 MB; Prefetching buffer = 10 MB

Running sequentially, the FIFO based extraction shows how critical the scheduling can be on the performance. LIFO scheduling is optimal for the sequential case and guarantees contiguous reading of factors from disk.

In parallel, we cannot guarantee a post-ordering of the nodes. Thus two leaf nodes can be processed in parallel at the same time. But the parent node waits for all of its children before being processed, respecting the topology of the tree (topological ordering). A contiguous access to the factors may not be respected and thus similar effects to those presented here can be expected.

5.5.2 Parallel performance with LIFO scheduler

Here we analyse in more detail the parallel behaviour of the DIRECT_IO approach. We first compare in Table 5.5 the time for the forward and backward steps with the minimum time (T_min) to load factors from the disk on the QIMONDA07 matrix. Note that T_min depends on the maximum bandwidth (16MB/s) and the factor size on the most loaded processor (column `Factor Size per proc` of Table 5.5). We also report in the 4 last columns the number and the type of buffer requests per step. On one processor, a LIFO order to extract tasks from the pool leads to a contiguous access to the hard disk. In parallel, we cannot guarantee that the order of processing of the tasks (and the factor blocks) will correspond to the order used to write them to the disks. We see in Table 5.5

that work needs to be done on the scheduling to reduce the gap between the minimum time to load factors and the actual time, particularly for the backward substitution.

Strategy	Nb of Procs	Factor Size per proc$^{(*)}$ (MB)	T_min (sec)	Fwd (sec)	Bwd (sec)	Max Nb Requests per step			
						Fwd $^{(*)}$		Bwd $^{(*)}$	
						Prefetch	Emg zone	Prefetch	Emg zone
LIFO	1	2 534	158.4	171.5	176.8	541	0	496	0
LIFO	2	1 270	79.9	89.6	88.7	274	0	250	0
LIFO	3	846	57.9	64.9	262.1	190	3	169	422 497
LIFO	4	635	41.3	47.2	91.6	138	0	127	0
LIFO	6	423	31.5	38.0	186.7	102	6	86	422 498
LIFO	8	317	21.8	24.9	137.6	70	0	64	321 871
LIFO	16	159	11.9	13.2	94.4	39	2	32	214 245
LIFO	24	105	9.0	10.9	48.5	42	5	38	119 792
LIFO	32	79	8.2	9.1	53.1	25	1	30	116 209

Table 5.5: Influence of the parallelism on QIMONDA07 using LIFO strategy. Emg=emergency buffer:1 MB; Prefetch buffer:10MB per processor; $^{(*)}$: Max per processor.

In fact, this gap is correlated with the large number of emergency calls during the backward step. Note that, in this example, we have many fewer emergency requests during the forward step than during the backward step. One reason is that the QIMONDA07 matrix has many nodes of relatively small size, so we have a relatively small number of Type 2 tasks that could require the use of the Emg buffer. Another reason, illustrated in the following discussion, is that one can expect the backward step to be more sensitive to scheduling than the forward step. Indeed, at the beginning of the backward step, we have in general a small number of root nodes, mapped onto few processors. The other processors have no work and are waiting. During the backward step, the end of one task results in the activation of multiple other tasks on other processors. Furthermore, if we choose to process a node *Inode*, a LIFO strategy will induce the processing of all of its children before the brother of *Inode*. If the factors of this node, *Inode*, are not in memory then the factors of the children of *Inode* will not be in memory either. This will lead to emergency requests.

5.5.3 Illustration of the high number of emergency calls with LIFO scheduler

We illustrate the limitations of the LIFO scheduler on the small example described in Figure 5.5. For the given assembly tree mapped onto two processors (P1 and P2), we show at the beginning of the backward step, the data in the prefetching zone and the pool of tasks. To simplify the illustration of our algorithm, we assume that the root is mapped on both processors and that all other nodes are mapped on only one processor (Type 1 nodes). We will comment on the effect of Type 2 nodes in our algorithm later. Some data are pre-loaded in the prefetching zone on both processors, respecting the backward step direction of needed data. With a LIFO scheduling, after processing the root node, P1 continues with the only node in its POOL (node 3). This node is not 'in memory' and requires an emergency access. Furthermore, if node 1 is added to the pool after the end of node 6 on processor P2 (that would add nodes 4 and 5 to the pool of processor P1), then accessing the factors of node 1 will lead to another emergency call.

On the other hand, during the forward phase, where we exploit the large task independence of the leaves, all processors often have at least one node to process. In this case, all processors start working at almost the same moment. As the work is distributed

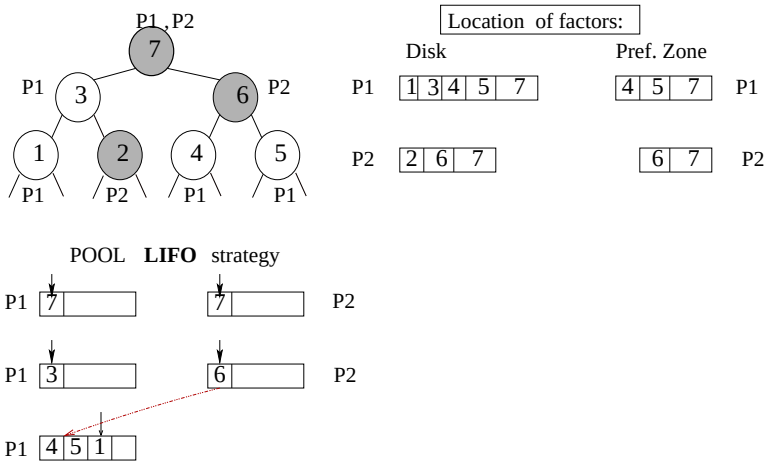


Figure 5.5: Limitations of LIFO scheduling in parallel execution during the backward step.

regularly among the processors, they will progress in a synchronous way. The algorithm will more naturally process the complete tree respecting the post-ordering of the nodes in the tree.

For all these reasons and since we have seen in Table 5.5 that the performance of the backward phase is critical even on a limited number of processors, we describe in the following chapter a modification of the scheduler.

Chapter 6

Scheduling to improve performance

6.1 NNS scheduler

6.1.1 Description of the algorithm

One way to limit the number of disk accesses is to follow strictly the write sequence of the factorization step. By doing so we will always get the node at the top of the memory. We hope that this new algorithm will free more contiguous space in the prefetch buffer, so that less emergency calls will be needed. We define the **Next Node in the Sequence (NNS)** to be the next node to be processed with respect to the write sequence on the disk (dynamically decided during factorization). During the forward step it will be the next non-processed master-node whereas during the backward step it will correspond to the previous non-processed master-node.

We will firstly focus on the backward phase.

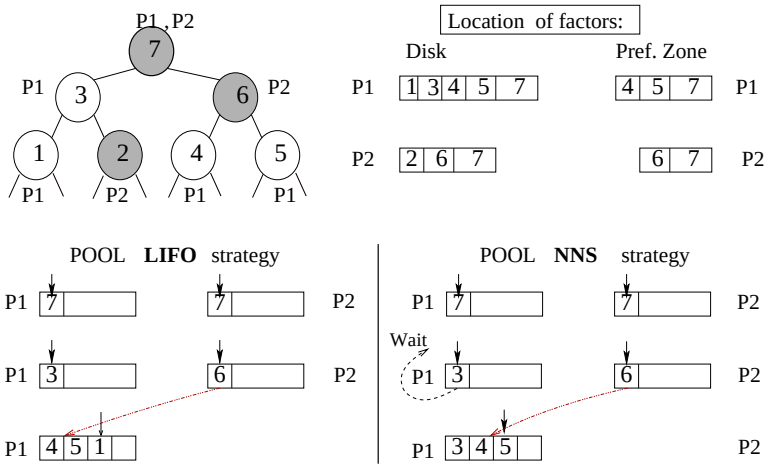


Figure 6.1: Comparison of LIFO and NNS extraction from the pool.

This so called NNS strategy is illustrated in Figure 6.1 showing a comparison with the LIFO scheduling. One can see that with a LIFO strategy, node 3 was added to the pool for P1 at the end of the process of the root node 7 mapped on both processes. Node 3 was then treated by P1 before nodes 4 and 5. On the other hand, with the NNS strategy, node

3 is not processed. P1 waits for node 5 to be added to the pool since it is the next node in the sequence after node 7.

Note that slave tasks are not considered in this sequence. The slave tasks, for Type 2 nodes, are processed on the fly (do not use the pool) and are driven by the order in which the messages are received. Our new algorithm (fully described in Algorithm 6.1) thus consists in respecting the sequence order to process nodes on each processor.

In the following, we first describe the new NNS strategy to extract work from the pool and we prove that we can safely wait for the NNS node.

In our NNS algorithm (Algorithm 6.1), a new 'blocking receive' (at line β) has been introduced with respect to Algorithm 3.1. The main difference between the blocking receive from the original algorithm (at line α of Algorithm 6.1) and the one introduced at line β is that, at line β, our blocking receive is performed while we have tasks ready to be activated in the pool. Since this is done separately on each processor (local pool) we will have to prove that it does not introduce a deadlock between processes.

Changes made to our scheduling Algorithm 3.1 are written with larger font in Algorithm 6.1. All unchanged parts are written in tiny characters.

Algorithm 6.1 : Scheduling POOL with next node in the sequence (NNS) strategy

```
 1: Step = Fwd or Bwd
 2: if (Fwd) then
 3:    Initialise POOL with the leaf nodes mapped on Myid
 4:    Initialise NNS pointer to the first leaf node
 5: else
 6:    Initialise POOL with root nodes mapped on Myid
 7:    Initialise NNS pointer to the first root node
 8: end if
 9: while (Not finished) do
10:    if (POOL is not empty) then
11:       if a message is available Process_Message(message)        [See Algorithms 3.2 and 3.3]
12:    else
13:       [α] Wait for a message and then Process_Message(message)      [See Algorithms 3.2 and 3.3]
14:    end if
15:    if (POOL is not empty and Process_Message not called) then
16:       if (NNS in POOL) then
17:          Inode = NNS ; Update NNS
18:          if ( Fwd ) Fwd_Process_node(Inode)   [See Algorithm 3.2]
19:          if ( Bwd ) Bwd_Process_node(Inode)   [See Algorithm 3.3]
20:       else
21:          [β] Wait for a message and then Process_Message(message)
22:       end if
23:    end if
24: end while
```

To prove the correctness of our new algorithm, we will formulate and demonstrate two more properties, based on the assembly tree and the task dependency.

Property 6.1. *Forcing the sequence to schedule nodes as in Algorithm 6.1 does not introduce deadlock.*

Proof: First of all, as explained before, Type 2 slave tasks do not go through the pool of tasks and are processed 'on the fly' (at the reception of a message MASTER2SLAVE for both forward and backward steps). Therefore, our blocking receive will not prevent us from treating such slaves tasks. Type 3 tasks are only concerned with the largest root node of which only the master task will go through the pool. In our proof, we can thus

focus on the master tasks (of any type) since they are the only ones that might be blocked in the local pool.

Let us focus on the backward case. (The proof for the forward case is similar and can be easily deduced from the backward case.)

Let $NBps$ be the number of processes and let us suppose that we have a deadlock between r processes ($r \leq NBps$). On each process P_i ($i \in [0 \,..\, r-1]$), let N_{P_i} be the next node not processed in the sequence of processes P_i .

We first mention/prove a simple intermediate property between nodes ready to be activated in the local pools.

Property 6.2. *During the backward step, if node j is ready on process P_i , then j is not an ancestor of N_{P_i} .*

Proof: Thanks to the main elimination property, if j were an ancestor of N_{P_i} then it would be in the sequence of the backward step before N_{P_i} . This contradicts the definition of N_{P_i} . $\diamond$

Proof of property 6.1 (continued)
Let N_{P_i} $i \in [0 \,..\, r-1]$ be the nodes in the sequence that processes P_i are waiting for. If N_{P_0} is not ready (not in the pool), then it means that one of its ancestors (j_1) has not been processed. Because of Property 6.2, j_1 cannot be ready in the pool of P_0 . Let us suppose, without loss of generality, that j_1 is in the pool of process P_1 . Furthermore, on process P_1 , N_{P_1} is not in the local pool. (Note that N_{P_1} might be equal to j_1). Therefore there exists an ancestor j_2 of N_{P_1} , ready to be activated on another process P_2 . Either N_{P_2} is equal to N_{P_0} and we have a cycle of dependencies between processes, or we can continue and will end up with a cycle between r processes.

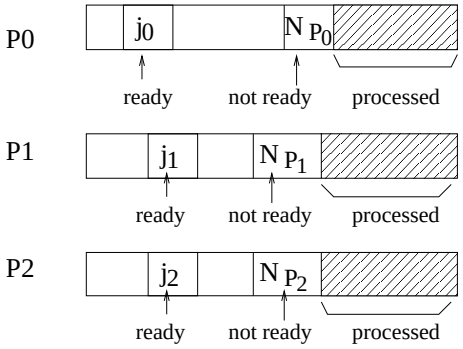


Figure 6.2: Local factor sequence on each process. N_{P_i} is the next node in the sequence, not processed and not ready in the local pool or process P_i ; j_{i+1} is a ready node, an ancestor of N_{P_i}

Let us suppose that we have reached a cycle of size r' , $r \geq r' \geq 2$. Let

$$(N_{P_0}, j_1), (N_{P_1}, j_2), (N_{P_2}, j_3), ...(N_{P_{r'}}, j_0)$$

be such a cycle, where j_0 is ready on process P_0 and is an ancestor of $N_{P_{r'}}$. In each couple (N_{P_i}, j_{i+1}) j_{i+1} is an ancestor of N_{P_i} and is thus processed strictly before N_{P_i} in the backward sequence. Furthermore, by the definition of N_{P_i} , N_{P_i} is in the sequence before any node in the local pool of P_i . Let $\rightarrow$ denote the precedence in the backward sequence. $x \rightarrow y$ mean that x is before y in the backward sequence. $\stackrel{a}{\rightarrow}$ indicates an

ancestor relation, $x \xrightarrow{a} y$ indicates that x is before y because x is an ancestor of y. (Note that $x \xrightarrow{a} y$ implies $x \rightarrow y$ and $x \neq y$). We thus have :

$$j_0 \xrightarrow{a} N_{P_{r'}} \rightarrow j_{r'} \xrightarrow{a} N_{P_{r'-1}} \ \ldots \ j_2 \xrightarrow{a} N_{P_1} \rightarrow j_1 \xrightarrow{a} N_{P_0} \ ,$$

which means that N_{P_0} is not the first ready node *in the sequence of process* P_0, since j_0 is ready and is before N_{P_0} in the sequence. Thus j_0 is equal to N_{P_0}. Furthermore, thanks to our cycle, j_0 is before j_1 in the sequence ($j_1 \neq j_0$), which contradicts the fact that j_1 is an ancestor of $N_{P_0}(= j_0)$ located on process P_1. We have thus proved that our algorithm does not introduce any deadlock. $\diamond$

Normally, the next node in the sequence is located at the end of the prefetch buffer, and processing this node will free more contiguous space in the buffer. We hope that this will lead to more regular disk access and will improve the performance especially for the backward step in a parallel environment.

Strategy	Nb of Procs	Fwd (sec)	Bwd (sec)
LIFO	1	171.5	176.8
LIFO	2	89.6	88.7
LIFO	3	64.9	262.1
LIFO	4	47.2	91.6
LIFO	6	38.0	186.7
LIFO	8	24.9	137.6
LIFO	16	13.2	94.4
LIFO	24	10.9	48.5
LIFO	32	9.1	53.1

Strategy	Nb of Procs	T_min (sec)	Bwd (sec)	Nb_Req$^{(*)}$ in Bwd step	
				Prefetch	Emg
NNS	1	158.4	177.2	496	0
NNS	2	79.9	93.7	250	0
NNS	3	57.9	65.5	174	1
NNS	4	41.3	50.5	117	0
NNS	6	31.5	37.9	93	0
NNS	8	21.8	45.2	57	0
NNS	16	11.9	13.8	36	0
NNS	24	9.0	13.2	38	0
NNS	32	8.2	10.7	34	0

Table 6.1: Influence of the NNS scheduling on QIMONDA07. Emg=emergency buffer:1 MB; Prefetch buffer:10MB per processor; $^{(*)}$: Max per processor.

The results, presented in Table 6.1 show that using the NNS strategy on the QIMONDA07 matrix significantly improves the performance in the backward step on parallel runs. The time for the backward substitution has a more realistic behaviour and is reduced by a factor of 5 (compare Tables 5.5 and 6.1 on 6 processors: LIFO strategy – 186.7 sec and NNS strategy – 37.9 sec). As shown, the NNS strategy is much closer to the minimum time for loading factors from disk. Indeed, we obtain a performance only 20% more than the minimum. The only exception is with 8 processors, when the performance/ the run-time is twice as slow as the T_min. This unusual behaviour of the performance with 8 processors shows that the NNS strategy is not the optimal way to schedule tasks in the pool. Until now, we have only focused on reducing the emergency calls. As the NNS strategy uses the factorisation write sequence, it is related also to the elimination tree structure computed during analysis and the partial mapping of the tasks onto the processors as well as the tree traversal resulting from the dynamic decision taken during the factorization phase. In some cases, as with 8 processors, choosing to follow the write sequence obtained during factorization could produce the undesirable effect of stalling a particular node by a blocking receive, while other nodes, in the pool *and in the user buffer*, are ready to be processed.

In our new algorithm the slave tasks of type 2 nodes might still involve requests to the Emg buffer and/or to be prefetched out of sequence. (This is the case for the backward step, when 3 processors are used, in our case.)

QIMONDA07 has a large number of relatively small nodes, with a relatively small number of Type 2 nodes. This explains why our NNS algorithm usually has no emergency calls in both steps of the solve phase. The influence of the slave tasks on the performance can thus be expected on large 3D matrices for which a large number of Type 2 nodes is requested.

6.1.2 Experiments with LIFO and NNS strategies

In this section the NNS and LIFO schedulings are compared on all our test matrices. One main difference with respect to the QIMONDA07 matrix used in the detailed analysis of the previous sections is that for the other matrices the factor block is on average much larger and thus results in a large number of type 2 nodes.

The parallel behaviour of each matrix is reported on the minimum number of processors required to run the out-of-core factorization phase with one MPI process per node of the CRAY XD1. For each matrix and each run with the same number of processes, the same physical processors are used with LIFO and NNS to guarantee similar experimental conditions. The workspace size for the solve phase is divided between two buffers – $Prefetch$ and Emg (see Figure 6.2). The average (Avg) and the maximum factor size (Max) are included in our tables firstly to show that the factors are well equilibrated among the processors and secondly to compare the maximum factor size with the effective maximum workspace used during the solve phase. Indeed, one main property of the DIRECT_IO strategy is that we explicitly control the size of the working space used. Increasing the workspace would help our algorithm so that it is critical to show that our runs are performed in a limited-memory environment. For each test we report the performance (time and number of accesses to the buffers) obtained during forward (Fwd) and backward (Bwd) substitutions.

We first comment on the effect of equation (5.1) on the size of the prefetch zone. On CONESHL with 1 processor, 709 MB of working space are used for 5.9 GB of factor data (209 MB for the emergency buffer and 500 MB for the prefetching zone). For larger number of processors, the increase in the number of Type 2 nodes leads to a decrease in the size of the factor blocks which results in a decrease in the size of the buffers. In Table 6.9, however, we see that with the matrix NICE9HZ when the size of the emergency buffer remains relatively large with respect to the maximum factor size then equation (5.1) limits the size of the prefetch zone to 500 MB which is only 1.27 times the size of the emergency buffer. This will limit the capacity of the algorithm to perform prefetching.

Furthermore, for a given matrix, the decrease in the size of the factors often leads to a decrease in the time for both the forward and the backward steps. As observed in the previous section, one can see a correlation between the performance and the number of accesses to the emergency buffer. However, although an access to the emergency buffer will always block the process during the time to load the corresponding block factor from the disk, its effect on the node tasks mapped on other processes will depend on the mapping of the tree to the processes. Therefore one should not expect that the smallest number of emergency calls will result in the best performance (see, for example, Table 6.3 on 8 processors with strategy NNS : 14 Emg calls and 64.3 sec during forward compared to 2 and 67.0 sec during backward). It is clear, however, that the backward step is more sensitive to the accumulation of those time delays even if with the NNS strategy this is significantly reduced with respect to the LIFO strategy. On all matrices, we see that the

NNS strategy is better both during forward and backward steps in limiting such effects by forcing an order compatible with the order used to write the factor blocks during the factorization. For both phases, the NNS strategy also ensures more regular disk access and significantly improves the execution time for all our test matrices.

Strategy	Nb of Procs	Facto Time (sec)	Factor Size per proc (MB)		Workspace per proc (MB)		Solve Time		Max Nb Requests per step			
							Fwd	Bwd	Fwd		Bwd	
			Avg	Max	Prefetch	Emg	(sec)	(sec)	Prefetch	Emg	Prefetch	Emg
LIFO	2	509.4	2416	2547	500	59	336.3	270.1	11	0	11	3279
NNS							334.0	269.7	11	0	10	0
LIFO	4	264.6	1200	1291	300	34	221.0	356.3	11	10	10	133594
NNS							220.0	190.6	12	1	14	1
LIFO	8	158.4	596	756	149	34	165.5	203.3	20	68	10	74582
NNS							117.7	99.9	14	8	10	1
LIFO	16	99.5	295	336	74	10	102.7	156.0	25	129	10	37861
NNS							63.9	84.1	21	28	10	4
LIFO	32	76.2	146	170	36	6	47.0	102.3	16	74	13	37055
NNS							44.5	69.8	15	10	10	2

Table 6.2: Parallelism on CAS4R-L15

Strategy	Nb of Procs	Facto Time (sec)	Factor Size per proc (MB)		Workspace per proc (MB)		Solve Time		Max Nb Requests per step			
							Fwd	Bwd	Fwd		Bwd	
			Avg	Max	Prefetch	Emg	(sec)	(sec)	Prefetch	Emg	Prefetch	Emg
LIFO	1	706.8	5908	5908	500	209	375.2	378.3	27	3	26	5
NNS							374.7	378.3	27	3	26	5
LIFO	4	220.8	1465	1481	366	77	102.6	139.0	9	6	8	2
NNS							102.4	133.9	9	6	8	1
LIFO	8	134.3	726	987	181	52	63.9	95.7	15	14	13	12
NNS							64.3	67.0	13	14	12	2
LIFO	16	80.3	360	393	90	12	36.3	64.6	12	17	9	6488
NNS							33.6	48.2	10	10	9	2
LIFO	32	76.9	179	221	44	7	24.8	40.2	19	60	11	4040
NNS							22.9	33.6	20	89	20	21

Table 6.3: Parallelism on CONESHL

Strategy	Nb of Procs	Facto Time (sec)	Factor Size per proc (MB)		Workspace per proc (MB)		Solve Time		Max Nb Requests per step			
							Fwd	Bwd	Fwd		Bwd	
			Avg	Max	Prefetch	Emg	(sec)	(sec)	Prefetch	Emg	Prefetch	Emg
LIFO	6	418.2	1537	1689	384	107	148.7	225.2	10	6	8	5602
NNS							134.5	158.6	10	0	9	0
LIFO	8	351.4	1147	1232	286	90	126.9	153.2	10	20	11	4570
NNS							120.7	135.3	14	9	9	2
LIFO	16	236.3	564	774	141	26	116.9	116.7	24	205	13	5042
NNS							92.7	80.1	18	42	13	27
LIFO	32	162.9	276	399	69	21	67.2	76.2	37	214	22	2578
NNS							57.5	57.1	40	114	21	607

Table 6.4: Parallelism on NICE20MC

Strategy	Nb of Procs	Facto Time (sec)	Factor Size per proc (MB)		Workspace per proc (MB)		Solve Time		Max Nb Requests per step			
							Fwd	Bwd	Fwd		Bwd	
			Avg	Max	Prefetch	Emg	(sec)	(sec)	Prefetch	Emg	Prefetch	Emg
LIFO	4	728.9	2741	2872	500	176	231.8	355.2	3	1	4	7179
NNS							218.3	233.5	14	0	12	1
LIFO	8	407.1	1354	1480	338	216	152.5	215.5	15	45	13	12523
NNS							147.8	166.2	11	23	10	1
LIFO	16	306.4	664	955	166	81	144.8	159.0	29	65	16	7314
NNS							118.2	121.0	22	52	20	452
LIFO	32	202.7	325	573	81	20	73.7	101.4	29	86	27	4315
NNS							73.3	80.4	42	151	36	63

Table 6.5: Parallelism on AUDI

Strategy	Nb of Procs	Facto Time (sec)	Factor Size per proc (MB)		Workspace per proc (MB)		Solve Time		Max Nb Requests per step			
							Fwd	Bwd	Fwd		Bwd	
			Avg	Max	Prefetch	Emg	(sec)	(sec)	Prefetch	Emg	Prefetch	Emg
LIFO	2	753.8	7860	7900	500	139	507.0	519.0	37	0	34	0
NNS							506.0	513.6	37	0	34	0
LIFO	4	403.7	3919	3951	500	139	273.6	383.3	20	0	17	191695
NNS							273.2	293.0	20	0	17	0
LIFO	8	209.0	1948	1994	487	139	174.1	289.9	14	24	9	96156
NNS							144.9	184.9	9	1	8	0
LIFO	16	131.5	963	1041	240	139	104.5	207.0	20	27	9	48039
NNS							87.3	125.6	21	22	9	1
LIFO	32	131.0	472	593	118	39	95.6	149.4	39	225	23	60630
NNS							74.2	83.3	53	100	38	39

Table 6.6: Parallelism on GRID3.5M

Strategy	Nb of Procs	Facto Time (sec)	Factor Size per proc (MB)		Workspace per proc (MB)		Solve Time		Max Nb Requests per step			
							Fwd	Bwd	Fwd		Bwd	
			Avg	Max	Prefetch	Emg	(sec)	(sec)	Prefetch	Emg	Prefetch	Emg
LIFO	8	797.7	2702	2970	500	160	397.1	476.5	20	16	17	26981
NNS							298.6	351.3	18	12	18	10
LIFO	12	589.6	1793	2154	448	160	249.1	447.8	18	34	15	23368
NNS							230.1	325.3	16	11	13	1
LIFO	16	503.7	1340	1584	335	160	261.7	353.7	21	52	14	14278
NNS							220.1	303.8	19	28	23	3675
LIFO	32	329.6	660	820	165	45	189.9	310.2	30	142	39	9090
NNS							185.9	218.1	22	47	19	10

Table 6.7: Parallelism on COR5HZ

Strategy	Nb of Procs	Facto Time (sec)	Factor Size per proc (MB)		Workspace per proc (MB)		Solve Time		Max Nb Requests per step			
							Fwd	Bwd	Fwd		Bwd	
			Avg	Max	Prefetch	Emg	(sec)	(sec)	Prefetch	Emg	Prefetch	Emg
LIFO	20	2874.5	1404	1625	351	74	725.9	964.8	31	114	23	40323
NNS							678.0	866.1	20	70	14	4
LIFO	24	2132.0	1171	1364	292	74	679.8	1071.6	25	156	27	37950
NNS							475.5	629.5	19	37	16	8
LIFO	32	1677.1	872	1028	218	43	358.9	814.6	19	37	28	28334
NNS							350.9	564.6	15	42	10	6

Table 6.8: Parallelism on AMANDE

Strategy	Nb of Procs	Facto Time (sec)	Factor Size per proc (MB)		Workspace per proc (MB)		Solve Time		Max Nb Requests per step			
							Fwd	Bwd	Fwd		Bwd	
			Avg	Max	Prefetch	Emg	(sec)	(sec)	Prefetch	Emg	Prefetch	Emg
LIFO	20	2030.5	3208	3651	500	393	685.9	1050.2	38	66	30	30735
NNS							651.8	696.7	45	78	28	35
LIFO	24	1724.3	2661	3048	500	228	642.9	844.7	39	86	30	29098
NNS							571.6	684.8	32	25	32	8765
LIFO	32	1517.7	1989	2454	497	228	559.8	734.0	52	59	51	21501
NNS							471.9	604.7	45	58	58	9293

Table 6.9: Parallelism on NICE9HZ

Strategy	Nb of Procs	Facto Time (sec)	Factor Size per proc (MB)		Workspace per proc (MB)		Solve Time		Max Nb Requests per step			
							Fwd	Bwd	Fwd		Bwd	
			Avg	Max	Prefetch	Emg	(sec)	(sec)	Prefetch	Emg	Prefetch	Emg
LIFO	4	447.4	4259	4356	500	199	325.4	554.0	19	1	19	413902
NNS							347.9	321.5	19	15	19	1
LIFO	8	277.6	2120	2583	500	136	247.8	368.1	15	56	10	138017
NNS							186.7	223.6	13	3	11	4
LIFO	16	165.8	1048	1113	262	66	122.7	236.0	33	116	9	71173
NNS							84.5	133.4	9	0	10	2
LIFO	32	106.4	519	567	129	30	62.6	171.6	20	221	9	34909
NNS							45.2	77.1	20	15	9	1

Table 6.10: Parallelism on GRID5M

6.2 BPN scheduler

6.2.1 Description of the algorithm

Our NNS strategy, strictly follows the write sequence on disk. Thus sometimes we chose to wait for a specific node (NNS node) even if there are other nodes in the pool of tasks, ready to be processed. Our motivation was to have the read sequence of factors follow the write sequence resulting from the factorization. Thus with the NNS strategy even if some of the ready nodes in the pool of tasks have already been prefetched in memory, we will still wait for our NNS-node. Our objective in this section is to relax the NNS strategy to allow 'out-of-order' processing of the loaded factors.

To illustrate an undesirable effect of waiting when using the NNS strategy, we will take a small example of an elimination tree and an ordering of the tasks. Note that this example, although simplified for the purpose of our discussion, results from our analysis of matrix Qimonda07 on 8 processors.

We present an elimination tree where every node is associated with some level of the tree ($L0, L1, L2, ...$). As we said in Section [2.1] the type of parallelism depends on the size of the node but also on its level in the elimination tree. We recall that for level $L0$ each subtree is mapped onto a single processor. Above level $L0$, type2 nodes are authorized. The tree of Figure 6.3 is mapped on two processors ($P0$ and $P1$) and has the characteristic of a long chain of nodes, mapped onto the same processor - $P0$. Nodes 13 and 11 have a child at level $L0$ of the tree. The static mapping of all nodes is performed during the analysis phase that tries to equilibrate the work among the processors in terms of factors to compute and messages to send. In our example, the whole tree is divided into two branches that are relatively equal in terms of the volume of the factor.

We first briefly describe why the proposed order (see Figure 6.3) for traversing the

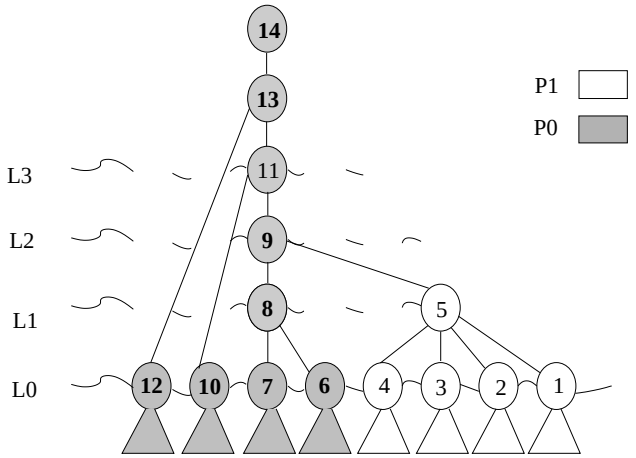


Figure 6.3: Illustration of a tree mapped onto two processors during the analysis phase. Node numbers correspond to the order in which the factor nodes have been written onto disk during the factorization phase. Nodes on level $L0$ and all their subtrees are processed on a single processor.

tree in parallel during the factorization phase makes sense with respect to memory usage during factorization.

During the factorization step we use three storage areas - one for the factors, one to stack the contribution blocks, and another for the current frontal matrix[3]. During the tree traversal, the memory required by the stack (containing contribution blocks) varies depending of the order of the operations. When the partial factorization of the frontal matrix is performed, a contribution block (CB) is stacked which increases the size of the working memory. When the frontal matrix is formed and assembled, the contribution blocks of the child nodes are discarded and the size of the stack decreases.

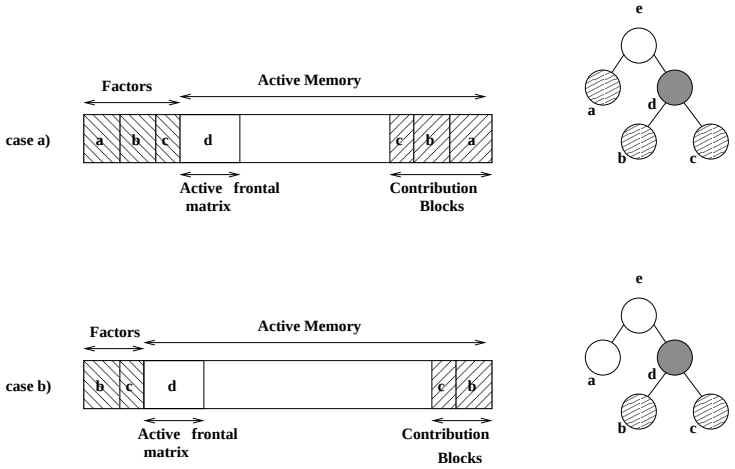


Figure 6.4: Stack memory management schemes. Node d is being assembled. Case a) nodes a, b, c have been already processed. Case b) only nodes b, c have been already processed.

In Figure 6.4, two possible situations are presented. In case a) all leaf nodes a, b, c are processed before the parent node d, which increases the stack memory for the CB blocks. In case b) we reduce the peak of the stack memory by following a post-order traversal of the tree. The decision to process a node before another one, is related to the objective of minimizing the stack memory [3, 53, 68]. The elimination algorithm follows a post-order

traversal of the assembly tree, which ensures minimum stack memory use which is critical for the OOC case [70, 71, 72, 73, 74].

In Figure 6.3, the node numbering corresponds to the final factorization sequence, (with the assumption that nodes 1 and 6 are processed in parallel by processors $P1$ and $P0$ respectively). Note that for $P0$ other topological orderings that would require more intermediate working memory are possible (for example with the order $6, 7, 10, 12, 8, 9, 11, 13, 14$ we have to keep on the stack the contributions produced by 10 and 12 while processing node 8).

Waiting for the NNS node far too long while other nodes are available in the user buffer can also happen in the backward step. During the backward step, the tree is traversed in reverse order with respect to the local numbering (indicated in Figure 6.5) on each processor. Following the NNS strategy P0 will first process root node 14, then node 13 and 12 processing the whole sub-tree on node 12. Then P0 will process node 11 and 10 again with its whole sub-tree. During this period P1 will wait for its first NNS node 5 which will be freed after processing node 9. For this particular example, local scheduling decisions cannot improve the performance of the backward step (reduce the waiting time on processor P_1). Since our strategy of extracting nodes is local, idle processors do not communicate with the other processors to influence the scheduling decision and to reduce their waiting time. This is a limitation of our approach.

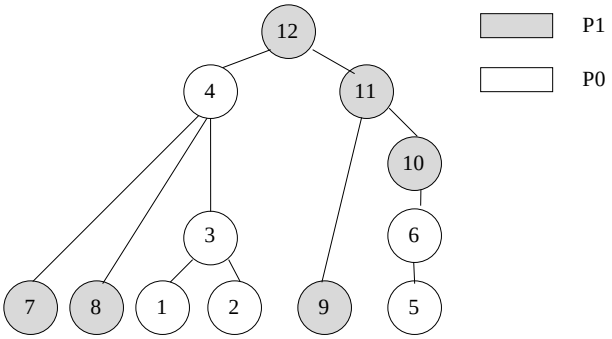


Figure 6.5: Illustration of the case when NNS scheduling is not the optimal one for the BWD step. Tree mapping on 2 processors.

However, as illustrated in the following, one can still improve the local scheduling strategy to reduce the waiting-time of the NNS node. When the NNS node is not in the pool of ready-tasks, we can choose to process another ready-node (best with respect to the write sequence), knowing that each time an NNS node will become ready we will process it and will thus follow the order given by the factor write sequence. Let us illustrate the situation using the example of Figure 6.5 mapped on two processors. Factor size equilibration has driven the local ordering of the factorization per processor ($1, 2, 3, 4, 5, 6$ on processor P0 and $7, 8, 9, 10, 11, 12$ on processor P1). In this case the NNS node at the beginning of the backward step on processor P0 is node 6 (last node processed during the forward step on processor P0). However in our example node 4, mapped on the same processor, will be inserted in the pool much earlier (just after processing of the root node 12 on processor P1). If we use the NNS strategy, we will not process node 4 and will wait with a blocking receive until our NNS node (node 6) is ready. If we authorize node 4 to be processed as soon as its associated factors are loaded then we can process it with no delay because of disk access.

Our NNS algorithm has thus been revisited to reduce time spent in the blocking receive. In our new strategy, we can select a node ready in the pool that is not the NNS node. Thus we are not always blocked if the NNS node is not ready. However we also want to preserve regular access to the disk and limit the number of emergency calls. For this reason the prefetching mechanism continues to be related to the factor's write sequence. Each time we have to extract a node, when the NNS node is not in the pool, we extract the best node in pool (so called **BPN** node, for Best Possible Node) with respect to its position in the write sequence. Such a node is thus in the pool and has its associated factors loaded. Note that, since our prefetch mechanism always follows the write sequence, it means that *the NNS node factors have also been loaded.* (That is, on the example in Figure 6.5, when BPN node 4 is ready, the node factors for NNS node 6 are also loaded.) To extract the BPN node from the pool we keep the pool sorted with respect to the write sequence (the last node in pool is the closest to the NNS node in terms of the write sequence). We keep this property each time we insert a node in the pool.

Furthermore, it is easy to prove that if the factors of the BPN node have not been loaded from disk, then all the other nodes in the pool will not be in memory either. Thus this is a good way to limit the number of emergency calls while continuing to work when the NNS node is not available.

If the NNS node were ready in the pool, then it would be the last node in the pool and then it **is** the BPN node. We have thus changed the precondition of the blocking receive $\boxed{\beta}$ of Algorithm 6.1. With the BPN strategy this blocking receive is performed to wait for the NNS node only if no BPN node is in the pool.

Treating a BPN node earlier on a processor P might interact with the work of other processors only through slave tasks sent by processor P. Since at any rate such Type 2 slave tasks are processes 'on the fly' and do not go through the pool of the destination processor, Properties 6.1 and 6.2 are still valid. This proves the correctness of our new scheduling algorithm.

6.2.2 Experiments with BPN strategy

We report in Table 6.11 some preliminary results with QIMONDA07 matrix illustrating the potential of the BPN scheduling strategy. We focus in this run time performance and on the backward substitution step since it is the most sensitive to the scheduling strategy. We can see that the BPN strategy is in general faster than the NNS strategy. On eight processors the time for backward step (45.2 sec) with the NNS strategy is quite far from the minimum time to load factors (21.8 sec) and we see that with the BPN strategy the performance (24.5 sec) has been significantly improved. An important issue for the performance of the BPN strategy is related to the granularity of the read operations. Indeed, since it checks the state of the nodes to select the BPN one and since the state of the nodes is modified at the end of the corresponding I/O operation, it is more critical for the size of the I/O operations to be small (large enough to ensure a good performance for the I/O operations but not too big to avoid delays).

Strategy	Nb of Procs	T_min (sec)	Bwd (sec)	Nb_Req$^{(*)}$ in Bwd step	
				Prefetch	Emg
NNS	1	158.4	177.2	496	0
BPN			177.3		
NNS	2	79.9	93.7	250	0
BPN			89.7	370	0
NNS	3	57.9	65.5	174	1
BPN			69.7	178	2
NNS	4	41.3	50.5	117	0
BPN			45.1	127	0
NNS	6	31.5	37.9	93	0
BPN			38.8	93	0
NNS	8	21.8	45.2	57	0
BPN			24.5	66	1

Table 6.11: Influence of the scheduling BPN of the tasks on QIMONDA07. Emg=emergency buffer:1 MB; Prefetch buffer:10MB per processor; $^{(*)}$: Max per processor.

Furthermore, it is important to note that the BPN strategy may perturb the prefetching and memory management mechanisms. Indeed, with this strategy when the NNS node is not in memory, we may choose to process a node that already has its factor block in memory. At the end of the processing of the current BPN node, we free the memory area corresponding to its factor block. this may then happen on more than one BPN node before returning back to the processing of the NNS node. These memory areas (corresponding to the BPN nodes) can then to be used to prefetch data. The problem in this case is that the BPN nodes are not necessarily contiguous in memory which may induce memory management operations (compress to make free memory contiguous) in the prefetch buffer that can be costly. To further improve and stabilize the behaviour of the BPN strategy on should thus either modify/adapt the memory management mechanism or limit the activation of BPN nodes when we diverge too much form the NNS sequence.

It is why we feel that those results are preliminary and that some more algorithmic work is needed on this strategy.

Concluding remarks

We have described the main steps of a multifrontal algorithm for distributed forward and backward substitutions. We have shown that our original algorithms can be easily adapted for OOC execution. We have then compared two different approaches to read factors from the hard disk. In this context, a 'naive' SYSTEM_BASED OOC approach is not suitable mostly because of its large and unpredictable memory use.

A DIRECT_IO access to the disk with relatively small prefetch buffers has thus been introduced to control the memory use. In a sequential environment, we have first shown how critical the task scheduling can be. We have observed that one important issue is to control the number of hard disk accesses. Another issue is to obtain 'regular' disk accesses. While controlling the memory used, we then studied the parallel behaviour of our solver. We have shown that the task scheduling that is optimal in the sequential case is not efficient in a parallel context (LIFO scheduling). To obtain more regular disk access, especially for the backward step, we have constrained the scheduler to follow the factorization write sequence of factor blocks during the solve phase (NNS scheduling). We have proved the correctness of the algorithm and have shown that we perform consistently better and often significantly reduce the time for solution on a set of large real problems. Finally we have shown that the strict ordering resulting from our NNS scheduling can be relaxed to allow out of order execution of nodes with factors already loaded from disk (BPN scheduling).

Part II

Exploit Sparsity of Sparse Right-Hand Sides in OOC Environment

Résumé de la Partie 2: Exploitation de la nature creuse des seconds membres dans un environnement hors-mémoire (ooc)

Introduction

Jusqu'à présent nous avons exploité la structure creuse de la matrice d'origine et celle des facteurs. On se pose maintenant la question de comment utiliser la structure creuse du second membre.

Dans le cas où un grand nombre de seconds membres doit être traité, comme l'espace de travail nécessaire à la résolution croît linéairement avec le nombre de seconds membres, la mémoire n'est pas suffisante pour garder et résoudre en une fois tous les systèmes. Dans de telles circonstances, on divise les seconds membres par paquets et chaque paquet est résolu indépendamment des autres. En environnement hors-mémoire (OOC) les données du disque sont préchargées pour chaque paquet de seconds membres.

Souvent les seconds membres sont creux. On peut alors exploiter leur sparsité pour diminuer le nombre de calculs, mais surtout pour diminuer le nombre de données à précharger du disque dur. Comme le temps de la phase de résolution est dominé par le temps de préchargement des données, réduire la quantité des données à charger influencera fortement le temps de toute la phase de résolution.

Applications choisies

Nous avons choisi quelques domaines d'applications, où l'exploitation de la structure creuse des seconds membres peut être très utile: électromagnétisme, astrophysique et applications avec des matrices réductibles.

En l'électromagnétisme on s'intéresse au calcul d'une base du noyau de matrices déficientes (en d'autres termes au calcul de l'espace des vecteurs propres associé aux valeurs propres presques nulles). La dimension de cet espace peut être assez importante (jusqu'à 4000 vecteurs) et, faute de mémoire, l'obtention de cet espace en une seule fois n'est pas possible.

La deuxième application concerne le calcul de la variance et co-variance associé à un problème de moindres carrés linéaires, qui se réduit (dans notre cas) à calculer certains éléments diagonaux (la variance) ou non-diagonaux (co-variance) de l'inverse de la matrice d'origine. Si tous les éléments diagonaux sont requis (demandés à être calculer), cela signifie que le nombre de systèmes à résoudre est égal à l'ordre de la matrice.

Finalement, dans toute résolution impliquant des matrices réductibles, nous pourrons aussi exploiter la structure creuse et réductible des matrices pour diminuer le nombre de données à accéder du disque dur. La Figure 6.6 présente la forme d'une matrice de facteurs associée à une matrice réductible avec un second membre creux. On voit sur cette figure qu'une partie des neouds d'un seul des deux arbres (noeuds non barés) est concernée lors de l'étape de descente (forward step) de la phase de résolution.

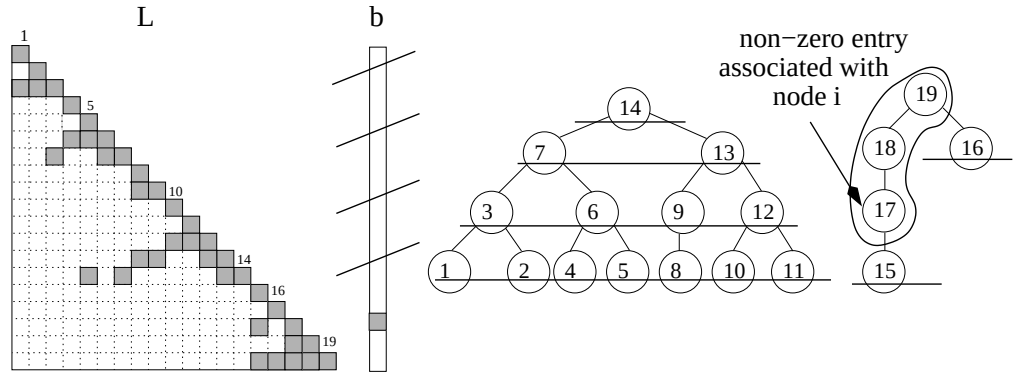


Figure 6.6: Exploitation de la nature creuse des seconds membres dans l'étape de descente. Les premières entrées nulles ne modifient pas la structure du vecteur y et ne sont pas utilisées dans les calculs.

Algorithmes d'élimination des données non-utilisables (prunning)

La principale propriété que nous utilisons pour prédire les facteurs utiles pour le calcul devant être accédés est basée sur le théorème de Gilbert et Liu dans leur article [62]. Elle relie la structure de la matrice des facteurs, et celle du second membre, à la structure du vecteur de solution.

Propriété 2. *Pour chaque matrice A et sa factorisation $A = LU$ (ou $A = LDL^T$), il est possible de prédire la structure du vecteur de solution à partir des noeuds associés aux entrées dans le second membre et suivant des chemins dans le e-dag de L^T suivi par des chemins dans le e-dag de U^T (ou L).*

Cette même propriété est utilisée dans les méthodes directes où l'arbre d'élimination représente un cas particulier du e-dag. L'algorithme de sélection des données utiles dans chaque étape de la résolution diffère d'une application à l'autre. On distingue deux classes, selon la manière dont les données ont été sélectionnées : sélection des branches et sélection des sous-arbres.

La sélection des branches est utilisée dans les applications de type moindres carrés, ou à partir de(s) noeud(s) associé(s) avec le(s) second(s) membre(s) tout(s) le(s) chemin(s) jusqu'à la racine de l'arbre est(sont) sélectionné(s) (voir Figure 6.6).

L'algorithme de sélection des sous-arbres est utilisé dans le calcul d'une base d'une matrice déficiente. Pour chaque ligne déficiente dans la matrice des facteurs, on parcourt tous les noeuds dans le sous-arbre du noeud associé à cette ligne (voir Figure 6.7).

Permutations

Après avoir identifié les données utiles, la phase de résolution se déroule 'normalement', sauf que le préchargement ne concerne que les données précédemment sélectionnées. Reste à voir comment les seconds membres sont divisés en blocs et s'il est possible d'améliorer encore la gestion des données à précharger.

A première vue, grouper les entrées de la matrice des facteurs associées au même noeud pourrait permettre de regrouper les seconds membres par paquets. Mais que ce passe-t-il si cela ne suffit pas? Une méthode intuitive consiste à grouper les seconds

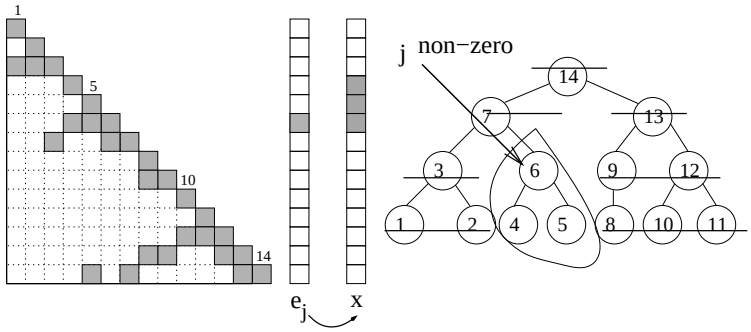


Figure 6.7: Sélection des sous-arbres: à partir d'un noeud tout le sous-arbre est sélectionné. Illustration de la structure du vecteur x obtenu après avoir résolu $Ux = e_j$ ($j = 6$).

membres en suivant un parcours de l'arbre de type post-ordre. Il faut noter que les noeuds proches dans l'arbre (du point de vue de leur chemin vers la racine et donc du post-ordre associé) partagent un grand nombre de noeuds de l'arbre et donc d'accés aux facteurs associés à ces noeud. Donc, le préchargement explicite des données pour la résolution d'un noeud va implicitement aider à la résolution (au préchargement des données) de l'autre noeud. Comme indiqué en Figure 6.8, les éléments $a^{-1}_{10,2}$ et $a^{-1}_{11,7}$, dont les noeuds sont proches dans l'arbre d'élimination, partagent une grande partie du chemin devant être parcouru lors de la résolution. Si les seconds membres des deux éléments sont regroupés, le préchargement supplémentaire de données pour calculer $a^{-1}_{11,7}$ sera d'un seul noeud, le reste étant déjà préchargé pour traiter $a^{-1}_{10,2}$.

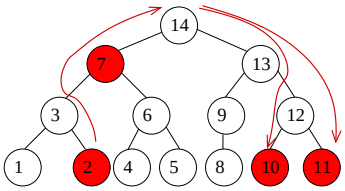


Figure 6.8: Chemin commun dans la forward substitution pour les éléments $a^{-1}_{10,2}$ et $a^{-1}_{11,7}$

Dans le même esprit, une permutation inverse du post-ordre (pré-ordre) pour les seconds membres aidera aussi à recouvrir les chemins entre les noeuds associés. Une dernière proposition de permutation, moins évidente que les précédentes, est de regrouper les seconds membres en fonction du poids total de données à précharger. Pour pouvoir modéliser ce poids, on utilise la notion d' hypergraphe. Les hypergraphes sont un ensemble de 'nets' (ensemble de noeuds) et d'arêtes. On utilise les nets pour représenter les différentes parties de chemin (les noeuds) dans l'arbre d'élimination. Ensuite, on associe des coûts à chaque net, correspondant aux données à précharger du disque dur. A la fin, en fonction de ces coûts, on choisit de grouper les nets entre eux, en regroupant ainsi les seconds membres. Minimiser le coût total revient alors à grouper les seconds membres de sorte que le coût total de préchargement de données soit minimum.

Chapter 7

Introduction

When solving $Ax = b$ in an out-of-core (OOC) environment, the time for the solution phase is dominated by the time for loading factors from disk. Therefore, the critical issue for applications using the solution phase intensively is not the flops count but the amount of factor data loaded. Note that, in our context, factors are loaded by blocks associated with nodes of the elimination dags (edags).

When the right-hand side b is dense there is not much we can do, since all nodes in the elimination dags are visited during the solution phase and hence all factor data are to be loaded. When the right-hand side is sparse we will show in this chapter that the structure of the elimination dags can be used to control the amount of factors loaded.

Of the many possible problems with sparse right-hand sides, we will focus on three of them:

- null-space vector computations [55] (with test matrices from electromagnetism),
- computing elements of the inverse of the matrix (with test matrices coming from least-squares data-fitting problems [67, 88] and from applications in astrophysics [23]),
- use of sparse right-hand sides with reducible matrices (with applications in linear programming [50]).

In this (second) part of the thesis, we will focus on these three problems. Our objective is to characterise the dependency graph of the computations that take place during the solution phase ($LUx = b$ or $LDL^Tx = b$) when b is very sparse, especially when it has a single nonzero entry. This graph will then be used to devise algorithms and models to optimise the load of the needed data from the disk.

In Chapter 8 we first introduce some background theory, relate it to our applications, and then discuss the important issue of sparse right-hand sides. We show that processing/pruning the elimination tree is needed. Building upon the notions developed in Chapter 8, we develop algorithms for tree pruning in Chapter 9. In the same chapter, we explain how some standard topological tree ordering methods, i.e., the pre- and post-orders, can be used to facilitate the partition of the columns of the multiple right-hand sides. In Chapter 10, we further propose a hypergraph model to partition the columns of the right-hand sides for reducing the amount of factor data loaded. In Chapter 11, we conclude this part of the thesis by discussing experimental results on real test problems.

Chapter 8

Exploiting sparsity of the right-hand sides: Context and applications

In this chapter, we first introduce some background theory on the relationship between the graph structure of A and the sparsity of the solution vector x (Section 8.1.1) using the notion and facts from the general introduction (Chapter 1)). We then describe in Section 8.1.2 two existing methods from the literature for computing entries in the inverse of a matrix and compare the amount of factors to load when computing a particular entry in A^{-1}. Finally in Section 8.2, for each of our three problems, we explain why we have to handle sparse right hand-sides and how this sparsity can be exploited. We also discuss in the same section the important issue of multiple sparse right-hand sides.

8.1 Context of our study

8.1.1 Relationship between the matrix graph and the structure of the solution vector

We provide in this section a summary of the results from Gilbert and Liu ([63]) by giving them as "Properties". For a more complete treatment of the interplay between the structures of a given matrix and the results of various computations, we refer the reader also to [64] and [62].

In [63] the authors provide a relationship between the structure of the original matrix A, the right-hand side b and the solution vector x. They show that the structure of x can be defined without computing x explicitly. The main property used in this section is stated as Property 8.4. This property results mostly from two sets of reasonings: first, the application of a theorem from [62] (cited as Theorem 2.1 in [63]); second, the use of the edags introduced in [63] to simplify the paths used in Theorem 2.1 of [62]. To follow this logic, we report a set of intermediate properties. We also comment on the simplification of the properties to the case of symmetric matrices or matrices with symmetric pattern, since they are used in our symmetric pattern multifrontal solver.

Let us first consider the equation $Lx = b$, where L is the structure of the lower triangular matrix. We study the structure of the solution vector x as a function of the structure of the right-hand side b and the structure of the lower triangular matrix L. In the following, we use notation presented in the global introduction of the thesis. Using

the nonzero pattern of L, the graph $G(L)$ is constructed. The next property expresses the relationship between paths in $G(L^T)$, the structure of the right-hand side b and the structure of the solution vector x. Note that the graph of L^T has the same structure as the graph of L but with reverse edges.

Property 8.1 (Theorem 2.1 in [63])**.** *For any lower triangular matrix L, the structure of the solution vector x to $Lx = b$ is given by the set of nodes, reachable from nodes associated with right-hand side entries by paths in the directed graph $G(L^T)$ of the matrix L^T.*

The property above is further developed in [63] to give the following result, liberating the previous one from the graph of L^T (in case L itself is result of a computation, e.g., LU factorization, the property specifies the structure without resorting to the computed L).

Property 8.2. *For any lower triangular matrix L, the structure of the solution vector x is given by the set of nodes reachable from nodes associated with right-hand side entries by paths in the edag of L^T.*

This properties says that it is enough to follow paths in the edags to get the structure of the solution, as the edag provides a precise representation of paths in $G(L^T)$ (as recalled in Chapter 1) the edag is the unique transitive reduction of the acyclic graph of L^T).

The properties of the structure of the solution vector x can be extended to any square non-singular matrix A. Let A have an LU factorization ($A = LU$) without pivoting.

We consider the two substitution steps of the solution phase of $LUx = b$:

$$\begin{cases} Ly = b \\ Ux = y \end{cases} \tag{8.1}$$

From the first line of equation (8.1) and Property 8.1, the structure of the vector y is given by the set of nodes reachable from the nodes corresponding to the nonzero entries in b by paths in the directed graph $G(L^T)$. Similarly, from the second line, the structure of x is given by the set of nodes reachable from the nodes corresponding to the nonzero entries in y by paths in the directed graph $G(U^T)$. Thus, for any matrix A with an LU factorization, the structure of the solution vector can be predicted as a function of the structure of its triangular matrices (L, U) and the structure of the right-hand side b.

We state this observation in the next property.

Property 8.3. *For any matrix A such that $A = LU$, the structure of the solution vector is given by the set of nodes reachable from the nodes associated with right-hand side entries by paths in the directed graph of L^T, followed by paths in the directed graph of U^T.*

Since the edags preserve all paths in the graphs of L^T and U^T, we can express Property 8.3 in terms of edags.

Property 8.4. *For any matrix A such that $A = LU$, the structure of the solution vector is given by the set of nodes reachable from nodes associated with right-hand side entries by paths in the edag of L^T, followed by paths in the edag of U^T.*

If the matrix A has a symmetric structure, the directed graph of L and the directed graph of U^T are equal and can be represented by the edag of L. In this case, the edag becomes a tree (the so called elimination tree) and one can just follow paths in the elimination tree to compute the values of the solution vector.

Example 8.1. *Consider the L factor given in Figure 8.1 of a matrix with a symmetric pattern. Suppose that the only nonzero entry in b is b_4. During the forward substitution step ($Ly = b$), all nonzero entries in y are obtained from b_4 by following paths in the graph of L^T.*

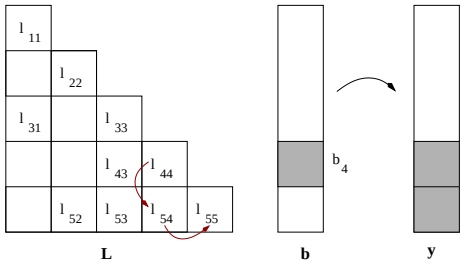


Figure 8.1: Influence of the structure of L and b on the solution vector y during the forward step.The arrows show all reachable entries from b_4 to the root $G(L^T)$.

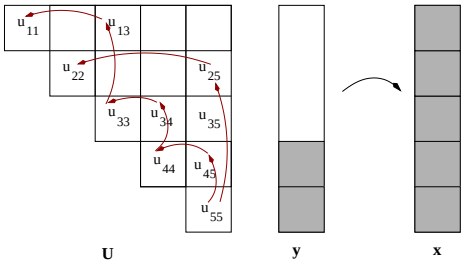


Figure 8.2: Influence of the structure of U and y on the solution vector x during the backward step. The arrows show all reachable entries from the root node in $G(U^T)$.

During the backward step ($Ux = y$), the structure of x will depend on the structure of y and can be obtained by following paths in the directed acyclic graph $G(U^T)$ as shown in Figure 8.2.

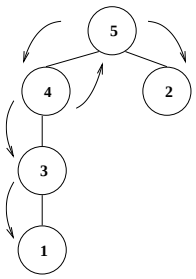


Figure 8.3: The elimination tree $T(A)$. The arrows show the path followed from the nonzero entry b_4 to construct the vector x.

The structure of x can be computed more efficiently from the structure of y through paths in the elimination tree. As shown in Figure 8.3, first the structure of y can be obtained by following the path from node 4 (associated with b_4) up to the root node 5 (following upward arrows, in other words, following the edges in the edag of L^T). Then the structure of x can be obtained by visiting the nodes reachable from the node 5 (following downward arrows, in other words, following the edges in the edag of U).

Note that if the matrix is irreducible, the unique root is reached during the forward step so that vector x will be completely full after the backward step [46]. All paths from the root will be taken in the backward step, see the backward step shown in Figures 8.2 and 8.3.

For **reducible matrices** the graph/elimination tree is **not strongly connected** and consists of the union of several connected components. Thus only a part of x will be updated during the backward step. We will further develop this idea in Section 8.2.1. Furthermore, on irreducible matrices when we are only interested in a specific entry in the solution x of $Ux = y$, then one can easily deduce from Property 8.4 that only paths from the root to this specific entry are needed. This is stated as a property.

Property 8.5. *Let us suppose that we want to compute x_j of $Ux = y$ with $y_n \neq 0$. Only the paths from the root to node j in the edag of U^T need to be visited.*

Note that the paths in the unsymmetric case from the root to a node may not be unique, as shown in Figure 6 of Chapter 1. In the context of a matrix with symmetric pattern Property 8.5 simplifies since the path from the root to node j is unique.

Property 8.6. *Let A be a matrix with symmetric pattern. To compute x_j of $Ux = y$ with $y_n \neq 0$ then only the path from the root to node j in the elimination tree need be visited.*

8.1.2 Background on computing entries in the inverse of a matrix

In many applications such as least-squares data-fitting problems [16, 20, 60, 61] and short circuit study [108, 109, 111] the inverse of the matrix or a subset of the inverse entries are very useful. Usually the inverse of a matrix is a dense matrix, even if the initial matrix is sparse [47, 58, 61]. Thus using A^{-1} as an operator is usually much less efficient than the direct use of the sparse LU factors.

Our main objective is to be able to efficiently compute some of the entries of the inverse matrix A^{-1}. If a direct relationship is established between the required entries in A^{-1} and the required part of the LU factors then our out-of-core application can be implemented more efficiently.

Let us suppose that we want to compute a few entries in A^{-1}. We first describe two methods to compute the inverse entries, based on the traditional solve phase of direct methods (Section 8.1.2.1), or based on Takahashi's equations (Section 8.1.2.2). Using Takahashi's equations and a recursive algorithm, Erisman and Tinney [54] proved that the subset of entries in A^{-1} corresponding to a nonzero position in the LU factors may be computed with only the LU factors and other entries in the same structure. In this context, Campbell and Davis [25] have shown that a multifrontal like approach can be introduced to use dense kernels and to show the dependencies in the computation. We focus in Section 8.1.2.3 on computing a few entries of A^{-1}. We compare the amount of factors to load using the traditional multifrontal and the alternative methods and determine the best method to use for our problems in Section 8.1.2.4.

8.1.2.1 Computing A^{-1} using traditional solution phase

Using the traditional solution phase of direct methods, we solve the system

$$AA^{-1} = I$$

where I is the identity matrix. Using the LU factors of A, the previous equation becomes:

$$\begin{cases} LY = I \\ U(A^{-1}) = Y \end{cases} \tag{8.2}$$

which corresponds to forward and backward substitutions of the solution phase. Note that in the forward step ($LY = I$) the right-hand side is very sparse and thus Property 8.1 can be applied to exploit the sparsity.

If all entries in A^{-1} are required, all the LU factors will be accessed. However, one might have (for memory issues) to compute columns in A^{-1} by blocks. In this case again, exploiting sparsity of the corresponding columns of I will be interesting. If only a few entries of A^{-1} are needed then only part of the LU factors will be accessed. We will further comment on this when comparing the traditional method with an alternative approach in Section 8.1.2.3. We will also explain in more details how to exploit the sparsity of the right-hand sides using Properties 8.4 and 8.5.

8.1.2.2 Takahashi equations and alternative methods to compute entries in A^{-1}

In this section we present an alternative method to compute the inverse entries of A using an LDU factorization of A, where L and U are respectively unit lower and unit upper triangular matrices and D is diagonal. Note that to obtain this LDU factorization from a standard LU factorization of a full rank matrix, it is enough to introduce a diagonal matrix D from the diagonal entries of U and use it as follows:

$$A = LU' = LDD^{-1}U' = LD(D^{-1}U') = LDU$$

The first direct use of the factors of a matrix to compute the inverse entries was done by Takahashi, Fagan, and Chin [111]. They relate the LDU factors of any nonsingular matrix A with its inverse entries $Z = A^{-1}$ in equations which are known as Takahashi's equations:

$$Z = D^{-1}L^{-1} + (I - U)Z \tag{8.3}$$
$$Z = U^{-1}D^{-1} + Z(I - L) \tag{8.4}$$

One way to establish the relation 8.4 is to note that from $Z = A^{-1}$ we obtain

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

which corresponds to equation (8.4). Equation (8.3) can be obtained in a similar way.

To compute entries in the upper part of Z one can simplify equation (8.3), since $D^{-1}L^{-1}$ is lower triangular and $(I - U)$ is strictly upper triangular matrix (see Figure 8.4). Equation (8.5) is thus derived from equation (8.3).

$$z_{ij} = d_{ij}^{-1} - \sum_{k>i}^{n} u_{ik}z_{kj} \quad , \; i \leq j \tag{8.5}$$

The following equation (8.6) is derived from equation (8.4) in a similar way:

$$z_{ij} = d_{ij}^{-1} - \sum_{k>j}^{n} z_{ik}l_{kj} \quad , \; i \geq j \tag{8.6}$$

Takahashi's equations do not use L^{-1} or U^{-1} and thus exploit the sparsity of the computations. Therefore, to compute the upper triangular entries of Z, equation (8.5) is used and to compute the lower triangular entries of Z, equation (8.6) is used. For the special symmetric case, a single equation can be used:

$$Z = D^{-1}L^{-1} + (I - L^T)Z \tag{8.7}$$

Some properties to note about the dependency of computations of the inverse entries:

- any off-diagonal entry $z_{ij}, \quad (j > i)$ directly depends on **all factors in its row** i (u_{ik}), and all z_{kj} in its column j (for $k > i$) , as we can see from equation (8.5).

$$z_{ij} = -\sum_{k>i} u_{ik} z_{kj}$$

- if a_{ii} is the only entry in row and column (i), then $z_{ii} = d_{ii}^{-1}$, since $u_{ik} = 0$ for all k in equation (8.5)

$$z_{ii} = d_{ii}^{-1} - \sum_{k>i} u_{ik} z_{ki}$$

Example 8.2. *Illustration of the dependency of the inverse entries*
We comment on a matrix with a symmetric pattern shown in Figure 8.5. As shown by the arrows, z_{33} *directly depends on the inverse entries* z_{43} *and* z_{53} *in its column and on all factors in its row (* u_{34} *and* u_{35}*). (If* u_{33} *were the only entry in row* 3, $u_{34} = 0$ *and* $u_{35} = 0$, *then* $z_{33} = d_{33}^{-1}$ *.) Finally,* z_{43} *and* z_{53} *directly depends on* z_{54} *and* z_{55} .

Using Takahashi's equations recursively, the whole matrix Z can thus be computed in reverse Crout order, starting from z_{nn} .

Erisman and Tinney [54] focus on the case when only a subset of the inverse entries is needed. They define an adjacency matrix C with the sparsity pattern of $L + U$. More formally $c_{ij} = 1$, whenever $l_{ij} \neq 0$ or $u_{ij} \neq 0$.

Using the definition of C , we define the subset $Zsparse$ as the subset of inverse entries with nonzero positions in the transpose of the matrix C .

$$z_{ij} \in Zsparse \Leftrightarrow c_{ji} = 1$$

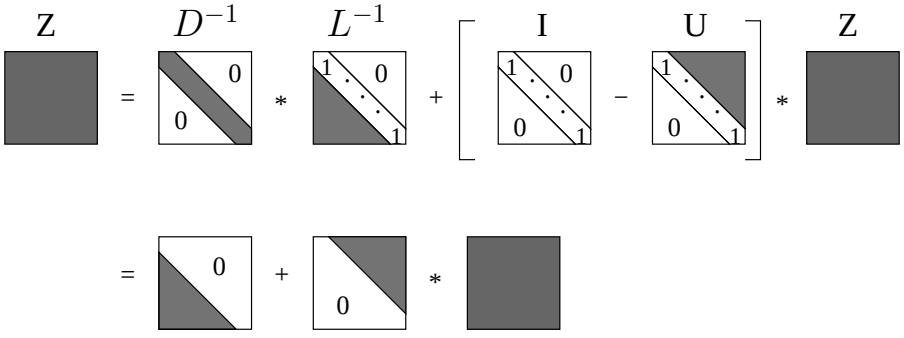


Figure 8.4: Takahashi equations: illustration of the simplification of equation (8.3) to obtain equation 8.5.

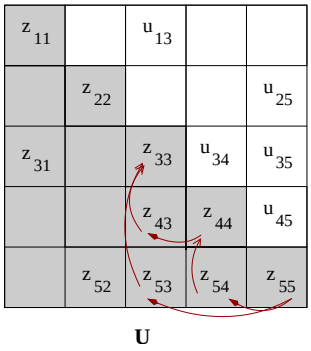


Figure 8.5: Pattern of LDU factorization of the matrix. Arrows represent direct dependency to compute z_{33} and z_{53} .

Therefore the structure of the transpose of the adjacency matrix and the structure of the subset of inverse entries with nonzero positions in C^T are identical : **Structure(** $Zsparse$ **) = Structure(** C^T **).**

Theorem 8.1 provides a recursive algorithm to compute any inverse entry in $Zsparse$.

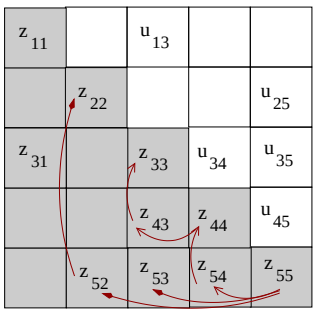
Theorem 8.1 (Theorem 1 in [54])**.** *Any* z_{ij} *where* $c_{ji} = 1$ *can be computed as a function of* L *,*U *and* z_{pq} *where* $c_{qp} = 1$ *,* $(q \geq j\,,\ p \geq i\,)$*.*

However, for a particular entry z_{ij} it may not be necessary to compute all of the entries in $Zsparse$, as mentioned in [54] and illustrated in the next example. Theorem 8.1 only provides a sufficient set of elements to compute entries in $Zsparse$.

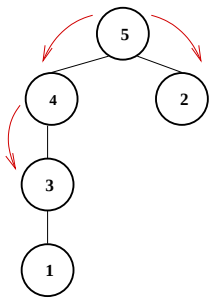
Example 8.3. *We compute entry* z_{22} *of a matrix with a symmetric pattern and associated factors shown in Figure 8.6. Starting from* z_{55} *and working in the reverse Crout order, all the entries*

$$z_{55}, z_{44}, z_{54}, z_{33}, z_{43}, z_{53}, z_{22}, z_{52}$$

will be computed. Note that only z_{22} *,* z_{55} *and* z_{52} *are needed:*
from equation (8.5) $z_{22} = d_{22}^{-1} - u_{25}z_{52}$
from equation (8.6) $z_{52} = -l_{52}z_{55}$
finally, $z_{55} = d_{55}^{-1}$ *.*



a) Pattern of LDU factorization of a matrix with symmetric pattern.



b) The elimination tree captures the dependency between entries of A^{-1} in a top down traversal of the tree.

Figure 8.6: Dependency of computations in Z_{sparse} .

Note that for the symmetric case $U = L^T$ so $l_{ji} = u_{ij}$ and $c_{ji} = c_{ij} = 1$. Thus the structure of C^T becomes the same as that of C ($C^T = C$) and paths in C can be represented by paths in the elimination tree $T(A)$. Finally $z_{ij} = z_{ji}$ and only one of the equations (8.5) or (8.6) will be used to compute all of the inverse entries.

Campbell and Davis have shown that for symmetric matrices the elimination tree captures the dependency between entries of A^{-1}. On our example node 2 is a son of node 5, as shown in Figure 8.6. The first nonzero entry in row 2 of U is thus in column 5 so that entries in rows 3 and 4 of $Zsparse$ need not be involved in computation.

Computing entries outside $(L|U)^T$: To compute z_{ij}, using the Takahashi equations where $c_{ji} = 0$, the entry c_{ji} has to be set to 1. Then one should update all possible fill-in related to the c_{ji} entry. Theorem 8.1 can thus be applied to the updated $Zsparse$ matrix. The modified adjacency matrix C^T_{new} captures the structure of all entries z_{pq} required to compute the entry z_{ij} outside the original $Zsparse$.

Campbell and Davis [25] focus on the special case of computing complete $Zsparse$ on numerically symmetric matrices. They prove that when computing $Zsparse$ with the Takahashi equations, the elimination tree processed from the root to the leaf nodes captures the dependency relationships between all the inverse entries in $Zsparse$. Then they explain how Level 3 BLAS can be used in this context. Note that the elimination tree can then also be used to parallelize the computation of entries in $Zsparse$. The method proposed by Campbell and Davis differs from other parallel implementations of computing the inverse entries [6], by the use of the dense kernels for Level 2 and 3 BLAS optimization.

8.1.2.3 Computing a few entries in Z_{sparse} of a matrix with symmetric pattern

We are interested in characterizing the LU factors needed to compute a few entries in $Zsparse$. It is an important issue in an out-of-core (OOC) environment, since the time for loading factors often dominates the computation part. By exploiting the sparsity of the right-hand sides, we can determine which factors are needed and load only that data from disk. If only a part of the factors are loaded, we can significantly improve performance.

We will compare two methods: the traditional solution phase and the one based on Takahashi equations. Each time Property 8.1 will be used to characterize dependency between the right-hand side and the solution vector.

We will consider the case of a symmetric matrix, where the structure of C^T is equal to the structure of C.

a) **Using Takahashi equations**

In [25] a relationship between the elimination tree and $Zsparse$ is established. To compute a particular entry z_{ij} we will have to compute all inverse entries in $Zsparse$ from the root node to the nodes i and j in the associated elimination tree.

We want to go further by proving that the only factors which have to be loaded/used to compute a particular entry z_{ij} are situated on a path from nodes i and j up to the root. Thus the other branches of the elimination tree are not involved and their factors need not be loaded. The amount of loaded data will thus be significantly reduced.

Without loss of generality, we can assume that $j > i$. If z_{ij} is needed, then $u_{ij} \neq 0$. From the elimination tree properties, node j is an ancestor of node i, and there is a path in the elimination tree from i to the root going through j. We can thus state this as a property.

Property 8.7. *For any $z_{ij} \in Zsparse$ ($j > i$), nodes i and j are on the same path to the root of the elimination tree of the symmetric matrix A.*

Note that for any $z_{ij} \notin Zsparse$, the structure of the adjacency matrix C will be modified by setting $c_{ij} = 1$. Thus node j becomes an ancestor of node i, and they belong to the same path to the root of the resulting elimination tree.

Property 8.8. *To compute a particular entry z_{ij} ($j > i$), in the inverse of a symmetric irreducible matrix A the only factors that need to be loaded in $(D^{-1}U)$ are on the path from node i up to the root node.*

Proof: We start by computing the inverse of the root node n, and we will use recursion to prove the property. Node z_{nn} directly depends only on d_{nn}^{-1} ($z_{nn} = d_{nn}^{-1}$). Let m be the child of the root node on the same branch as nodes i and j. Then z_{mn} depends on the nonzero factors u_{mn} and z_{nn}. Thus z_{mn} recursively depends on u_{mn}, d_{mm}^{-1} and d_{nn}^{-1}. Thus we have proved for the root node and its child m that the only factors which have to be loaded from $(D^{-1}U)$ are on the path from node m to the root node n.

Suppose that i is the first node in this branch which depends on some factor u_{st} not belonging to the path from i to the root. From Takahashi's equations we must have $s \geq i$ and $t \geq j$. Let node j be the parent of node i. Then z_{jj} recursively depends only on factors in its path to the root node (i was the first node not having the property in this branch). Since all recursive dependencies come from the parent node j, then u_{st} must be in the same row as u_{ij}, and thus $s = i$. Node i directly depends on factor $u_{it} \neq 0$ in its row ($i \leq t, j$). With respect to the elimination tree property, this means that node t is an ancestor of node i and belongs to the same path to the root node. We obtain a contradiction with the assumption that u_{st} does not belong to the path from node i up to the root node n. We have thus proved that the only factors which have to be loaded for a particular entry $z_{ij,\ (j \geq i)}$ in $(D^{-1}U)$ are on the path from node i to the root node n. $\diamond$

b) **Using traditional solution phase**

To compute a specific entry a_{ij}^{-1} in the inverse of the matrix using direct methods, we have shown that we can solve equation $AA^{-1} = I$ using a direct approach:

$$a_{ij}^{-1} = (A^{-1}e_j)_i \tag{8.8}$$

where $A^{-1}e_j$ is column j of A^{-1} for which we are only interested in entry i, $(A^{-1}e_j)_i$. We show in this section that our general framework can be used to exploit sparsity during the computation of entries in A^{-1} with a direct approach.

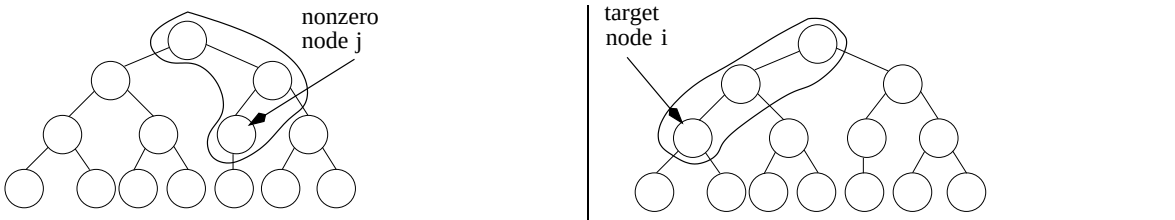
Decomposing equation (8.8) into forward and backward substitution we obtain:

$$y_j = (L^{-1}e_j) \tag{8.9}$$
$$a_{ij}^{-1} = (U^{-1}y_j)_i \tag{8.10}$$

Note that at each solution step either the right-hand side is sparse (equation (8.9)) or only a specific entry in $U^{-1}y_j$ is needed (equation (8.10)). In the following we show that

only part of the factors needs to be accessed and that we can use previously introduced pruning algorithms at each step.

Knowing in advance the columns of the required entries in A^{-1}, we can predict the factor data needed, by applying Property 8.4 of Section 8.1.1 for the structure of the solution vector.



forward step:*from the nonzero entry up to the root*

backward step: *from root node to a particular entry*

Figure 8.7: Nodes that must be visited to compute entry $z_{ij} \in Z_{sparse}$

Applying Property 8.4 to equation (8.9) we can state that all nonzero entries in vector y follow the path from the nonzero entry - node j, up to the root (see Figure 8.7 - forward step). Similarly from equation (8.10) and Property 8.5 we can state that to compute a_{ij}^{-1} one must follow paths in the elimination tree with a top-down traversal (see Figure 8.7 backward step).

We summarize the previous observations in the following property.

Property 8.9. *To compute a particular entry z_{ij} 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.*

Note that, since we use equations (8.9) and (8.10), our a_{ij}^{-1} need not be in $Zsparse$ (i.e. no longer related to $u_{ij} \neq 0$).

8.1.2.4 Conclusion about the method to use

To conclude, let us compare the amount of LU factors that have to be accessed with the proposed approaches – based on Takahashi equations or based on a traditional solution phase. The following properties are direct consequences of the previous properties and can easily be generalized to matrices with unsymmetric pattern, where the elimination tree will be replaced by the edags.

Property 8.10. *Let A be an unsymmetric irreducible matrix with symmetric pattern and let T be its corresponding elimination tree (one variable per node). For both approaches to compute a_{ii}^{-1} we need to access all rows of U and columns of L from node i of T to the root.*

Property 8.11. *Let A be an unsymmetric irreducible matrix with symmetric pattern and let T be its corresponding elimination tree (one variable per node). To compute an off diagonal entry entry a_{ij}^{-1} both the columns of the L factors from node i of T to the root and the rows of the U factors for node j of T to the root need to be loaded with both approaches.*

One should also add that, with both approaches, to compute an entry in A^{-1} the other entries of A^{-1} that need be computed are identical. In our OOC context, there is thus no benefit in terms of access to LU factors between both methods. Furthermore we would like our approach to address other applications with sparse right-hand sides (null-space computation, solution with reducible matrices). Property 8.4 thus provides a natural common framework to efficiently exploit sparsity for all our target applications including computing entries in the inverse of A.

8.2 Sparsity of the right hand-sides and applications

In the previous section, we have shown that it is possible to compute x by using only part of the factors when the right-hand sides are sparse. Our purpose in this section is to show how the properties introduced in the previous section can be used on our application in the context of a parallel out-of-core multifrontal solver. We then discuss for each application the issue of processing multiple right-hand sides and comment on memory issues.

We will assume that our matrix is symmetric in structure and the edag associated with the factors is a tree, the elimination tree.

8.2.1 Sparse right-hand sides / reducible matrices

The dependency graph of reducible symmetric matrices can be structurally represented by disconnected trees (or a forest). Each part of the forest is a completely independent tree, not reachable from other trees (see Figure 8.8).

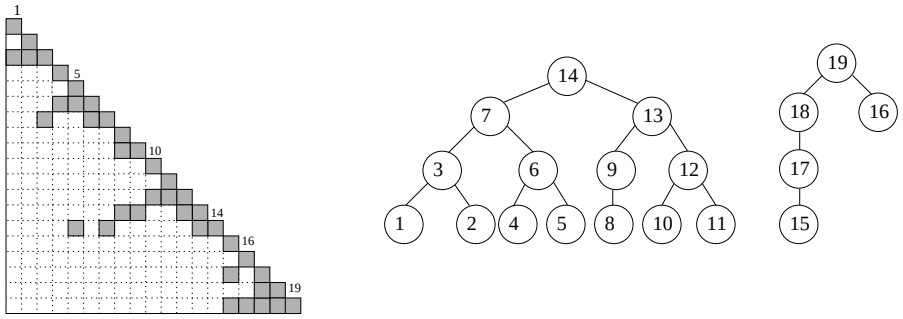


Figure 8.8: Reducible symmetric matrix pattern and its associated forest

For reducible matrices, exploiting the sparsity leads to working on the tree associated with the nonzero entry in the right-hand sides. The disconnected part of the forest will not be concerned by the computations, because it is not reachable (see Property 8.4).

We can also apply Property 8.4 during the forward substitution to sparse right-hand sides. As shown in Figure 8.9), when the first entries from the top of the right-hand side are zeros they will not modify the zero structure of the solution vector y. The first node which has to be taken into consideration for computations in this step is the first nonzero entry in the right-hand sides. Then we follow paths in L or in the edag associated with L (which is a tree/forest for a symmetric matrix); using the reachability of the nonzero entries. For example, on our test example shown in Figure 8.9, there is a single entry in

the right-hand side – b_{17} . We need to compute only factors associated with nodes $17, 18$ and 19 to complete the whole forward step.

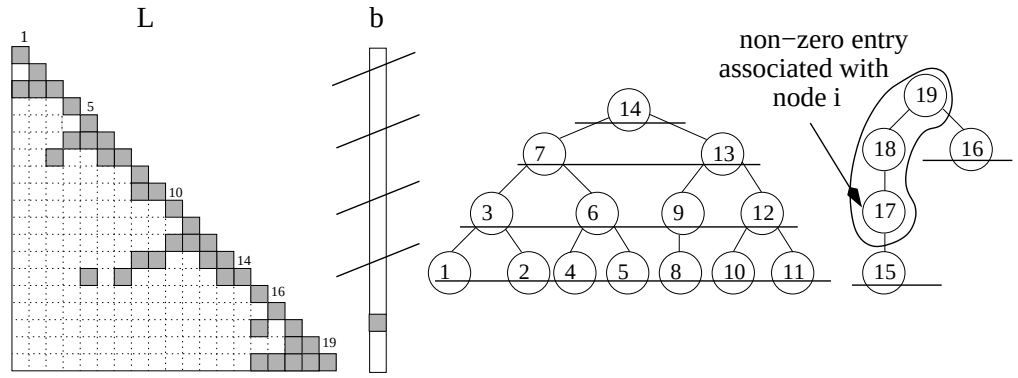


Figure 8.9: Exploiting the sparsity of the right-hand sides in the forward substitution. The first zero entries in the right-hand sides do not modify the structure of the vector y and thus are not used in the computations.

We use **'tree pruning'** to refer to the mechanism of suppressing unneeded nodes of our original elimination forest. In Figure 8.9 we illustrate the pruning mechanism on our elimination forest from Figure 8.8, where only some of the nodes in the tree are kept and the other nodes are pruned.

During the forward substitution, for each entry in the right-hand sides, all the ancestor nodes of the associated node in the tree need to be considered. The union of all needed nodes represents branches of the elimination tree (chains of ancestors up to the root node). As this functionality selects nodes in the branch from the node to the root of the elimination tree, we will name it **'branch detection'**.

When a backward substitution follows a forward substitution with sparse right-hand sides, all the root nodes reached during forward phase will correspond to a nonzero entry in the right-hand side of the backward step (on our example root node 19 has been reached during the forward step). During the backward step is it enough to consider all the root nodes reached during the forward steps to determine the structure of the solution (we follow paths in the in the edag of U^t which is a tree in our symmetric case). The pruning mechanism where a whole (sub)tree will be processed will be referred to as **'subtree detection'**.

Both 'branch detection' and 'subtree detection' will be described in more detail as algorithms in Chapter 9.

8.2.2 Null-space computations

We describe null-space computations in the context of rank revealing LU factorization code on general unsymmetric matrices. We suppose that so called 'null pivot rows' have been detected during factorization. In fact to do so we combine two approaches. We first perform a 'normal' factorization with modified numerical pivoting strategies and detect small pivots 'on the fly'. Some of the small pivots will be considered as null pivots and some of them will be postponed potentially up to the root of the elimination tree. At the root of the elimination tree a more standard rank-revealing approach based on QR factorization with column pivoting is used. This work is the object of a collaborative task

within the SOLSTICE ANR project (ANR-06-CIS6-010) between CERFACS, INRIA-LIP and IRIT.

The rank-detection performed during factorization phase leads to two types of deficient row entries: null pivots associated with the root of the elimination tree (root-deficiency) and those detected in the lower level of the tree. We will see in this section that root-deficient rows provide no scope for exploiting the sparsity of the right-hand-sides during null-space computation.

By 'null-space computations' we mean solving the equation

$$Ax = 0. \tag{8.11}$$

The solution set is then named as the **null-space of** A. Using direct methods, this equation becomes:

$$\begin{aligned} &A = LU, \quad LUx = 0, \quad det(L) \neq 0 \\ &Ux = 0 \end{aligned} \tag{8.12}$$

As we can see, only the backward substitution has to be performed. Of course, $x = 0$ is always a solution of the equation, but in some cases there are more x vectors satisfying equation (8.11). If the matrix has zero (or numerically very small) entries in a whole row of U, then the matrix is numerically **deficient** and there exists a nonzero vector x solution to the equation (8.11).

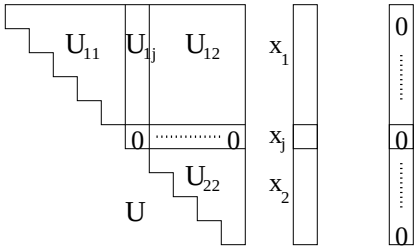


Figure 8.10: Case of 'zero line' in U : U is divided into blocks with respect to 'zero row' detected.

To simplify our discussion, let us first suppose that the matrix U has the structure described in Figure 8.10.

We can thus write the following system of equations:

$$\begin{cases} U_{11}x_1 + U_{1j}x_j + U_{12}x_2 = 0 \\ 0x_j + 0x_2 = 0 \\ U_{22}x_2 = 0 \end{cases} \tag{8.13}$$

Here $x_2 = 0$ is a solution of $U_{22}x_2 = 0$ (it might not be the only one, if U_{22} is rank deficient). From the second line of equation (8.13) x_j is free. From the first line, we have thus to solve:

$$U_{11}x_1 = -U_{1j}x_j \tag{8.14}$$

We can set x_j to 1, and thus

$$U_{11}x_1 = -U_{1j}\ . \tag{8.15}$$

The final null-space vector is thus $x = (x_1\ 1\ 0)^T$.

Note that if the entry u_{jj} of U is set to 1 during the factorization, and if we set the right-hand side to e_j, then by solving $Ux = e_j$ we obtain the same set of equations.

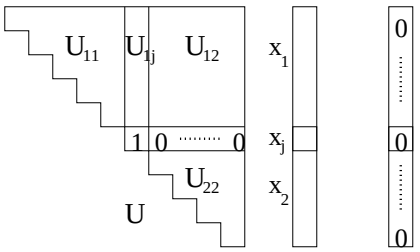


Figure 8.11: Case of 'zero line' in U with modified factorization

Our final equation for solving $Ax = 0$ with the described modified factorization is thus:

$$Ux = e_j \tag{8.16}$$

In general, the structure of U is not so simple and the block U_{22} might itself be rank deficient (with again null or 'pseudo null' rows in U_{22}). We can easily generalize the computations associated with one pseudo-null pivot to a more general case as described in the following. Let us suppose that we solve $Ux^j = e_j$ for each pseudo null pivot j detected during the factorization. We compute them in the order in which they have been detected following the tree from the leaves to the root. Then the solution of $UX = E$, where E is the set of e_j columns, and X^j is the j^{th} column of X, solution of $UX^j = e_j$, will have by construction a structure as shown in Figure 8.12.

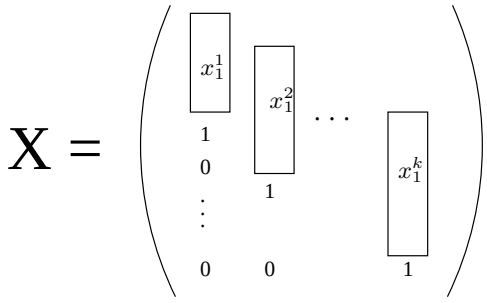


Figure 8.12: Structure of X

Therefore, by construction, each vector is linearly independent from the others and we get a basis of the null-space associated with the pseudo-null pivots detected during factorization. If all 'null' pivots have been detected ('accurate' rank revealing LU factorization) then we have a full basis of the null-space of A. The number of columns k of X is then the **deficiency** of the matrix A or the dimension of its null-space basis.

8.2.2.1 Some properties

The memory needed by our parallel multifrontal solver during the solve phase includes the number of the solution vectors, the order of the largest frontal

matrix($Size(Max_Frontal_Matrix)$), and a working area. Both the working area and the solution vector are of size $N \times deficiency$, where N is the order of the matrix, so that our peak of memory is:

$$Peak\, memory = 2\ N \times deficiency + Size(Max_Frontal_Matrix)$$

Thus, if all columns of X are computed in one pass then the memory requirement will be very large on matrices with high deficiency. For example, on our test matrix from electromagnetism (see Table 5 in Section 1.3) the deficiency is larger than 4 000 on a matrix of order $33\ 000$). To address this issue which is common to the general case of processing multiple right-hand sides, a blocking factor N_s is introduced so that columns of the right-hand sides are processed by block of size N_s . In our case, we thus divide X into s blocks of size N_s :

$$X_j \in \{X_1, ..., X_s\}\ , \qquad |X_j| = N_s$$

Then $UX = E$ is solved by s blocks of N_s right-hand sides at a time. The working space for the backward phase becomes a function of N_s :

$$Peak\, memory = 2\ N \times N_s + Size(Max_Frontal_Matrix) \tag{8.17}$$

Property 8.12. Total size of factors to be loaded (without exploiting the sparsity of the right-hand sides) *:*
The total size of the factors to be loaded ($Factors_loaded$) during the solution phase is equal to the sum of factors U loaded at each block iteration, i.e.

$$Factors_loaded\ =\ s \times |U| \tag{8.18}$$

$$If\ N_s = 1\ :\qquad Factors_loaded\ =\ |U| \times s$$
$$If\ N_s = deficiency\ :\quad Factors_loaded\ =\ |U|$$

8.2.2.2 Pruning for null-space computations: subtree detection

The equation to be solved is $Ux\ =\ e_j$, where e_j is a sparse right hand-side (the only nonzero entry corresponds to the null pivot j in the original matrix. Node j of the elimination tree being the node on which row j of U was computed during the factorization.) This nonzero entry becomes the starting root node for our 'pruning' mechanism for the backward step, as it was in the backward step for reducible matrices.

Property 8.4 of the reachability in the elimination tree can thus be used; from this starting node the whole subtree has to be loaded (and not only one of the ancestors as was the case in 'branch detection'). We state this as a property.

Property 8.13. *When solving $Ux\ =\ e_j$ only the subtree rooted at node j of the elimination tree need be processed.*

The previous property implies that the complete elimination tree need be visited when a null pivot is located at the root of the elimination tree.

Example 8.4. *To illustrate this we show the pattern of* L, e_j *and* x, *and the corresponding elimination tree of* L *in Figure 8.13. We suppose that the structure of* U *is equal to the structure of* L^T. *The nonzero entry in* e_j *(*$j = 6$*) broadcasts its nonzero pattern on the solution vector to the whole subtree rooted at node* $j = 6$.

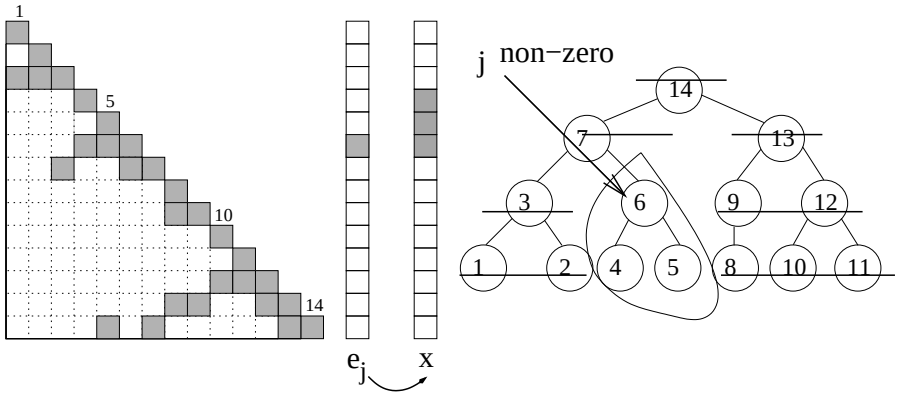


Figure 8.13: 'subtree detection' for backward step with sparse right-hand sides. Illustration of the structure of x obtained after solving $Ux = e_j$ ($j = 6$) and of node traversal during computation.

Complete subtrees are always concerned during null-space computations, so that we will use the term **'subtree detection'** to refer to this type of tree-pruning algorithm.

Finally, we can define a lower bound to the size of the factors to be loaded in a sequential environment. It is related to the size of each node of the elimination tree, to the number of requests for each node ($nb_requests(node)$), and the number of blocks (s) solved. We define $nb_requests(node)$ as the sum of the number of requests to this $node$ and the number of requests of its direct parent node. (Example: if the root is requested twice, and its son is requested three times, the $nb_requests(root) = 2$ and $nb_requests(son(root)) = 2 + 3 = 5$.)

Property 8.14. Lower bound of the amount of factors to load in a sequential environment: *The lower bound of the size of the factors to be loaded is the sum of the products of the size of each node multiplied by the number of its requests (*$nb_requests$*), divided by the number* s *of blocks solved:*

$$\sum (size(node) * nb_requests(node)/s)$$

8.2.3 Computing entries in A^{-1}

To compute a specific entry a_{ij}^{-1} in the inverse of the matrix using direct methods, we have shown that we can either use the Takahashi equations or solve equation $AA^{-1} = I$ using a traditional solution phase. In the following we show that only part of the factors needs to be accessed and that we can use previously introduced pruning algorithms at each step.

When solving $Ly_j = e_j$, as explained in Section 8.2.1, factors of L associated with nodes from the starting node j and all nodes in the path to the root node are needed.

We will thus use here the 'branch detection' algorithm to characterize nodes of the tree required for this step.

At the end of the forward step, entries in y_j corresponding to the path from node j to the root node of the tree to which it belongs are nonzero. Let us assume for the sake of clarity that our matrix is irreducible and has thus a single root node in its associated elimination tree. In this case, as shown in Figure 8.14, the solution vector is full. However, we are interested in only the i^{th} entry, in the solution vector. In this case

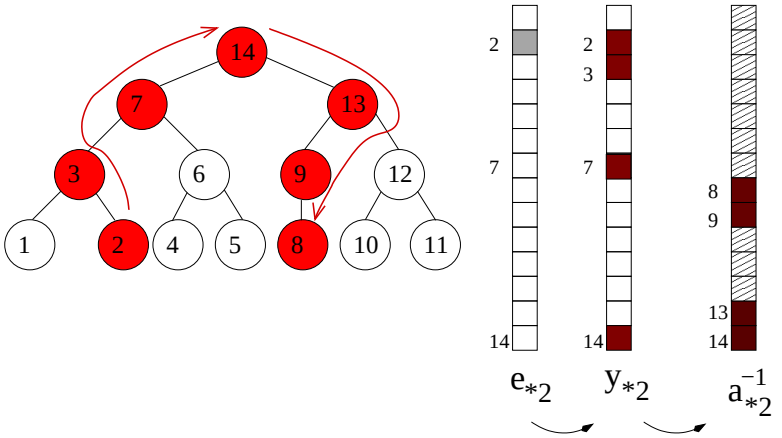


Figure 8.14: Illustration of tree traversal to compute $a^{-1}_{8,2}$. $a^{-1}_{*,2}$ corresponds to column 2 of A^{-1}. Filled entries in $a^{-1}_{*,2}$ correspond to entries in column 2 of A^{-1} while computing $a^{-1}_{8,2}$. $a^{-1}_{8,2} = (U^{-1}L^{-1}e_2)_{i=8}$; $y_2 = L^{-1}e_2$; $a^{-1}_{8,2} = (U^{-1}y_2)_{i=8}$;

pruning can be done. Note that the pruning algorithm here will be similar to the one in the forward substitution. We illustrate this complete process in Figure 8.14. We have first to load L factors associated with nodes $2, 3, 7, 14$ during the forward step. We then load U factors associated with nodes $14, 13, 9, 8$ during the backward step. Thus, for both substitution phases only a single path in the tree is concerned. Note that while computing $a^{-1}_{8,2}$ all entries on the path from the root to node 8 will correspond to entries in column 2 of A^{-1} that are computed.

8.2.3.1 Computing multiple entries in A^{-1}

In practice, more then one entry in A^{-1} is often requested. For example all diagonal entries are requested to estimate the variances in least-squares data-fitting problems. Thus, we are also interested in computing a few entries in A^{-1}.

In this case during the forward step we consider equation

$$LY = E \tag{8.19}$$

where each column of E is a column if the identity matrix. For example, $e_j \in E$ expresses the fact that at least one entry in column j of A^{-1} (a^{-1}_{*j}) is requested. Then during the backward step we need to solve:

$$Ua^{-1}_{*j} = y_j \tag{8.20}$$

and we are only interested in computing part of the solution vector. Our user interface to this problem can be very simple and efficient in terms of memory usage. We use a sparse

array to characterize all the requested target entries in A^{-1}. On output, the same array holds the entries of A^{-1} requested. However, for larger matrices, computing entries in A^{-1} need still be divided into s blocks of size N_s, for memory issues, as observed for the null-space basis computations with large deficiency (see Section 8.2.2.1).

In the following enumeration, we list simple observations illustrated on Figure 8.15 that will be used to understand why processing columns of the right-hand sides in the same block can be interesting. Those observations will be formalized in Chapter 9 to guide the design of our algorithms.

1. Several entries in a column of A^{-1} are computed at once (as shown in Figure 8.14: entries $a^{-1}_{14,2}$, $a^{-1}_{13,2}$, $a^{-1}_{9,2}$, ... are computed during the computation of entry $a^{-1}_{8,2}$).

2. When computing a diagonal entries ($a^{-1}_{i,i}$) the same 'branch' for both forward and backward steps is concerned (for example $a^{-1}_{2,2}$ will have an ascending path from node 2 up to root and descending path again to node 2).

3. Requested entries with the same row number use the same path in the backward step (for example, in Figure 8.15 entries $a^{-1}_{8,2}$ and $a^{-1}_{8,5}$ have the same descendant ascendant path from the root to node 8).

4. Requested entries with the same column number use the same path in the elimination tree in the forward step (for example, in Figure 8.15 entries $a^{-1}_{8,2}$ and $a^{-1}_{11,2}$ will use the same path from node 2 to the root).

5. Combining properties of the previous two cases: if requested row/column entries are associated with a common ancestor node in the elimination tree, they share the branch in the elimination tree from the ancestor node to the root (example: factors loaded to compute $a^{-1}_{8,2}$ completely include factors needed to compute $a^{-1}_{9,3}$, see Figure 8.15).

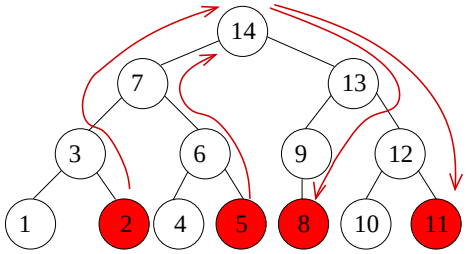


Figure 8.15: Paths in the elimination tree to compute entries $a^{-1}_{8,2}$, $a^{-1}_{11,2}$, $a^{-1}_{8,5}$ and $a^{-1}_{11,5}$.

8.2.4 Pruning and concluding remarks

To exploit sparsity in the right-hand sides, we have identified two mechanisms of tree pruning – branch detection and subtree detection to characterize entries in the factors that need to be loaded in an out-of-core context. For reducible matrices, branch detection allows us to follow paths in the forward step, and the subtree detection, when used during the backward step, identifies the tree rooted on the nonzero requested entry. For computing entries in A^{-1} we use branch detection during both forward and backward

substitution steps. In null-space computation, only subtree detection is used during backward substitution.

In the context of multiple right-hand sides, we have explained why we need to split the right-hand sides by blocks, for memory issues. The difficulty is then to decide how to efficiently order the right-hand sides to better exploit the branch and subtree detection algorithms.

In a parallel context, the situation becomes even more complex. In this case we want to combine the efficiency of exploiting the sparsity of the right-hand sides with balancing the amount of work among the processors. Thus, the partitioning of the right-hand sides should also take into account the data mapping of the factors onto the processors.

Chapter 9

Algorithms to exploit sparsity

9.1 Introduction

In this chapter we describe the algorithm for pruning the elimination tree. 'Pruning tree' algorithms change the tree structure. Almost all the global information is changed - number of leaf nodes, number of roots (in case of reducible matrices), number of sons per node. Two algorithms are presented, which we name with respect to the final graph, branch detection and subtree detection. The branch detection corresponds to a bottom-top traversal of the tree, starting from a (several) node(s) and going up to the root node. The subtree detection corresponds to a top-down traversal of the tree where for each local root node the whole subtree information will be kept. Finally we propose some permutations of the right-hand sides to better exploit the sparsity when the multiple right-hand sides are grouped into blocks for memory issues.

9.2 Pruning algorithms

For each initial nonzero entry in the right-hand sides we obtain its node number in the tree (*Inode*). We call these nodes **starting_nodes** for our pruning algorithm. Then using the dependency relationships (ancestor/descendant), we determine all the needed data. Note that for both branch and subtree detection algorithms, if an already selected node is encountered, we know that the remaining part of the tree is already selected. We thus cycle to the next entry in the initial set of starting nodes.

Note also that both algorithms work with single or multiple right-hand sides and with single or multiple entries per right-hand side.

9.2.1 'Branch detection'

The 'branch detection' refers to the detection of all branches from a initial set of nodes to the root(s) of the elimination tree. We recall that 'branch detection' can be involved during the forward step to exploit sparsity with reducible matrices or to compute entries in A^{-1}. It can also be used during the backward step when computing selected entries in A^{-1}.

We first mark all nodes as non-visited. Then we mark nodes in the tree belonging to the starting_nodes list associated with nonzero entries in the right-hand side, as described in Algorithm 9.1

Algorithm 9.1 : branch detection

for each 'non-visited' node in the starting_nodes **do**
  mark it and mark all nodes as 'visited' on the path to the root up to next visited node
end for

A detailed algorithm for branch detection is provided as Algorithm 9.2 and is commented in the following.

The first node is taken from the starting_node set. It is marked as 'visited' with all its ancestors up to the root, or up to the first already marked node. Then another node is taken from the starting_node set and, if it is not visited, it is marked in a similar way.

The list of selected leaf nodes is defined at the end of the algorithm, as the subset of the starting_nodes with no children. The list of selected leaf nodes will be used later during the forward substitution to initialize the pool of tasks ready to be processed or during the backward substitution to count the number of leaves to be processed before ending. The list of roots is set on the fly by the branch detection algorithm (each time a node without a father is encountered).

The number of selected sons – *nb_sons(Inode)* of each node has also to be computed since it is used during the solution phase to decide when the node can be activated (forward substitution) and added to the pool of task ready to be processed. We initialize *nb_sons(Inode)* to zero. It is then incremented during the algorithm even if the father of the current node has already been visited.

Algorithm 9.2 : detailed branch detection

Input: ELIMINATION_TREE, STARTING_NODES.
Output: SELECTED_TREE *(list_selected_leaves, list_selected_roots)*

nb_sons(*Inode***)** - selected sons of *Inode*, initialized to 0.
father(*Inode***)** - directed ascendant of *Inode*, (set to 0 for the root node)
list_starting_nodes -
List_Selected_leaves - list of selected leaves nodes, initialized to $\oslash$.

for each non-visited *Inode* from starting_nodes **do**
  mark *Inode* as visited
  Fnode $=$ father(*Inode*)
  while *Fnode* non-visited and *Fnode* $\neq 0$ **do**
    mark *Fnode* as visited
    increment nb_sons(*Fnode*)
    Fnode $=$ father(*Fnode*)
  end while
  if (*Fnode* $\neq 0$) increment nb_sons(*Fnode*)
end for
for *Inode* $\in$ starting nodes **do**
  If (*nb_sons(Inode)* $= 0$) add *Inode* to the List_Selected_leaves
end for

The output of the algorithm of branch detection is then used during the solution phase as input to the forward and backward substitution steps.

9.2.2 Subtree detection

The 'subtree detection' algorithm is used only during the backward step for reducible matrices or for null-space computations. We recall that the main objective of this algorithm is to detect for each target node, the complete subtree rooted at that node. We first describe the main feature of the subtree detection algorithm.

As in the previous case, we start by marking all nodes as non-visited. Then we get the list of starting_nodes, associated with nonzero entries in the right-hand sides. We select the subtree rooted at each node of starting_nodes. Note that we go through the elimination tree at most once to select all the subtrees.

Algorithm 9.3 : subtree detection

```
for each non-visited node in the list of starting_nodes do
   mark it and mark all nodes in its subtree down to leaf nodes or to next visited node
end for
```

A detailed subtree detection mechanism is described in Algorithm 9.4.

Initially, the list of selected leaves is empty, and subtree detection begins from one of the starting_nodes. At this stage, potentially each node in the starting_nodes set can be a root. A node is marked as not_root_node, if it is a son of another requested node. We traverse the elimination tree only once in a top-down traversal. For each not visited node in the starting_nodes, its whole subtree is marked. If an already marked son is encountered then we can skip to the next son or to another entry in the starting_nodes.

Algorithm 9.4 : detailed subtree detection

```
Input: ELIMINATION_TREE, STARTING_NODES.
Output: SELECTED_TREE (list_selected_leaves, list_selected_roots)
Local : List_Selected_nodes - to manage local subtree traversal

for each non-visited node Inode in the list of starting_nodes do
   if Inode is visited then cycle
   mark Inode as 'root_node'
   add Inode into List_Selected_nodes

   Process subtree rooted at Inode:
   for  each node, Idescendant, in List_Selected_nodes do
      for  each son, Ison, of Idescendant  do
         if  Ison is non-visited then
            mark Ison as visited
            add Ison to List_Selected_nodes
         else
            mark Ison as 'not_root_node'
         end if
         if (last_son of Idescendant) then remove Idescendant from List_Selected_nodes
      end for
   end for

end for

for each node Inode in the list of starting_nodes and marked as 'root_node' do
   add Inode into List_selected_roots
end for
```

During subtree detection the number of sons for selected nodes remains unchanged.

The total number of selected leaf nodes is also needed to terminate the backward step. For clarity, our algorithm computing the list of selected leaf nodes was not included in Algorithm 9.4 but this data can easily be updated each time in the algorithm we encounter a node with no son.

9.3 Topologically-based permutations

We have shown that the sparsity of the right-hand sides can be exploited to reduce both the amount of factors loaded in an out-of-core environment and the total number of operations during the solution phase.

Our objective in this section is to understand how one should permute the right-hand sides to better exploit their sparsity. As explained in Sections 8.2.2.1 and 8.2.3.1, for memory issues multiple right-hand sides are processed by blocks of fixed size N_s. The partitioning of the right-hand sides gives scope to further algorithmic improvement. For example one might want to group columns with as large as possible overlapping of the factors to be loaded.

We first recall properties of our applications with sparse multiple right-hand sides that can guide our choice of permutations.

Computing entries in A^{-1}. Forward and backward substitution steps exploit a branch detection algorithm to compute entries in each column of A^{-1}.

- The right-hand sides are partitioned into blocks by columns. Entries in the same column are thus always processed together (in a block). Entries in same rows can be processed in different blocks since their column indeces are different. We can influence only the grouping of the right-hand sides by ordering columns with similar properties in a single block.

- The column or row index of each entry in the right-hand sides is associated with a node in the elimination tree. To compute any inverse entry a_{ij}^{-1} we start with a forward substitution step following ascending path from node associated with column j to the root node, and continue with the backward substitution step following a descending path from the root to the node associated with row i.

- From the previous property it results that, if two columns of the right-hand sides have indices on the same path to the root are processed simultaneously. then factors will be loaded only once during the forward step.

  Example 9.1. *Let us suppose that we want to compute $a_{10,2}^{-1}$ and $a_{11,7}^{-1}$ (see Figure 9.1). These entries are associated with nodes 2 and 7 in the tree and share the path from node 7 (the ancestor of node 2) up to the root. Thus if columns 2 and 7 of the right-hand side vectors are grouped together, the block of factors associated with the nodes in the shared path will be accessed only once.*

- If two columns that are processed simultaneously, have only one row entry that is on the same path to the root then factors will be loaded only once during the backward step.

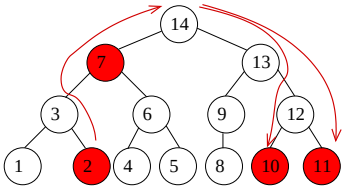


Figure 9.1: Common path in the forward substitution for entries $a^{-1}_{10,2}$ and $a^{-1}_{11,7}$

Example 9.2. *Let us suppose that we want to compute* $a^{-1}_{13,2}$ *and* $a^{-1}_{11,5}$ *(see Figure 9.2). For both entries the backward substitution will follow the same branch of the tree – from the root down to node* 13 *and to node* 11*. If the requested entries are processed in a single block, factors from the root node* 13 *will be loaded and accessed only once.*

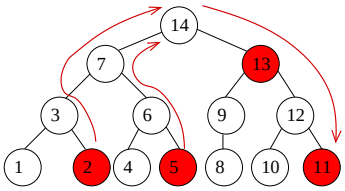


Figure 9.2: Common path in the backward substitution for entries $a^{-1}_{13,2}$ and $a^{-1}_{11,5}$

Null-Space Computations. Each null or quasi null pivot is associated with one column of the multiple right-hand sides. The backward substitution on each column will benefit from a subtree detection algorithm.

- A node includes in its subtree the subtree of all its descending nodes. Thus null pivots associated with nodes in the same subtree should be processed simultaneously to load only once factors that are common.
- In other words, two null pivots that are not associated to nodes on the same path to the root should be processed in different blocks.

  Example 9.3. *Let us suppose that we want to compute quasi null pivots associated with row indices* $3, 7, 12$ *and* 13 *(elimination tree shown in Figure 9.3). If pivots associated with nodes* 7 *and* 3 *are processed simultaneously, the factors in the subtree rooted at node* 7 *will be loaded only once for computations of both pivots. Nodes* 7 *and* 13*, are not on the same path to the root, thus, there is no benefit to process them together.*

In the context of a large number of right-hand sides and large matrices, we are interested in finding algorithms to permute and group the right-hand sides using some global information/property. From the previous observations it results that in both applications (computing A^{-1} and null-space basis), the branch detection and subtree detection algorithms can exploit the topology of the tree to group columns of the right-hand sides.

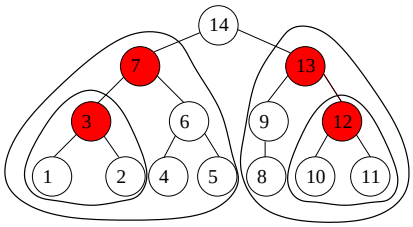


Figure 9.3: Illustration of subtrees, rooted on filled nodes.

With a post-order traversal of the tree all nodes in a subtree will have consecutive numbers. For null-space computations it is clear that permuting the null-pivots to follow a post-order of the tree will enable a simple splitting of the reordered list of right-hand sides to be efficient. For computing entries in A^{-1}, if one considers the special and interesting case of computing only diagonal entries of A^{-1} (see Section 1.2), then again permuting columns of the right-hand sides to follow a post-order will provide a good locality for factor reuse between consecutive columns of the right-hand sides.

9.3.1 Post-order permutation of the right-hand sides

We first describe post-order based permutation of the right-hand side vectors. In a sequential environment, a post-order traversal of the tree is used during both factorization and solution phases. Therefore, with a post-order permutation of the columns of right-hand sides, we can expect good overlapping of factors to be loaded for two consecutive columns. For entries in A^{-1}, permuting the right-hand sides such that the associated columns of A^{-1} follow a post-order ensures good locality for consecutive right-hand sides as shown in Figure 9.4. For null-space basis computations, processing null pivots with a post-order also gives scope to good locality within the subtrees.

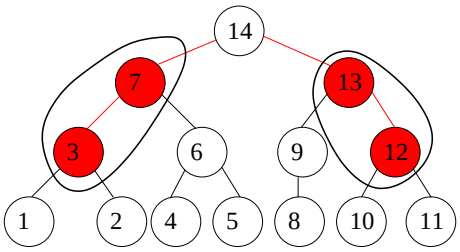


Figure 9.4: Ancestor nodes selected to be processed in a same block

A post-ordering based permutation of the columns of the right-hand sides provides good reuse of the factor between consecutive columns and can thus be use to drive a simple blocked processing.

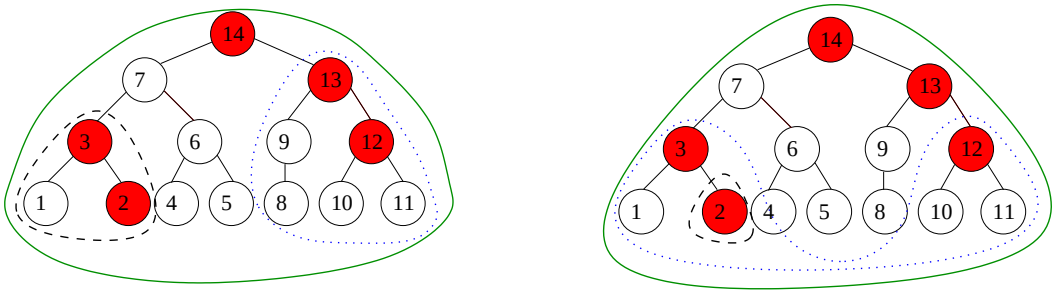
9.3.2 Pre-order permutation of the right-hand sides

We use the term pre-order to define the inverse of the post-order permutation. If the size of the right-hand sides is a multiple of N_s (the size of the block) then there is no difference between both permutations. When the size of the right-hand sides is not multiple of N_s, the last block might contain less columns compared to the other blocks.

If post-order permutation is used, the last right-hand side columns correspond to nodes in the upper part of the tree (close to the root node). For entries in A^{-1}, these last nodes will select shorter ascending paths in the elimination tree. For quasi null pivots associated with nodes from the upper part in the tree, larger subtrees will correspond to the last block. In this case, we may want to have as large as possible the last block of right-hand sides to better exploit the large amount of factors already loaded.

Example 9.4. *Let us suppose that we want to compute null-space vectors associated with null pivot rows* $2, 3, 12, 13$ *and* 14 *(see Figure 9.5). Let the block size be* 2*, that is* 2 *right-hand sides can be processed simultaneously. If post-order is used, the pivots will be processed as shown in Figure 9.5 a) in the order* $\langle 2, 3\rangle$ *then* $\langle 12, 13\rangle$ *and finally* $\langle 14\rangle$*. Note that in this case, the root node for which the whole tree is needed, is not associated with any other node. If a pre-order is used as shown in Figure 9.5 b) the pivots will be processed in the order:* $\langle 14, 13\rangle$ *then* $\langle 12, 3\rangle$*, and finally* $\langle 2\rangle$*. When using a pre-order, the root node is associated with another node from the tree while the last block to process has less factors to be accessed (since it corresponds to a node lower in the tree).*

However note that in this case it would be even better to process 12 *alone and* $\langle 2, 3\rangle$*.*



a) Post-order processing

b) Pre-order processing

Figure 9.5: Null pivots processed in blocks of size 2 at a time. Filled nodes represent the requested pivots. Subtrees associated with the grouped nodes with respect to the ordering are shown by lines.

As shown in Example 9.4 pre-order can reduce the amount of factors loaded with respect to post-order but is however not an optimal solution. In a sequential environment, we want within each block of right-hand sides to have a maximum overlapping between the factors to be loaded and thus to have a large overlapping of the node traversal between columns belonging to the same block. We have proposed topological orderings of the columns for which one can expect good overlapping of the factors to be loaded between consecutive columns. With a regular natural splitting of the reordered columns in blocks of fixed size N_s one can thus expect to limit the amount of factors loaded.

9.4 Permuting columns of the right-hand sides to address parallelism

In a parallel environment we now also want within each block of right-hand sides to give work to each processor, since each processor may have parallel access to its local disk on which factors have been written. We do not propose in this section a sophisticated solution/heuristic to this combinatorial problem. Instead we only describe an extension/generalization of our sequential algorithm to address parallelism.

We recall that, thanks to the subtree to subcube mapping [57] performed during analysis and exploited during factorization, nodes in the lower part of the tree are often mapped on a single processor. Nodes in upper part of the tree are most frequently of type 2 and thus are mapped onto more than one processor. For nodes in the upper part of the tree almost all processors can be expected to work. When right-hand side vectors associated with nodes in the lower parts of the elimination tree are selected, only one processor is thus active. This observation has motivated the idea of interleaving nodes mapped in a single subtree with nodes mapped on other subtrees. We thus interleave nodes in the lower parts of the elimination tree, mapped on different processors into the same block of right-hand sides (see Algorithm 9.5).

Algorithm 9.5 : Interleave local lists of ordered nodes on Nprocs processors

```
Input: LocalLists(Nprocs)   // local lists of ordered nodes
Output: GlobalList   // global ordered list of nodes, initialized to zero
Local: CurLocPos (Nprocs)   // position in local lists (initialized to start of list)
       CurProc   // current processor (initialized to 1)

while (Size(GlobalList) < total number of right-hand-sides) do
  if (CurLocPos(CurProc) < Size(LocalList(CurProc)) then
    insert current node from LocalList(CurProc) into GlobalList
    CurLocPos(CurProc)=next element in LocalList(CurProc)
    if (the node associated with CurLocPos(CurProc) is of type 1) then
      // type 1 nodes are mapped on one processor:
      change CurProc
    else
      // type 2 node are mapped on more than one processor: do no change CurProc
    end if
  else
    // no elements left in LocalList(CurProc)
    change CurProc
  end if
end while
```

The input of Algorithm 9.5 is a local ordering of the right-hand sides per processor. A simple way to build such a local ordering is to first perform a global reordering of the right-hand sides as proposed in the previous section. Then the mapping of the nodes onto the processor can be used to obtain an ordered local list of nodes per processor. In practice the situation is slightly more complex since each column of the right-hand sides is associated with nodes in the elimination tree which are mapped either on one process (master process of a type 1 node) or to a subset of processes with a master process and slave processes (type 2 node). To compute the local ordered lists of columns per processor, we only rely on the master processor mapping of the columns. Furthermore type 2 nodes are then processed differently during Algorithm 9.5 since the associated factors nodes are distributed among many processors.

This parallel extension of a global ordering works with any initial global ordering, including the topological orderings described in the previous section as well as with the hypergraph orderings presented in the next chapter.

Chapter 10

Hypergraph models to exploit the sparsity

Recall that we are solving $AX = B$ in an out-of-core context, where B represents a set of sparse right-hand side vectors. For memory issues, the right-hand side vectors are processed by blocks. In other words, the solution phase will process the right-hand sides in blocks. Within a block, the LU factors are accessed only once for all computations regarding the right-hand sides-vectors of the block. Therefore, our aim is to find a partitioning or a blocking of the right-hand sides vectors with similar computational requirements so as to reduce the cost of loading the factors. We show that the partitioning problem can be cast as a hypergraph partitioning problem.

Hypergraph partitioning was first used in VLSI (Very-Large-Scale Integration) design [109]. Later, it found applications in parallel computing, starting from [26, 28, 27, 115, 116] where hypergraph partitioning models are used for efficient parallelization of matrix-vector multiplies. Different hypergraph partitioning models are used in parallelizing scientific computing applications such as computation of response time densities in large Markov models [41], restoration of blurred images [117], and integer factorization in the number field sieve algorithm in cryptology [19]. A common setting in these applications is to model a given matrix with a hypergraph model. There are other parallel and distributed computing applications where hypergraph models are used, for example, workload partitioning in data aggregation [30], image-space-parallel direct volume rendering [24], data declustering for multi-disk databases [87, 91], and scheduling file-sharing tasks in heterogeneous master-slave computing environments [84, 85, 86]. We note also that hypergraph partitioning finds applications outside of the parallel computing domain: road network clustering for efficient query processing [36, 37], pattern-based data clustering [100], reducing software development and maintenance costs [18], topic identification in text databases [32], and processing spatial join operations [110].

In this chapter, we present an alternative method to partition the right-hand side vectors using hypergraph models. We represent certain parts of the elimination tree with nets and associate costs with those nets based on the loading cost of factors associated with those parts. Our aim is to reduce the cost of loading the factors. After a short review of hypergraphs and hypergraph partitioning in Section 10.1, two hypergraph models will be described: a model for enabling efficient computation of the diagonal entries of A^{-1} in Section 10.2, and another one for efficient computation of null-space vectors in Section 10.3. The two models differ in the way that the pins of the nets and the cost of

the paths in the tree are defined. Both hypergraph models have very particular structure where nets (see definition in Section 10.1) are defined as subsets of other nets and the cost of a net is strongly related to the cost of the nets in the subset. Finally, the hypergraph partitioning problem minimises the cost associated with paths in the elimination tree by grouping right-hand sides vectors with similar properties in a single partition/block.

10.1 Introduction

As introduced in the global introduction, a hypergraph is defined as a set of vertices V and a set of nets N. Each net, n_i, is a subset of vertices and has a size $|n_i|$ equal to the number of its vertices. Weights are associated with the vertices, and costs $c(i)$ are associated with the nets. The vertex set can be partitioned into s nonempty parts where the union of all parts gives V. We define Π to be such a partition, $\Pi = \{V_1, \ldots, V_s\}$. A net is said to be connected to a part V_i in Π if it has at least one vertex in that part. Thus the connectivity $\lambda(i)$ of the net n_i is the number of parts connected by n_i (see the general introduction for more details and an example on hypergraph partitioning).

In the hypergraph partitioning problem, the objective is to minimize the cut-size of the vertex partition Π.

$$cutsize(\Pi) = \sum_{n_i \in N} c(i)(\lambda(i) - 1) \; . \tag{10.1}$$

We define the following hypergraph $H = (V, N)$. The vertex set V is equal to the set of nodes associated with each column b_i of right-hand sides. (All requested columns for inverse entries in A^{-1} or all requested columns for null pivots to compute a null-space basis of A.) Each net n_i in N corresponds to a path in the elimination tree. For later use we note that these paths do not necessarily terminate at the root. We define as $P(i, j)$ the set of vertices on the path from node i to node j in the elimination tree. For simplicity, if a path starts from node i and goes up to the root we will denote it as $P(i)$. We define $Cost(P(i))$ to be the cost of loading factors from node i to the root. To describe the loading cost between two nodes, $Cost(i, j)$ will be used, corresponding to the sum of the weights of all nodes in the path $P(i, j)$ from node i to node j, without including node j:

$$Cost(i, j) = \sum_{k \in P(i,j),\ k \neq j} w(k) \; .$$

The cost $c(i)$ of a net n_i corresponding to the node i in the elimination tree will be defined using the function $Cost(i, j)$ defined above for some special j (we will define the appropriate j later). After defining the costs and the structure of the hypergraph, we will establish an exact correspondence between the cutsize function (Equation 10.1) and the amount of the factors to be loaded.

10.2 Model for entries in A^{-1}

Recall that to compute entries in A^{-1} we follow paths in the elimination tree in both substitutions phases. Consider the first node f that resides in the intersection of paths

$P(i)$ and $P(j)$ from nodes i and j to the root. Suppose that the solves associated with nodes i and j are performed in two different blocks. Then, the factors of nodes residing in the path $P(f)$ have to be loaded twice. In general, if node f is performed in k blocks, then the factors in $P(f)$ have to be loaded k times. Since we have to load those at least once, the overhead is $Cost(P(f))\,(k-1)$. In the remainder of this chapter we will show that, if we minimize this overhead, we minimize the cut-size of equation (10.1).

Let $H = (V, N)$ be a hypergraph where the vertex set V is equal to the set of all nodes associated with each column of the right-hand sides where at least one inverse entry is requested. Each vertex v_i has unit weight. We define two types of nets. The first set N_1 contains a net n_i for each vector b_i in the set of all right-hand sides RHS, i.e. $N_1 = \{n_i : b_i \in RHS\}$. Each net $n_i \in N_1$ contains only a single vertex v_i corresponding to b_i. We define **node $\mathbf{f_i}$** to be the first ancestor node of a requested node i in the intersection of the path $P(i)$ with a path from another requested node (can be defined as the least common ancestor [112] or the lowest ancestor [5] of node i with another node).

The set N_2 contains a net for each node f_i in the intersection of a number of paths to the root. Each net $n_j \in N_2$ is the union of any net n_k where node j is ancestor of node k in the elimination tree. Note that a node can be requested and also can be at the intersection of paths to the root. In such cases, a net of type N_1 and another of type N_2 (can happen, for example, when the node and a descendant of it correspond to requested entries) are associated with the same node. Since the nets of the set N_1 each have a pin list of size one, they can never be in the cut. Therefore, after building the hypergraph, those nets can be deleted from the model for simplicity.

Example 10.1. *In Figure 10.1 the filled nodes are associated with requested columns in right-hand sides. The set N_1 corresponds to nets associated with each requested node. Nets n_3, n_4 and n_{14} are defined as nets in N_1, each net contains only a single vertex (respectively v_3, v_4 and v_{14}). Net $n_7 \in N_2$ is associated with v_7, because it is in the*

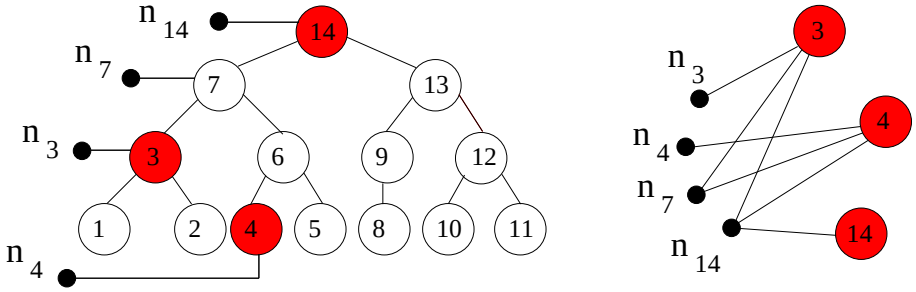


Figure 10.1: Nets associated with nodes in the elimination tree and corresponding hypergraph

intersection of paths $P(3)$ and $P(4)$. Then n_7 is the union of nodes 3 and 4, that is $n_7 = \{3, 4\}$. Finally net n_{14} is associated with a requested node and is also an intersection point, then $n_7 \subset n_{14}$.

The **cost of a net** is defined according to paths or parts of a path. As seen above, each net corresponds to a path that starts at a particular node of the elimination tree. The cost of a net $c(n_i)$ is defined as the weighted sum of the size of the nodes ($w(i)$) in the path $P(i, f)$ from the net n_i associated with node i to the net associated with the node f_i.

This cost is also noted as $Cost(i, f_i)$:

$$c(n_i) = Cost(i, f_i) = \sum_{k \in P(i,f),\ k \neq f} w(k) \tag{10.2}$$

Our hypergraph partitioning problem consists in finding partitions such that the cutsize metric of equation (10.1) is minimized:

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

where $\lambda(i)$ is defined as the connectivity of a net n_i. Minimizing this equation is equivalent to minimizing the cost of loading factors while solving for at most N_s right-hand side vectors in each block.

Example 10.2. *We give an example of the associated cost* $c(n_i)$ *on nets associated with nodes in the elimination tree as shown in Figure 10.2.*
$c(n_3) = Cost(3, 7) = w(3)$ *, because there is only* v_3 *in* $P(3, 7)$ *to the next net* n_7 *.*
$c(n_4) = Cost(4, 7) = w(4) + w(6)$
$c(n_7) = Cost(7, 14) = w(7)$
$c(n_{14}) = Cost(14) = w(14)$

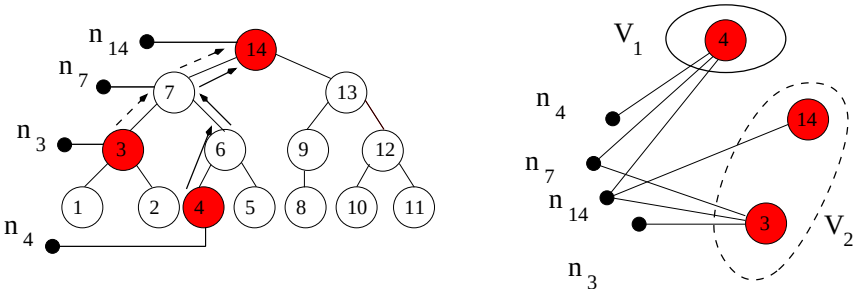


Figure 10.2: LEFT: Nets associated with the nodes in the elimination tree. Filled nodes correspond to requested entries. RIGHT: Corresponding hypergraph model, containing 4 nets and 3 vertices. The hypergraph is partitioned into 2 parts, represented by ellipses.

Our hypergraph model in Figure 10.2 is partitioned into two parts. The connectivity of each net is respectively: $\lambda(4) = 1$ *, because* n_4 *is connected to only one part;* $\lambda(7) = 2$ *, because* n_7 *is connected to both parts;* $\lambda(14) = 2$ *;* $\lambda(3) = 1$ *.*

Each part represent a path in the elimination tree. The corresponding paths are shown with line or cut-line arrows that show the ascending path to the root for each part. Following the paths, data to be loaded for the first part, associated with node 4 *is:*

$$w(4) + w(6) + w(7) + w(14).$$

Data needed to be loaded for the second part are:

$$w(3) + \mathbf{w(7) + w(14)},$$

where the bold data correspond to repetitive load of already loaded data. Knowing that in any case all the necessary data is loaded at least once, that is:

$$w(3) + w(4) + w(6) + w(7) + w(14),$$

thus the overhead of loading data with partitioning the hypergraph model in two parts is equal to:

$$w(7) + w(14).$$

Note that the overhead of loading data, correspond to the specific partition of the hypergraph model, and it could be easily computed by the formula:

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

Thus the overhead of data load is directly computed by the cutsize of our hypergraph: $cutsize(\Pi) = c(n_7)\ (\lambda(7) - 1) + c(n_{14})\ (\lambda(14) - 1) + 0\ c(n_3) + 0\ c(n_4) = w(14) + w(7)$

Nets will be associated with all columns in the right-hand sides, even if some columns are amalgamated in an unique node. In this case, we consider one of the amalgamated variables as the principal variable, and the others as secondary variables. The weight of the nets associated with all secondary variables in amalgamated nodes is zero, since there is no node between these kind of nets. Since these kind of nets do not add to the cutsize we drop them from the model.

Example 10.3. *Let us change our example by introducing node* 15 *amalgamated to node* 3*, as shown in Figure 10.3. Nets* 3 *and* 15 *are thus associated with node* $3, 15$ *in the assembly tree. Let* 3 *be the primary variable and* 15 *be the secondary variable in the amalgamated node. The cost of the secondary variable* 15 *is then* $c(15) = Cost(15, 3) = 0$*. The cost of net* 3 *has not been changed:* $c(3) = Cost(3, 7) = w(3)$*.*

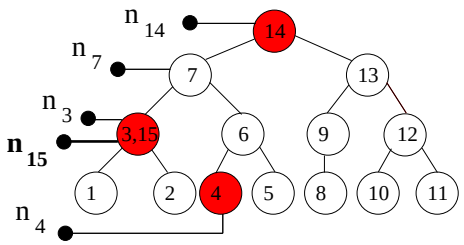


Figure 10.3: Net n_{15} associated with amalgamated node 15 in the tree

Remarks on the proposed model: Our hypergraph model for entries in A^{-1} has some properties which should be noted. First, for nets associated with ancestor nodes in the elimination tree (as nodes 3 and 7 in Figure 10.3), if the net associated with the ancestor node is cut into several blocks, then all its descendants will be cut too (for example: if net n_7 is cut, then n_3 will also be cut). Second, this is a very special hypergraph, represented as intersecting paths of a tree. In partitioning such a hypergraph, one may develop specialized algorithms by taking advantage of the particular structure. We also recall the minimum bin packing problem [56], given a bin size N_s, and a set $U = \{u_1, u_2, \ldots, u_n\}$ of n non-negative/positive integers, partition U into s disjoint sets $U_1, U_2, \ldots, U_s$ such that the sum of the elements in each U_i is less than or equal to N_s. The objective is to minimize the number of sets, s, used. Since we have $u_1 = 1$, our problem is not exactly the bin packing problem; but any bottom up approach that decides to pack the children of a node has to solve this problem (when children have already been packed, the problem is exactly equivalent to the bin packing one). Finally,

the cost of the net takes into account this special structure of nets that are subsets of other nets ($n_3 \subset n_7 \subset n_{14}$). A single net thus contributes to the cost of almost all nets. Furthermore, the cost of each block will change with respect to the right-hand sides vectors included in it; and will also influence the cost of the other blocks.

Example 10.4. *In the Figure 10.3, if node* 4 *is not requested, net* n_4 *will not be associated and net* n_7 *will not be defined as an intersection between* n_3 *and* n_4 *. Thus* n_3 *will have a different cost:* $c^{'}(3) = Cost(3, 14) = w(3) + w(7)$ *.*

10.3 Model for null-space computations

We recall that to compute the null-space basis of a matrix, first all the null pivots are detected and are stored in a list during the factorization phase. At the beginning of the solution phase, the nodes associated with null pivots become **starting nodes** for our pruning algorithms. Then the subtrees rooted at the starting nodes are processed during the solution phase.

Let $H = (V, N)$ be our hypergraph. In our hypergraph model we use vertices to represent all nodes associated with requested vectors of the null-space basis. As in the previous hypergraph model, the weights ($|v_i| = 1$). A net is associated with any node (starting node) in the tree associated with a null pivot. We use nets to represent subtrees or parts of subtrees. The cost of a net is thus related to the subtree mapped onto the associated node.

We introduce the term **directly related nets** to refer to nets associated with nodes in the same branch of the tree, such that in the path between them there are no other nets. (Example: in Figure 10.4 the directly related nets are n_1 and n_3 ; n_3 and n_{14} ; n_4 and n_{14} . Nets n_1 and n_{14} are not directly related because n_3 is in the path from n_1 to n_{14}). Let $S(h)$ be the subtree rooted at node h . For our null-space model we define the cost of a net $c(n_h)$ to be the weighted sum of nodes rooted at the node h . If in the subtree there are nets directly related to net n_h their cost should be subtracted.

Example 10.5. *In Figure 10.4 the filled nodes are associated with null-space vectors. Nets are associated with each of these nodes:* n_1 *,* n_3 *,* n_4 *and* n_{14} *, The costs of the nets*

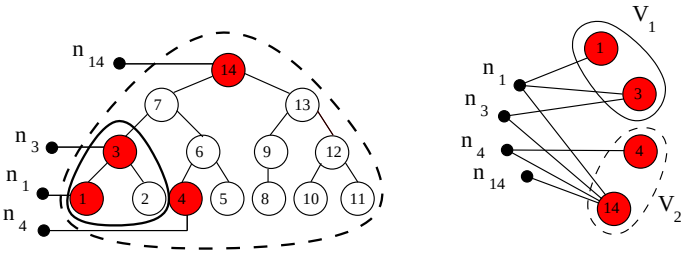


Figure 10.4: LEFT: Nets associated with nodes in the elimination tree. Filled nodes are associated with null-space vectors. RIGHT: Corresponding hypergraph model for null-space computations, containing 4 nets and 4 vertices.

$c(n_i)$ *are defined according to the subtrees rooted at each net and are as follows:*

$$c(n_1) = \sum_{k \in S(1)} w_k \qquad c(n_4) = \sum_{k \in S(4)} w_k$$
$$c(n_3) = \sum_{k \in S(3)} w_k \; - c(n_1) \qquad c(n_{14}) = \sum_{k \in S(14)} w_k \; - c(n_3) - c(n_4)$$

Note that the cost of n_3 includes the cost of net n_1, its directly related net. Thus the cost of n_1 is subtracted from the weighted sum of nodes rooted on node 3. Similarly, the cost of net 14 is the weighted sum of its subtree nodes with its directly connected nets n_3 and n_4.
Our hypergraph model in Figure 10.2 is partitioned into two parts: $V_1 = \{1, 3\}$ and $V_2 = \{4, 14\}$. The connectivity of each net with respect to these partitions is: $\lambda(1) = 2$, because n_1 is connected to both parts; $\lambda(3) = 2$; $\lambda(4) = 1$, because n_4 is connected to only to one part; $\lambda(14) = 1$. Each part represent (sub)tree in the elimination tree, shown by continuous and dotted line in the left part of Figure 10.2. The overhead of loading data with partitioning the hypergraph model in the corresponding parts is equal to: $cutsize(\Pi) = \sum_{n_i \in N} c(n_i)\ (\lambda(i) - 1) = c(n_1) + c(n_3)$.

Nets associated with amalgamated nodes in the elimination tree are treated as in the previous hypergraph model. Each net associated with a principal variable in the amalgamated node is the representative and its cost is computed as the cost of a ordinary net. Nets associated with secondary variables have zero cost and thus are dropped from the model.

Remarks on the proposed model: The hypergraph model for null-space basis computations also has a special structure. Nets associated with nodes in the upper levels of the tree are included in the subtree of nets associated with nodes in the lower levels. (Example: in Figure 10.4 $n_{14} \subset n_3 \subset n_1$.) If a node is cut into several blocks, the net associated with an ancestor node will also be cut (if n_1 is cut, then n_3 and n_{14} are also cut). The cost of the net takes into account the special structure in the same way as for the model of entries in A^{-1}. A single net contributes to the cost of most of the nets in the hypergraph. Then the cost of each block will change with respect to the right-hand sides vectors included in it; and will also influence the cost of the other blocks.

Example 10.6. *Consider that node 3 is not requested in the Figure 10.4. Then net n_3 will not be associated. Net n_1 will be defined as the first directly related net to n_{14}. Net n_{14} will have a different cost: $c(n_{14}) = \sum_{k \in S(14)} w_k \ - c(n_1) - c(n_4)$.*

10.4 Conclusions

When solving multiple right-hand sides by blocks, we try to order the right-hand sides efficiently to minimize the amount of data accessed. An alternative method for grouping the vectors is to use a hypergraph partitioning. We use the hypergraph modelling to describe paths and part of paths in the elimination tree. We then associate a cost for each of these parts with respect to the application (entries in A^{-1} or null-space basis computations). Minimizing the cutsize of each partition is equivalent to ordering efficiently the right-hand sides into blocks. The obtained hypergraph has a very special structure where each net contains other nets and thus the cutsize of each part is strongly related to the cost of almost all nets of the hypergraph model.

In many problems, the associated hypergraph is completely apparent. For example, in [29] a net corresponds to a row or column of the matrix, or to a file shared by a number of tasks [85]. In such cases, removing a single net from the hypergraph usually results in a hypergraph model of an apparently modified version of the input. For example, in the case of matrix partitioning for hypergraphs, removing a net results in the hypergraph

model of a matrix obtained from the original one by deleting a single row or column. Nothing else needs to be changed for solving the problem for this resulting matrix. In our case, however, the cost and the connectivity of the nets depends on other nets. One cannot always delete a net and obtain in a simple way a modified hypergraph model. It is also necessary to modify the costs and the connectivity of the nets.

Chapter 11

Results and performance analysis

11.1 Introduction

We have described in the previous sections, algorithms to exploit the sparsity of the right-hand side vectors and algorithms to permute multiple right-hand sides in order to process them efficiently. We have also proposed an adaptation of our algorithm to address parallelism and improve work balancing among the processors (see Algorithm 9.5).

In this chapter we comment on the results obtained in terms of amount of factors accessed and the computing time during the solution phase. We first present results for null-space basis computations in Section 11.2. We then continue with the analysis of the cost of computing multiple entries in A^{-1} in Section 11.3. Our set of test matrices, described in Section 1.3, corresponds to applications in electromagnetism (for null-space computations) and to applications in astrophysics (for entries in A^{-1}). All matrices are symmetric and are available on the `gridtlse.org` web site.

11.2 Null-space computations

We analyse the volume of factors accessed to compute the null-space basis of highly deficient matrices. We compare the performance with and without exploiting sparsity of the right-hand sides. As expected, the amount of factors accessed strongly influences the computing time of the solution phase.

In a uni-processor environment, we analyse the behaviour of strategies split the right-hand sides into blocks. In a parallel environment, our objective is also to give work to all processors and we report very preliminary results with our interleaving strategy.

11.2.1 Sequential execution

We first present results on a small matrix for null-space computation – *Boxcave_8x5x3*, of order 619 with $3\,471$ nonzeros. The deficiency of the matrix is 56 (the number of right-hand sides is thus equal to 56), and the size of the factors is $0.144\,MB$.

We first compare the amount of factors loaded in a sequential environment without exploiting the sparsity of the right-hand sides for different block sizes (see Table 11.1 line *no ES*). Let Nb be the size of the block and Nb_Blocks be the number of blocks

($Nb_Blocks = Def/Nb$). Results clearly illustrate the fact that the amount of data loaded without exploiting the sparsity is $|U| \times Nb_Blocks$. When processing all right-hand sides with a single block (i.e. $Nb_Blocks = 1$) all factors are loaded only once (See column $Nb = 56$ in Table 11.1). On the other hand, when solving one right-hand side at a time, the amount of loaded factors is equal to $|U| \times 56$ ($8.088\ MB$).

	Solve a block of right-hand sides (RHS) at once			
Total Factors Loaded	Nb = 1 RHS	Nb = 10 RHS	Nb = 16 RHS	Nb = 56 RHS
no ES [MB]	8. 088	0.866	0.577	0.144
with ES [MB]	**4. 276**	**0.554**	**0.409**	**0.144**

Table 11.1: Comparing the total size of factors loaded when exploiting the sparsity (with ES) and without exploiting the sparsity (no ES) of the right-hand sides on Box-cave_8x5x3. The right-hand sides are solved by blocks of size Nb.

Line **with ES** of Table 11.1 shows results obtained when the sparsity of the right-hand sides is exploited. When all right-hand sides are solved at once, there is no difference in the amount of data accessed. This is because some of the requested 'null pivots' are associated with the root node of the elimination tree so that the complete $|U|$ factors must be loaded at least once even when sparsity is exploited. Note that, on larger problems (see next section), memory problems may occur when trying to solve all right-hand sides at once. If we process the right-hand sides one by one, then the total amount of the loaded factors decreases by a factor of two. However, the total amount of factors accessed remains important ($4MB$) compared to the case where the right-hand sides are processed by blocks.

To simplify our study we have decided in the remainder of this section to fix the number of right-hand sides in a block to 16 per processor.

On our small problem, we analyse in Table 11.2, the influence of the permutation, post-order, pre-order and hypergraph (HG) model, on the amount of factors loaded. We also report the amount of factors loaded without exploiting the sparsity (column 'no ES'). We also indicate (in column 'Min') the minimum amount of factors based on the formula given in Property 8.14 :

$$min_size = \sum (size(node) * nb_requests(node)/s)$$

where $nb_requests$ is the number of requests for each node associated with an inverse entry and $size$ is its size.

Both orderings combined with sparsity exploitation reduce the total size of factors to be loaded. The pre-ordering gives better reuse of data than the post-ordering. Hypergraph permutation also shows a competitive behaviour.

Matrix name	Deficiency	Total factors loaded [MB]				
			no ES	**with ES**		
		min		post-order	pre-order	HG
Box-cave_8x5x3	56	0.337	0.577	0.408	0.337	0.354

Table 11.2: Comparison of the total factors loaded with and without exploiting the sparsity and using post-order, pre-order and hypergraph (HG) modelling to permute the right-hand sides and then process them by blocks of size 16.

In Table 11.3, we report the same statistics on our larger problems from electromagnetism. For each matrix we recall its order (in column 2), the number of nonzero entries (column 'NZ') and the deficiency (column 'Defic'). We recall that the deficiency corresponds to the total number of right-hand side vectors to solve.

Matrix name	Order	NZ	Defic	Total factors loaded [MB]				
				Min	no ES	**with ES**		
						Po	Pr	HG
Box-cave_16x10x3	2 675	15 953	270	7	19	8	8	8
Box-cave_20x13x3	4 419	26 129	456	18	64	24	21	19
Box-cave_30x20x4	14 454	89 185	1 653	170	1 261	208	201	196
Box-cave_40x27x5	33 627	212 883	4 056	803	9 622	901	884	998

Table 11.3: Comparison of the total factors loaded with and without exploiting the sparsity and using post-order, pre-order and hypergraph based ordering. The right-hand sides are processed by blocks of size 16.

We first see that the gain that results from exploiting sparsity increases with the size of the problem. We also observe that the larger the deficiency, the bigger the difference in the amount of data accessed depending on sparsity exploitation. With all matrices, pre-order loads less factors than post-order. Hypergraph modelling provides a competitive ordering on the medium size matrices. On our largest matrix, the hypergraph approach is less efficient than pre-order. In this case, following the topology of the tree naturally provides an efficient global ordering of the right-hand sides since the amount of data accessed is close to the minimum. Thus there is little scope for hypergraph based permutations to obtain improved performance.

Matrix name	Def.	Time [s]			
			ES		
		no_es	post-order	pre-order	HG
Box-cave_16x10x3	270	1.6	0.9	0.8	0.9
Box-cave_20x13x3	456	5.0	1.9	1.6	1.5
Box-cave_30x20x4	1 653	87.6	16.1	16.0	15.3
Box-cave_40x27x5	4 056	712.1	67.2	65.8	75.7

Table 11.4: Comparing the sequential time performance with and without exploiting the sparsity of the right-hand sides on larger matrices on null-space computations.

In Table 11.4, we analyse the influence of the permutation on the computing time in a sequential out-of-core environment. As expected, we obtain significant time reduction when exploiting sparsity of the right-hand sides. The larger the deficiency, the larger the gain - up to 11 times faster, obtained on matrix *Box-cave_40x27x5* using pre-ordering.

We also note that pre-ordering is slightly better than post-ordering. On medium size matrices, the hypergraph permutation gives the best performance, as it has less data to access during the solution step. On the largest matrix, pre-order gives the best performance. This confirms the fact that the time is strongly related to the amount of data loaded from disk.

11.2.2 Parallel execution

We now show the parallel performance on our two largest matrices: *Box-cave_30x20x4* and *Box-cave_40x27x5*. We compare the amount of factors accessed during the solution phase and the time for the solution phase.

We report in column ‘Max" the sum of the amount of factors loaded on the most loaded process during the processing of each block of right-hand sides. In column ‘Min", we give the same information as previously but for the least loaded process. Finally in column ‘Total", we report the total amount of factors loaded over all steps and processes. More formally :

$$
\begin{aligned}
Max &= \sum_{i=1}^{Nb_Blocks} max_{p \leq Np}(local_factors_loaded_p^i) \\
Min &= \sum_{i=1}^{Nb_Blocks} min_{p \leq Np}(local_factors_loaded_p^i) \\
Total &= \sum_{i=1}^{Nb_Blocks} \sum_{p=1}^{Np} (local_factors_loaded_p^i).
\end{aligned}
$$

We report in Table 11.5 and Table 11.6 a comparative study illustrating the interest of the interleaving described in Algorithm 9.5. Note that for our parallel experiments we force the number of right-hand sides ‘per processor’ to be equal to 16. Therefore the total number of right-hand sides in a block is equal to $16 \times Np$. Note that the results are given for two permutations : post-order (rows ‘Po’) and pre-order (rows ‘Pr’).

		No Interleaving				Interleaving			
		Factors loaded [MB]			Time	Factors loaded [MB]			Time
Np	Strat	Max	Min	Total	[s]	Max	Min	Total	[s]
1	Po	208	208	208	14,8	-	-	-	-
	Pr	201	201	201	14,2	-	-	-	-
2	Po	70	40	111	8,1	82	59	141	9,7
	Pr	73	42	116	8,8	99	73	172	11,2
4	Po	40	15	98	5,8	44	27	143	6,3
	Pr	40	16	97	4,8	46	20	126	5,2

Table 11.5: Parallel execution: comparison of factors loaded and time performance on matrix *Box-cave_30x20x4*. The number of right-hand sides per block is set to $Np \times 16$.

As expected we observe that our parallel interleaving algorithm orders the right-hand sides in such a way that the total amount of the factors accessed is always greater than the amount of factors accessed without interleaving. This comes from the fact that interleaving is designed to distribute work on all processors. Therefore, for a fixed size of block of right-hand sides we will globally have less overlapping of factors between right-hand side columns. Let us illustrate this on two processors in a worst case. In the worst case, with interleaving, the total amount of factors loaded with 2 processors and a block size Nb is equal to the size of the factors loaded on one processor with a block size of $Nb/2$. We thus pay for an increase in the amount of factors loaded due to the relative decrease per processor of the block size. In practice we see, in Tables 11.5 and

11.6, that although we do have an increase in the total volume of factors loaded, it is less than a linear increase in the number of processors. Furthermore, for each processed block of right-hand sides the time for loading factors from disk is larger than the time to load factors on the most loaded processor. In this respect we also see that the interleaving algorithm behaves correctly. Thus there is scope for time reduction since parallel accesses to local disk should also improve the global I/O bandwidth.

		No Interleaving				Interleaving			
		Factors loaded [MB]			Time	Factors loaded [MB]			Time
Np	Strat	Max	Min	Total	[s]	Max	Min	Total	[s]
1	Po	897	897	897	67,2	-	-	-	-
	Pr	880	880	880	68,5	-	-	-	-
2	Po	374	143	517	34,7	402	239	641	36,0
	Pr	411	155	566	38.9	503	231	734	46,2
4	Po	172	45	380	21,5	200	87	522	23,7
	Pr	168	45	370	21,0	225	94	589	26,0
8	Po	81	16	323	17,6	123	42	706	21,7
	Pr	72	9	250	14,8	118	32	618	19,7

Table 11.6: Comparison of factors loaded on parallel execution with matrix *Box-cave_40x27x5*. The number of right-hand sides per block is set to $Np \times 16$

.

One can see that with both strategies (with or without interleaving) the time decreases with the number of processors. This is mostly due to the global increase of the block size equal to $Np \times 16$. However, on these preliminary results, we see that we do not benefit enough from parallel I/O access to the local disks with our interleaving approach to improve the performance. Some additional work is thus needed to understand the results provided in this preliminary work to address parallelism.

11.3 Computing elements in A^{-1}

We first analyse the behaviour of our algorithms (exploiting sparsity and ordering and blocking the columns of the right-hand sides in a sequential environment). We then comment on very preliminary results in a parallel environment. Most of our test matrices come from applications in astrophysics and are described in Section 1.3. Regular matrices corresponding to the discretization of the Laplacian operator are also used. As explained in Section 1.2 and 1.3, we suppose that only the diagonal element of A^{-1} need be computed and will also consider computing part of them. In all tables, matrices are ordered by increasing number of the right-hand sides (which is thus often equal to the order of the matrix).

11.3.1 Sequential execution

The number of right-hand sides can be very large since it is equal to the order of the matrix. In this case, the hypergraph models discussed in the previous chapter become prohibitively large in terms of net size to be useful. To compare the potential of the hypergraph modelling with the topological orderings, we will suppose that only 10% of the diagonal entries are requested (diagonal entries are selected randomly). We report in Table 11.7 the size of the factors to be loaded without exploiting sparsity (columns 'no es' and with exploiting sparsity (columns **'with ES'**). To our previous permutations (post-order (Po), pre-order (Pr) and hypergraph modelling (HG)) we add the natural ordering (column Nat in Table 11.7). We also indicate (in column 'Min'), the lower bound on the factors to be loaded based on the formula given in Property 8.14.

Matrix name	Nb RHS	Min	Factors to be loaded [MB]				
			no ES	**with ES**			
		[MB]		Nat	Po	Pr	HG
d11_20x12x5	120	2	8	5	2	2	2
a-1_08M	899	79	770	116	81	81	79
a-1_21M	2 153	864	7 694	2 800	873	882	872
a-1_46M	4 679	1 181	13 764	1 560	1 185	1 188	1 194
a-1_72M	7 235	212	40 664	682	236	235	212
a-1_148M	14 828	736	191 959	2 713	801	805	746

Table 11.7: Influence of column ordering on the amount of factor accessed in a sequential environment. Only 10% of the diagonal entries of A^{-1} are requested. The right-hand sides are processed by blocks of size 16.

We see in Table 11.7 that the larger the number of right-hand sides, the larger the difference in the amount of factors loaded when exploiting or not the sparsity of the right-hand sides (compare columns 'no es' and 'Nat'). When permuting the right-hand sides for better reuse of the loaded factors, all permutations (Po,Pr,HG) reduce the amount of factors to a value close to the minimum size (compare with column 'Min'). We note that using hypergraph ordering leads to the smallest amount of factors to be loaded in most cases. We finally observe that post-ordering and pre-ordering have a very similar behaviour and are also competitive permutations.

For applications in astrophysics and other least-squares data fitting problems, often all diagonal entries of A^{-1} need to be computed. We now focus on this application

in the remainder of this section and thus do not report results with hypergraph based permutations and focus on topological based permutations. In Table 11.8 we report the total amount of factors loaded when computing all diagonal entries in A^{-1}. We see that a larger number of right-hand sides (column 'order') does not always lead to a larger amount of data to load (see column 'Min'). As in the previous case, exploiting the sparsity of the right-hand sides with a natural ordering (column 'Nat') is less efficient than using topological permutations (columns 'Po' and 'Pr'). For example on matrix *a-1_72M* with post-ordering and the pre-ordering we reduce the amount of data accessed by a factor of 3. In general, both topological orderings have a very similar behaviour.

Matrix name	Order	Min [MB]	Total factors loaded [MB]			
			no ES	**with ES**		
				Nat	Po	Pr
d11_20x12x5	1 200	18	75	33	25	25
a-1_08M	8 999	714	7 609	931	790	786
a-1_21M	21 532	16 724	76 718	20 214	17 654	17 733
a-1_46M	46 799	11 105	137 407	12 165	11 628	11 629
a-1_72M	72 358	1 621	433 533	5 800	1 912	1 910
a-1_148M	148 286	9 227	1 677 479	18 143	9 450	9 461

Table 11.8: Influence of column ordering on the amount of factor accessed in a sequential environment. All diagonal entries of A^{-1} are requested. The right-hand sides are processed by blocks of size 16.

In Table 11.9 we report the computing time of the solution phase. Time is very much related to the total amount of factors loaded reported in Table 11.8. When sparsity is not exploited, we could not obtain a solution with our largest matrix *a-1_148M* in less than 24 hours. As one could expect from Table 11.8, we see that we obtain a large reduction in the solution time on matrix *a-1_72M* when permuting the right-hand sides. Topological orderings (post-ordering and pre-ordering) halve the solution time with respect to the natural ordering. However, the amount of loaded factors is not the only issue for performance since the number of requests in the emergency zone, and the regularity in the disk access can also influence the computing time. This is well illustrated by results on matrices *a-1_46M* and *a-1_72M*. On matrix *a-1_72M* we load $11\,628$ MB factors which is six times less than the factors of $1\,912$ MB with matrix *a-1_46M* and the computing time for the solution phase is only reduced by a factor of two (respectively 455.0 sec and 218.4 sec).

Matrix name	Order	Time performance [s]			
			with ES		
		no ES	Nat	Po	Pr
d11_20x12x5	1 200	4.4	1.4	1.1	1.2
a-1_08M	8 999	126.3	12.1	10.9	10.3
a-1_21M	21 532	738.0	270.5	256.7	233.4
a-1_46M	46 799	6 944.3	472.2	455.0	449.0
a-1_72M	72 358	27 728.1	408.2	218.4	213.6
a-1_148M	148 286	> 24h	1 391.6	986.2	996.8

Table 11.9: Influence of column ordering on the computing time in a sequential environment. All diagonal entries of A^{-1} are requested. The right-hand sides are processed by blocks of size 16.

In the following we discuss the influence of the block size on the performance of the solution phase (see Table 11.10 with results on matrix *a-1_72M* and Table 11.11 with

results on matrix *a-1_148M*). With both matrices, the amount of factors loaded decreases when we increase the number of right-hand sides per block. When the sparsity is not exploited, on matrix *a-1_72M* the amount of factors decreases from $433\,533$ MB using 16 right-hand sides per block to $6\,370$ MB using blocks of size $1\,000$; and on matrix *a-1_148M* the reduction is from more than $1\,677$ GB with 16 right-hand sides per block to 26.9 GB with blocks of size $1\,000$. The same behaviour is observed with exploiting sparsity. The amount of factors decreases by a factor of ten in the case of post-ordering and pre-ordering on matrix *a-1_148M*. We observed the same behaviour in the null-space case (see Table 11.1). The minimum size of factors loaded decreases by a factor of 15 in the case of matrix *a-1_72M* and by a factor of 29 in the case of matrix *a-1_148M*. Finally one should note that the amount of factors using topological permutations (Po and Pr) is much closer to the minimum size with a small number of right-hand sides per block. This probably also means that on a very large number of right-hand sides our lower bound is less accurate.

	Min	Total factors loaded [MB]				Time performance [s]			
	size	no ES	**with ES**			no ES	**with ES**		
NbRHS	[MB]		Nat	Po	Pr		Nat	Po	Pr
16	1 621	433 533	5 800	1 912	1 910	27 728.1	408.2	218.4	213.6
100	307	63 183	4 238	705	706	7 197.5	375.9	172.1	172.7
1 000	108	6 370	2 207	398	399	8 544.8	1 293.8	367.1	367.8

Table 11.10: Different sizes of the block applied on *a-1_72M* for requested all diagonal. entries in A^{-1}.

	Min	Total factors loaded [MB]				Time performance [s]			
	size	no ES	**with ES**			no ES	**with ES**		
NbRHS	[MB]		Nat	Po	Pr		Nat	Po	Pr
16	9 227	1 677 479	18 143	9 450	9 461	> 24h	1 391.6	986.2	996.8
100	1 572	297 616	9 733	2 335	2 343	32 758.5	1 167.9	680.6	682.6
1 000	317	26 969	7 328	927	928	43 940.9	5 193.7	1 321.5	1 328.2

Table 11.11: Different sizes of the block applied on *a-1_148M* for requested all diagonal. entries in A^{-1}.

11.3.2 Parallel execution and permutations

In this section, we report very preliminary results on an adaptation of our sequential algorithm to address parallelism. Our algorithm interleaves the permuted columns of the right-hand sides on all the processors (see Algorithm 9.5).

To simplify our study we analyse the behaviour of our algorithms on symmetric matrices, corresponding to a $3D$ 11point-discretization of Laplacian operator described in Table 11.12. We show the order, the number of nonzeros (NZ) and the factor size of each matrix using METIS and AMD reordering techniques.

As expected on very rectangular grids AMD is more efficient at reducing the factor size (compare columns 5 and 6 in row *Rect-25M*). Since the factor size is critical for OOC performance both orderings have been considered. In Table 11.13, one can see that the total volume of factors loaded without exploiting sparsity is only related to the factor size so that AMD ordering behaves better than METIS on *Rect-25M*. However since all paths to the root are on average significantly longer (6 times) with AMD than with METIS,

matrix	Grid	order	NZ	Factors [MB]	
				METIS	AMD
Rect-25M	500,10,5	25 000	249 720	15.7	10.1
Cub-25M	50,50,10	25 000	264 040	43.5	75.9

Table 11.12: Test matrices for study the parallel behaviour of the algorithm and factor size when different orderings are used.

the amount of factors loaded when exploiting the sparsity is much larger (3 times) with AMD than with METIS. We can also see the desastrious effect on performance when computing all diagonal entries of A^{-1} with AMD. Therefore, on both matrices, METIS is our best ordering in terms of both amount of factors to be loaded and computing time while exploiting sparsity. We will thus limit our analysis in the remaining of this section to the METIS ordering.

matrix	Ordering	Tree Depth	Total amount of factors loaded [MB]		Time with ES [s]
			Exploit Sparsity	Not Exploit Sparsity	
Rect-25M	AMD	11 127	9 426	24 505	1 080
	METIS	1 914	3 423	37 429	149
Cub-25M	AMD	2 649	64 002	198 017	3 817
	METIS	985	20 636	61 478	1 141

Table 11.13: Influence of the orderings on the tree structure, the amount of factors loaded and the run-time for computing all diagonal entries in A^{-1}. Statistics obtained on 1 processor, using post-ordering and 16 right-hand sides per block.

We first analyse in Table 11.14 the performance on our rectangular problem (matrix *Rect-25M*) for various block sizes and number of processors. We first note that the amount of factors loaded decreases when increasing the block size. We reduce by a factor of 2 the amount of factors loaded on one processor when increasing the block of size from 16 to 64 right-hand sides. This reduction of data accessed directly influences the time-performance so that with a block of size 64 the run-time is reduced by a factor of 1.5. We also note that when using an intermediate block size of 32 right-hand sides even if the amount of factors loaded is still relatively high, we have captured most of the benefits in terms of computing time (even in a parallel environment). Furthermore, note that, on a given tree, the total amount of factors loaded is by definition independant of the number of processors. In the MUMPS solver, the trees used for parallel execution and for sequential execution are not always identical. We see in Table 11.14 that this modification of the tree has a very strong and unexpected influence on the total amount of factors loaded between one and two processors. The tree between two and four processors is identical and we thus have constant results.

During the factorization phase (see 3.2.1) we exploit a subtree to subcube mapping of the elimination tree. Complete subtrees are thus mapped on the same processors. During the solution step, using a post-ordering of the columns of the right-hand sides, consecutive ones are likely to belong to the same subtree and thus mapped on the same processor. Furthermore, on this matrix, the top level separators obtained by METIS are very small so that matrices/nodes at the top of the tree are Type 1 nodes treated by only one processor. Therefore, without interleaving, most of the factors that need to be accessed to process a block of right-hand sides are mapped onto one and often the same processor. This leads to a strong imbalance of the amount of factor accessed without interleaving (compare

	No Interleaving					Interleaving			
	Size	Factors/proc loaded [MB]			Time	Factors/proc loaded [MB]			Time
Np	Block	Max	Min	Total	[s]	Max	Min	Total	[s]
1	16	3 423	3 423	3 423	148.7	–	–	–	–
	32	1 982	1 982	1 982	99.3	–	–	–	–
	64	1 261	1 261	1 261	98.8	–	–	–	–
2	16	1 353	413	1 768	87.8	1 483	1 102	2 585	91.2
	32	865	286	1 151	65.3	884	644	1 528	58.9
	64	608	235	844	61.9	600	446	1 046	57.4
4	16	926	97	1 768	82.3	1 201	668	3 722	89.3
	64	365	79	844	50.2	430	241	1 330	51.6

Table 11.14: Parallel behaviour of factors loaded and run-time on matrix *Rect-25M*. All diagonal entries in A^{-1} are computed exploiting the sparsity of the right-hand sides. Column of the right-hand sides are permuted with a post-ordering.

columns 'Max' and 'Min' in Table 11.14). With interleaving we see that we have a better equilibration between least and most loaded processors in terms of disk accesses. However, for a fixed value of the block size, the total amount of factors loaded is higher with interleaving than without. That is because at each step, the potential for overlapping factor accesses might have been reduced in the worse case by the number of processors. One can see in Table 11.14 (compare columns Total with and without interleaving) that we are far from this worse case. In terms of computing time, interleaving does not however lead to any significant performance decrease.

	No Interleaving					Interleaving			
	Size	Factors/proc loaded [MB]			Time	Factors/proc loaded [MB]			Time
Np	Block	Max	Min	Total	[s]	Max	Min	Total	[s]
1	16	20 636	20 636	20 636	1 140,8	–	–	–	–
	32	11 323	11 323	11 323	706,3	–	–	–	–
	50	7 752	7 752	7 752	484,8	–	–	–	–
	100	4 343	4 343	4 343	306,6	–	–	–	–
2	16	16 837	10 423	27 261	1 008,6	18 037	15 557	33 615	1 263,8
	32	8 722	5 927	11 323	565,2	9 320	7 941	17 261	686,6
	50	5 763	4 112	9 875	405,0	6 155	5 171	11 326	477,6
	100	3 081	2 327	5 408	246,3	3 295	2 687	5 982	278,4

Table 11.15: Parallel behaviour of factors loaded and run-time on matrix *Cub-25M*. All diagonal entries in A^{-1} are computed exploiting the sparsity of the right-hand sides. Column of the right-hand sides are permuted with a post-ordering.

We report in Table 11.15 results on the *Cub-25M* matrix. Even if *Cub-25M* and *Rect-25M* have the same size, *Cub-25M* has a significant larger tree with much larger frontal matrices (top level separators in METIS) than than *Rect-25M*. The relative gains due to increasing the block size are thus larger with *Cub-25M* than with *Rect-25M*. All nodes at the top of the tree are then processed in parallel (Type 2 and Type 3 nodes) so that in parallel the difference between the maximum and the minimum amount of factor loaded is smaller on matrix *Cub-25M* than on matrix *Rect-25M*. One can see in Table 11.15 that interleaving further improves the balance, the minimum and the maximum amount of factors loaded. However again this does not lead to any time reduction.

To conclude this section, we must insist on the fact that it is a preliminary study. We

have shown, that the shape of the tree very strongly influences the performance of the algorithm. On matrix *Rect-25M*, AMD was efficient at reducing the factor size but was very bad at exploiting sparsity of the right-hand sides because of the depth of the tree and the size of the upper-layers nodes. On both matrices the interleaving algorithm is successful at equilibrating the amount of factor loaded per processor but does not lead to any significant time improvement.

Chapter 12

General conclusion and future work

The context of our study is the solution of very large systems of linear equations with direct methods. With direct methods we have to store the matrices of factors ($A = LU$ or $A = LDL^T$) which are often significantly larger (ten to a hundred times larger) than our original matrix. The memory requirement of direct methods is thus a major limitation of the approach. One way to extend memory availability is to use parallel distributed memory computers. Another way to extend the main memory is to exploit the disk (so called out-of-core approach). In this work, we combine the two and so study parallel out-of-core methods. In this context the first difficulty is to efficiently store the matrices of factors on the local disk of the processors during the factorization phase. This work has been the main focus of E. Agullo's thesis at ENS-Lyon [1]. In an out-of-core context we have shown that the performance of the solution phase can be as time-consuming as the performance of the factorization phase. Furthermore when multiple right-hand sides are considered or for problems such as null space basis computation or computing entries in the inverse of a matrix then the cost of the solution phase can be even more critical. Our focus in this thesis has thus been the design of efficient algorithms for the solution phase ($LUX = B$ or $LDL^TX = B$ where X and B are matrices) assuming that the matrices of factors are distributed on the local disks of our parallel computers.

Out-of-core parallel solution phase

In the first part of this thesis we have described the parallel algorithms used during the solution phase and explained how they must be adapted to our parallel out-of-core context. This work was implemented within the parallel multifrontal solver MUMPS. A careful description of existing (i.e. incore) parallel solution phase algorithms has never been done before and so is also one contribution of this work.

During the solution phase, the amount of floating-point operations is in general three to four orders of magnitude smaller than for the factorization phase so that there is almost no scope to overlap disk access (I/O) with computations. In this context, the number and the regularity of the I/O has been shown to be very critical for the efficiency of the solution phase in both a sequential and a parallel environment. Even in the context of multiple right-hand sides, we have shown that for memory issues one must process the right-hand sides by blocks of reasonably small size. To process each block one may have to access all the factors so that even in this case the ratio of the volume of I/O over computation is still high.

We have compared in Chapter 5 two strategies for reading/writing data on disk (SYSTEM_BASED and DIRECT_IO). With a SYSTEM_BASED approach, although we get full benefit from the system cache mechanisms, we have shown that this strategy is not efficient when memory is limited. We have thus introduced DIRECT_IO access to the disk with small prefetch buffers and shown that it is more efficient than the SYSTEM_BASED approach both in terms of memory effectively used and computing time.

In a parallel environment, task scheduling can strongly influence the time performance of the solution. We have described in Chapter 6 a constrained scheduler that forces the solution phase to follow the write sequence of the factorization phase. We have proved the correctness of the algorithm and have reported results on a set of large problems to show the efficiency of our new scheduler.

Sparse multiple right-hand sides

In an out-of-core environment, what is most critical for the performance of the solution is the amount of factors loaded. When the right-hand side is sparse it has been shown in [62] and [64] that sparsity can be exploited to limit the amount of factors that need to be accessed.

Among many possible problems with multiple sparse right-hand sides, we have focused our attention on three of them (null space basis computation, computing entries in the inverse of a matrix, and sparse right-hand sides on reducible matrices) coming from applications in electromagnetism, astrophysics and linear programming.

We have first described and analysed in Chapter 8 different methods to compute the entries in the inverse of the matrix which preserve the sparsity of the computations. We have summarized the work based on the Takahashi equations and have compared the amount of factors to load using the Takahashi equations or with more a traditional solution.

We have then shown that each of our three problems on which we have focused requires sparsity to be exploited in a different way. For example to compute a null space basis we want to solve $UX = Y$ with a sparse matrix of right-hand sides Y, whereas to compute entries in the inverse of a matrix we must solve $LUX = B$ where B is a sparse matrix of right-hand sides and only few entries of the matrix X are requested. We have shown that, in all cases, exploiting the sparsity in one right-hand side can be seen as processing a pruned tree. In Chapter 9 we have shown that to cover all our problems two types of pruning must be introduced - chain detection and subtree detection. In a multiple right-hand side context, for memory issues, columns need to be processed by blocks. Columns “sharing” the same path in the pruned tree should then be grouped to reuse the data. We have identified properties on columns of the right-hand sides which share a common path in the forward or backward substitutions and proposed topologically based permutations to group similar right-hand sides in a single block. In Chapter 10 we proposed a hypergraph partitioning of the columns of the right-hand sides. We have used hypergraph models to describe paths and overlapping paths in the tree. Our models for each problem (type of pruning) differ in the way the nets and the cost of the nets are defined. The models obtained have a very particular structure of nets included in other nets. Thus partitioning the right-hand sides with respect to the cost associated to each net may become prohibitively expensive when increasing the number of right-hand sides (as in the case of computing all diagonal entries of A^{-1} where the number of right-hand

sides is thus equal to the order of the matrix). To address parallelism and benefit from parallel access to the local disks, we have proposed interleaving the permuted right-hand sides.

In Chapter 11 results on null-space basis computations and on computing diagonal entries in A^{-1} are reported. We have shown that exploiting sparsity of the right-hand sides leads to very significant gains in both time and factor accessed. Hypergraph based permutations are very competitive on medium size of matrices and in general all proposed permutations (topological or hypergraph) of the right-hand sides reduce the amount of data accessed to be close to the minimum. In parallel, our interleaving algorithm has a correct behaviour in terms increasing the volume of factors accessed in parallel; however more work is needed to analyse the computing time behaviour.

Perspectives

We have shown how critical a scheduler can be in a parallel out-of-core environment. We have proposed an efficient local scheduler to improve the performance of the solution phase. Even if significant gains have been obtained with the proposed scheduler, there is in some cases scope for improvement. In Section 6.2, we have explored a first track and have relaxed our scheduler to enable out-of-order task processing. To help its local decisions our scheduler could also be guided by a global strategy. On a large number of processors, scheduling the tasks of the slaves of Type 2 nodes can also be an issue. One could also influence the factorization phases or even the analysis phase to provide an elimination tree and/or a distribution of the factors onto the disks that is more suitable to our parallel out-of-core solution phase. This is possible during the factorization phase because we have quite some freedom to organize I/O operations in a way that is more suited to the solution step without affecting the performance significantly. This is in fact a more general remark. We have been working in the past mainly on algorithms (during analysis or during factorization) to improve the performance of the factorization. In many applications, as far as the factorisation is concerned, the most critical issue is the peak of memory used. Efficient parallel solution phases and analysis phases are then becoming the most critical issues.

When using sparse multiple right-hand sides, we have shown that exploiting the sparsity significantly improves the time-performance of the solution phase because in an out-of core context the volume of I/O can be significantly reduced. Our pruning of the tree will also lead to a reduction of the amount of operations and will thus also impact the in-core computing time of the solution phase. Although benefits can be expected using our current approach, our models have to be revisited in an in-core environment. In a parallel environment we have proposed a preliminary study to address parallelism. Combining the right-hand sides in parallel is a challenging combinatorial problem which should be further investigated.

Finally, another interesting possibility in an out-of-core context is to anticipate, when possible, the forward solution step. In this case one may want to even consider not writing the associated factors to disk. This is possible since the forward step processes the dependency tree in the same way as the factorization step. In this context, processing multiple sparse right-hand sides can be a complicated issue. One may then for example want to select the most time consuming right-hand sides as candidates for being processed during factorization. One may also want to exploit an a priori knowledge of the sparsity

of the right-hand sides and of the required entries in the solution to also limit the amount of factors that are stored on disk.

Bibliography

[1] E. Agullo. *On the Out-of-core Factorization of Large Sparse Matrices*. PhD thesis, École Normale Supérieure de Lyon, Nov. 2008.

[2] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. A preliminary out-of-core extension of a parallel multifrontal solver. In *EuroPar'06 Parallel Processing*, pages 1053–1063, 2006.

[3] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. A parallel out-of-core multifrontal method: Storage of factors on disk and analysis of models for an out-of-core active memory. *Parallel Computing, Special Issue on Parallel Matrix Algorithms*, 34(6-8):296–317, 2008.

[4] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1:131–137, 1972.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. *SIAM Journal on Computing*, 5(1):115–132, 1976.

[6] F. L. Alvarado and R. Betancourt. Parallel inversion of sparse matrices. *IEEE Transactions of Power Systems, PWRS*, 1:74–81, 1986.

[7] 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(1):15–41, 2001.

[8] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. MUMPS: A multifrontal massively parallel solver. *ERCIM News*, 50:14–15, July 2002. European Research Consortium for Informatics and Mathematics (ERCIM), http://www.ercim.org.

[9] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Parallélisation de la factorisation LU de matrices creuses non-symétriques pour des architectures à mémoire distribuée. *Calculateurs Parallèles Réseaux et Systèmes Répartis*, 10(5):509–520, 1998.

[10] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000.

[11] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. Technical Report RAL-TR-1999-059, Rutherford Appleton Laboratory, 1999. Revised version appeared in SIAM Journal on Matrix Analysis and Applications.

[12] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[13] P. R. Amestoy and C. Puglisi. An unsymmetrized multifrontal LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 24:553–569, 2002.

[14] C. Ashcraft. The fan-both family of column-based distributed Cholesky factorisation algorithms. In J. R. Gilbert and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 159–190. Springer-Verlag, NY, 1993.

[15] C. Ashcraft and R. G. Grimes. SPOOLES: An object oriented sparse matrix library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 22–24, 1999.

[16] J. K. Avila and J. A. Tomlin. Solution of very large least squares problems by nested dissection on a parallel processor. In *Proceedings of Computer Science and Statistics: Twelfth Annual Symposium on the Interface*, Waterloo, Canada, 1979. Department of Computer Science.

[17] C. Aykanat, A. Pınar, and Ü. V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. *SIAM Journal of Scientific Computing*, 25:1860–1879, 2004.

[18] R. Bisseling, J. Byrka, S. Cerav-Erbas, N. Gvozdenovic, M. Lorenz, R. Pendavingh, C. Reeves, M. Roger, and A. Verhoeven. Partitioning a call graph, 2005. Second International Workshop on Combinatorial Scientific Computing.

[19] R. Bisseling and I. Flesch. Mondriaan sparse matrix partitioning for attacking cryptosystems by a parallel block lanczos algorithm: a case study. *Parallel Computing*, 32:551–567, 2006.

[20] Å. Björck. Methods for sparse least squares problems. In J. R. Bunch and D. J. Rose, editors, *Sparse Matrix Computations*, pages 177–199, New York, 1976. Academic Press.

[21] Å. Björck. *Numerical methods for Least Squares Problems*. SIAM, Philadelphia, 1996.

[22] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM Press, 1997.

[23] L. Bouchet, J.P.Roques, P.Mandrou, A. Strong, R. Diehl, R. Lebrun, and R. Terrier. Integral spi observation of the galactic central radian: Contribution of discrete sources and implication for the diffuse emission. *Astrophysical Journal*, 635:1103–1115, 2005.

[24] B. B. Cambazoglu and C. Aykanat. Hypergraph-partitioning-based remapping models for image-space-parallel direct volume rendering of unstructured grids. *IEEE Transactions on Parallel and Distributed Systems*, 18:3–16, 2007.

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

[26] Ü. V. Çatalyürek and C. Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. *Lecture Notes in Computer Science*, 1117:75–86, 1996.

[27] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10:673–693, 1999.

[28] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10:673–693, 1999.

[29] Ü. V. Çatalyürek and C. Aykanat. PaToH: A multilevel hypergraph partitioning tool, version 3.0. Technical Report BU-CE-9915, Computer Engineering Department, Bilkent University, 1999.

[30] C. Chang, T. Kurc, A. Sussman, Ü. Çatalyürek, and J. Saltz. A hypergraph-based workload partitioning strategy for parallel data aggregation, 2001. SIAM Conference on Parallel Processing for Scientific Computing.

[31] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. *Computer Physics Communications*, 97:1–15, 1996.

[32] C. Clifton, R. Cooley, and J. Rennie. Topcat: Data mining for topic identification in a text corpus. *IEEE Transactions on Knowledge and Data Engineering*, 16:949–964, 2004.

[33] O. Cozette, A. Guermouche, and G. Utard. Adaptive paging for a multifrontal solver. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 267–276. ACM Press, 2004.

[34] T. A. Davis. Algorithm 832: UMFPACK V4.3 — an unsymmetric-pattern multifrontal method with a column pre-ordering strategy. *ACM Trans. Math. Softw.*, 30(2):196–199, 2004.

[35] A. de la Garza. An iterative method for solving systems linear equations. Technical Report K-731, Union Carbide, 1951.

[36] E. Demir, C. Aykanat, and B. B. Cambazoglu. Clustering spatial networks for aggregate query processing: a hypergraph approach. *Information Systems*, 33:1–17, 2008.

[37] E. Demir, C. Aykanat, and B. B. Cambazoglu. A link-based storage scheme for efficient aggregate query processing on clustered road networks. *Information Systems*, 2009. To appear.

[38] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *Appear in SIAM J. Matrix Anal. Appl.*, 1995.

[39] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[40] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[41] N. Dingle, P. Harrison, and W. Knottenbelt. Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large markov models. *Journal of Parallel and Distributed Computing*, 64:908–920, 2004.

[42] F. Dobrian and A. Pothen. Oblio: a sparse direct solver library for serial and parallel computations. Technical report, Old Dominion University, 2000.

[43] F. Dobrian and A. Pothen. The design of I/O-efficient sparse direct solvers. In *Proceedings of SuperComputing*, 2001.

[44] I. S. Duff. Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core. *SIAM Journal on Scientific and Statistical Computing*, 5:270–280, 1984.

[45] I. S. Duff. Sparse numerical linear algebra: direct methods and preconditioning. Technical Report TR-PA-96-22, CERFACS, 1996.

[46] I. S. Duff, A. M. Erisman, C. W. Gear, and J. K. Reid. Sparsity structure and Gaussian elimination. *SIGNUM Newsletter*, 23(2):2–8, Apr. 1988.

[47] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, London, 1986.

[48] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. Technical Report RAL-TR-97-059, Rutherford Appleton Laboratory, 1997.

[49] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. Technical Report RAL-TR-1999-030, Rutherford Appleton Laboratory, 1999. Also CERFACS Report TR/PA/99/13.

[50] I. S. Duff and J. K. Reid. A comparison of some methods for the solution of sparse overdetermined systems of linear equations. *J. Inst. Maths. Applics.*, 17:267–280, 1976.

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

[52] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.

[53] S. C. Eisenstat and J. W. H. Liu. A tree based dataflow model for the unsymmetric multifrontal method. *Electronic Transaction on Numerical Analysis*, 21:1–19, 2005.

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

[55] L. V. Foster. Rank and nullspace calculations using matrix decompositions without column interchanges. *Linear Algebra and its Applications*, 74:47–71, 1986.

[56] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[57] A. Geist and E. G. Ng. Task scheduling for parallel sparse Cholesky factorization. *Int J. Parallel Programming*, 18:291–314, 1989.

[58] A. George and J. W. H. Liu. A quotient graph model for symmetric factorization. In I. S. Duff and G. W. Stewart, editors, *Sparse Matrix Proceedings*, pages 154–175. SIAM, 1978.

[59] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ., 1981.

[60] A. George and E. G. Ng. On row and column orderings for sparse least squares problems. *SIAM J. Numer. Anal.*, 20:326–344, 1981.

[61] J. A. George, M. T. Heath, and E. G. Ng. A comparison of some methods for solving sparse linear least squares problems. *SIAM J. Scient. Statist. Comput.*, 4:177–187, 1983.

[62] J. R. Gilbert. Predicting structure in sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 15:62–79, 1994.

[63] J. R. Gilbert and J. W. H. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM Journal on Matrix Analysis and Applications*, 14:334–352, 1993.

[64] J. R. Gilbert and E. G. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. In J. G. A. George and J. Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 107–140. Springer-Verlag NY, 1993.

[65] J. R. Gilbert and S. Toledo. High-performance out-of-core sparse LU factorization. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, 1999. (10 pages on CDROM).

[66] L. Giraud, A. Marrocco, and J.-C. Rioual. Iterative versus direct parallel substructuring methods in semiconductor device modelling. *Numerical Linear Algebra with Applications*, 12(1):33–53, 2005.

[67] G. H. Golub and S. G. Nash. Nonorthogonal analysis of variance using a generalized conjugate-gradient algorithm. *J. of the American Stat.Assoc.*, 77:109–116, 1982.

[68] A. Guermouche. *Étude et optimisation du comportement mémoire dans les méthodes parallèles de factorisation de matrices creuses*. PhD thesis, École Normale Supérieure de Lyon, July 2004.

[69] A. Guermouche, O. Cozette, and G. Utard. Study of the paging activity of the parallel multifrontal method. In *3rd International Workshop on Parallel Matrix Algorithms and Applications (PMAA'04), CIRM, Marseille, France.*, Oct. 2004.

[70] A. Guermouche and J.-Y. L'Excellent. Memory-based scheduling for a parallel multifrontal solver. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, page 71a (10 pages), 2004.

[71] A. Guermouche and J.-Y. L'Excellent. Optimal memory minimization algorithms for the multifrontal method. Research report RR2004-26, LIP, 2004. Also INRIA report RR-5179.

[72] A. Guermouche and J.-Y. L'Excellent. Flexible task allocation for the memory minimization of the multifrontal approach, June 2005. Second International Workshop on Combinatorial Scientific Computing (CSC05), CERFACS, Toulouse, France.

[73] A. Guermouche and J.-Y. L'Excellent. Constructing memory-minimizing schedules for multifrontal methods. *ACM Transactions on Mathematical Software*, 32(1):17–32, 2006.

[74] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191–1218, 2003.

[75] A. Gupta. WSMP: Watson Sparse Matrix Package part i - direct solution of symmetric sparse systems version 1.0.0. Technical Report TR RC-21886, IBM research division, T.J. Watson Research Center, Yorktown Heights, 2000.

[76] A. Gupta. WSMP: Watson Sparse Matrix Package part ii - direct solution of general sparse systems version 1.0.0. Technical Report TR RC-21888, IBM research division, T.J. Watson Research Center, Yorktown Heights, 2000.

[77] A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar. Pspases: An efficient and scalable parallel sparse direct solver. Technical report, Department of Computer Science, University of Minnesota and IBM T.J. Watson Research center, 1999.

[78] M. T. Heath, E. G. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.

[79] P. Hénon, P. Ramet, and J. Roman. A mapping and scheduling algorithm for parallel sparse fan-in numerical factorization. In *EuroPar'99 Parallel Processing*, Lecture Notes in Computer Science, No. 1685, pages 1059–1067, Berlin, Heidelberg, New York, 1999. Springer-Verlag.

[80] P. Hénon, P. Ramet, and J. Roman. PaStiX: A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions. In *Proceedings of Irregular'2000, Cancun, Mexique*, number 1800 in Lecture Notes in Computer Science, pages 519–525. Springer Verlag, May 2000.

[81] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, Jan. 2002.

[82] HSL. HSL 2007: A collection of Fortran codes for large scale scientific computation, 2007.

[83] G. Karypis and V. Kumar. METIS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*. University of Minnesota, Sept. 1998.

[84] K. Kaya and C. Aykanat. Iterative-improvement-based heuristics for adaptive scheduling of tasks sharing files on heterogeneous master-slave environments. *IEEE Transactions on Parallel and Distributed Systems*, 17:883–896, 2006.

[85] K. Kaya, B. Uçar, and C. Aykanat. Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories. *Journal of Parallel and Distributed Computing*, 67:271–285, 2007.

[86] G. Khanna, N. Vydyanathan, T. Kurc, Ü. Çatalyürek, P. Wyckoff, J. Saltz, and P. Sadayappan. A hypergraph partitioning based approach for scheduling of tasks with batch-shared io. In *Cluster Computing and Grid*, 2005.

[87] M. Koyuturk and C. Aykanat. Iterative-improvement-based declustering heuristics for multi-disk databases. *Information Systems*, 30:47–70, 2005.

[88] C. L. Lawson and R. J. Hanson. *Solving Least Squares Problems*. Prentice Hall, Englewood Cliffs, New Jersey, 1974.

[89] T. Lengauer. Combinatorial algorithms for integrated circuit layout. *Wiley-Teubner, Chichester, U.K.*, 1990.

[90] X. S. Li and J. W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of Supercomputing*, Orlando, Florida, November 1998.

[91] D. Liu and M. Wu. A hypergraph based approach to declustering problems. *Distributed and Parallel Databases*, 10:269–288, 2001.

[92] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[93] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and Practice. *SIAM Review*, 34:82–109, 1992.

[94] J. W. H. Liu, E. G. Ng, and B. W. Peyton. On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 14:242–252, 1993.

[95] J. W. Longley. *Least Squares Computations Using Orthogonal Methods*. Marcel Dekker, Inc., New York, 1984.

[96] P. Manneback. *On Some Numerical Methods for Solving Large Sparse Linear Least Squares Problems*. PhD thesis, Facultés Universitaires Notre-Dame de la Paix, 1985.

[97] P. Matstoms. *The Multifrontal Solution of Sparse Linear Least Squares Problems*. PhD thesis, Linköping University, 1991. licentiat thesis.

[98] E. G. Ng and B. W. Peyton. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 14:761–769, 1993.

[99] E. E. Osborne. On least squares solutions of linear equations. *J. ACM*, 8:628–636, 1961.

[100] M. Ozdal and C. Aykanat. Hypergraph models and algorithms for data-pattern-based clustering. *Data Mining and Knowledge Discovery*, 9:29–57, 2004.

[101] G. Peters and J. H. Wilkinson. The least squares problem and pseudo-inverses. *The Computer Journal*, 13:309–316, 1970.

[102] J. K. Reid and J. A. Scott. An out-of-core sparse Cholesky solver. Technical Report RAL-TR-2006-013, Rutherford Appleton Laboratory, 2006. Revised March 2007.

[103] J. K. Reid and J. A. Scott. HSL_OF01, a virtual memory system in Fortran. Technical report, Rutherford Appleton Laboratory, 2006.

[104] E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, 1999.

[105] V. Rotkin and S. Toledo. The design and implementation of a new out-of-Core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software*, 30(1):19–46, 2004.

[106] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20(3):475–487, 2004.

[107] O. Schenk and K. Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Transaction on Numerical Analysis*, 23:158–179, 2006.

[108] O. Schenk, S. Röllin, and A. Gupta. The effects of unsymmetric matrix permutations and scalings in semiconductor device and circuit simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23:400–411, 2004.

[109] D. Schweikert and B. Kernighan. A proper model for the partitioning of electrical circuits. pages 57–62, 1972. Proceedings of the 9th Workshop on Design Automation.

[110] S. Shekhar, C.-T. Lu, S. Chawla, and S. Ravada. Efficient join-index-based spatial-join processing: a clustering approach. *IEEE Transactions on Knowledge and Data Engineering*, 14:1400–1421, 2002.

[111] K. Takahashi, J. Fagan, and M.Chin. Formation of a sparse bus impedance matrix and its application to short circuit study. In *Proceedings of the 8^{th} PICA Conference, Minneapolis*, pages 177–179. Minneapolis, June 1973.

[112] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26:690–715, 1979.

[113] S. Toledo. TAUCS: A library of sparse linear solvers, version 2.2, 2003. Available online at http://www.tau.ac.il/˜stoledo/taucs/.

[114] S. Toledo and A. Uchitel. A supernodal out-of-core sparse gaussian elimination method. In *Proceedings of PPAM 2007*, 2007.

[115] B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiples. *SIAM Journal of Scientific Computing*, 25:1837–1859, 2004.

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

[117] B. Uçar, C. Aykanat, M. Pınar, and T. Malas. Parallel image restoration using surrogate constraints methods. *Journal of Parallel and Distributed Computing*, 67:186–204, 2007.

[118] G. Vedrenne, J.-P. Roques, V. Schönfelder, P. Mandrou, G. G. Lichti, A. von Kienlin, B. Cordier, S. Schanne, J. Knödlseder, G. Skinner, P. Jean, F. Sanchez, P. Caraveo, B. Teegarden, von P. Ballmoos, L. Bouchet, P. Paul, J. Matteson, S. Boggs, C. Wunderer, P. Leleux, G. Weidenspointner, P. Durouchoux, R. Diehl, A. Strong, M. Cassé, M. A. Clair, and Y. André. Spi: The spectrometer aboard integral. pages 63–70, 2003.

[119] C. Winkler, T. J.-L. Courvoisier, G. D. Cocco, N. Gehrels, A. Giménez, S. Grebenev, W. Hermsen, J. M. Mas-Hesse, F. Lebrun, N. Lund, G. G. C. Palumbo, J. Paul, J.-P. Roques, H. Schnopper, V. Schönfelder, R. Sunyaev, B. Teegarden, P. Ubertini, G. Vedrenne, and A. J. Dean. The integral mission. In E. J. A. M. G. F. C. C. U. Press, editor, *Populations of High Energy Sources in Galaxies Proceedings of the 230th Symposium of the International Astronomical Union*, pages 59–65, 2003.

Abstract

We consider the solution of very large systems of linear equations with direct multifrontal methods. In this context the size of the factors is an important limitation for the use of sparse direct solvers. We will thus assume that the factors have been written on the local disks of our target multiprocessor machine during parallel factorization. Our main focus is the study and the design of efficient approaches for the forward and backward substitution phases after a sparse multifrontal factorization. These phases involve sparse triangular solution and have often been neglected in previous works on sparse direct factorization. In many applications, however, the time for the solution can be the main bottleneck for the performance.

This thesis consists of two parts. The focus of the first part is on optimizing the out-of-core performance of the solution phase. The focus of the second part is to further improve the performance by exploiting the sparsity of the right-hand side vectors.

In the first part, we describe and compare two approaches to access data from the hard disk. We then show that in a parallel environment the task scheduling can strongly influence the performance. We prove that a constraint ordering of the tasks is possible; it does not introduce any deadlock and it improves the performance. Experiments on large real test problems (more than 8 million unknowns) using an out-of-core version of a sparse multifrontal code called `MUMPS` (MUltifrontal Massively Parallel Solver) are used to analyse the behaviour of our algorithms.

In the second part, we are interested in applications with sparse multiple right-hand sides, particularly those with single nonzero entries. The motivating applications arise in electromagnetism and data assimilation. In such applications, we need either to compute the null space of a highly rank deficient matrix or to compute entries in the inverse of a matrix associated with the normal equations of linear least-squares problems. We cast both of these problems as linear systems with multiple right-hand side vectors, each containing a single nonzero entry. We describe, implement and comment on efficient algorithms to reduce the input-output cost during an out-of-core execution. We show how the sparsity of the right-hand side can be exploited to limit both the number of operations and the amount of data accessed.

The work presented in this thesis has been partially supported by SOLSTICE ANR project (ANR-06-CIS6-010).

Keyword: Gaussian elimination, multifrontal method, Distributed computing, parallel computing, sparse matrices, tasks scheduling, multiple right-hand side vectors.

Résumé

Nous nous intéressons à la résolution de systèmes linéaires creux de très grande taille par des méthodes directes de factorisation. Dans ce contexte, la taille de la matrice des facteurs constitue un des facteurs limitants principaux pour l'utilisation de méthodes directes de résolution. Nous supposons donc que la matrice des facteurs est de trop grande taille pour être rangée dans la mémoire principale du multiprocesseur et qu'elle a donc été écrite sur les disques locaux (hors-mémoire : OOC) d'une machine multiprocesseurs durant l'étape de factorisation. Nous nous intéressons à l'étude et au développement de techniques efficaces pour la phase de résolution après une factorization multifrontale creuse. La phase de résolution, souvent négligée dans les travaux sur les méthodes directes de résolution directe creuse, constitue alors un point critique de la performance de nombreuses applications scientifiques, souvent même plus critique que l'étape de factorisation.

Cette thèse se compose de deux parties. Dans la première partie nous nous proposons des algorithmes pour améliorer la performance de la résolution hors-mémoire. Dans la deuxième partie nous pousuivons ce travail en montrant comment exploiter la nature creuse des seconds membres pour réduire le volume de données accédées en mémoire.

Dans la première partie de cette thèse nous introduisons deux approches de lecture des données sur le disque dur. Nous montrons ensuite que dans un environnement parallèle le séquencement des tâches peut fortement influencer la performance. Nous prouvons qu'un ordonnancement contraint des tâches peut être introduit; qu'il n'introduit pas d'interblocage entre processus et qu'il permet d'améliorer les performances. Nous conduisons nos expériences sur des problèmes industriels de grande taille (plus de 8 Millions d'inconnues) et utilisons une version hors-mémoire d'un code multifrontal creux appelé `MUMPS` (solveur multifrontal parallèle).

Dans la deuxième partie de ce travail nous nous intéressons au cas de seconds membres creux multiples. Ce problème apparaît dans des applications en electromagnétisme et en assimilation de données et résulte du besoin de calculer l'espace propre d'une matrice fortement déficiente, du calcul d'éléments de l'inverse de la matrice associée aux équations normales pour les moindres carrés linéaires ou encore du traitement de matrices fortement réductibles en programmation linéaire. Nous décrivons un algorithme efficace de réduction du volume d'Entrées/Sorties sur le disque lors d'une résolution hors-mémoire. Plus généralement nous montrons comment le caractère creux des seconds -membres peut être exploité pour réduire le nombre d'opérations et le nombre d'accès à la mémoire lors de l'étape de résolution.

Le travail présenté dans cette thèse a été partiellement financé par le projet SOLSTICE de l'ANR (ANR-06-CIS6-010).

Mots-clés: calcul distribué, calcul parallèle, élimination de Gauss, matrices creuses, méthode multifrontale, séquencement des tâches, seconds membres multiples

Thèse préparée au CERFACS, CERFACS Report Ref: TH-PA-09-59
42, Avenue Gaspard Coriolis. 31057 Toulouse Cedex 01. France.