Généralités sur les solveurs linéaires directs et utilisation de MUMPS

Résumé
Dans le cadre des simulations thermo-mécaniques avec *Code_Aster*, **l'essentiel des coûts calcul provient souvent de la construction et de la résolution des systèmes linéaires**. Pour effectuer plus efficacement ces résolutions, ***Code_Aster* a fait le choix d'intégrer la méthode directe déployée dans le package MUMPS** ('Multifrontal Massively Parallel sparse direct Solver; P.R.Amestoy, J.Y.L'Excellent et al; CERFACS/CNRS/ENS LyonIRIT/INRIA/Université de Bordeaux). Ceci en complément de sa multifrontale «maison» (C.Rose) et de ses autres solveurs: 'LDLT', 'GCPC' et 'PETSC'.
En mode parallèle distribué et en Out-Of-Core, le couplage *Aster*+MUMPS procure des **gains en CPU de l'ordre de la douzaine** sur 32 processeurs. Et ce, pour des **consommations RAM**, un **comportement/périmètre fonctionnel** et une **précision des résultats** au moins aussi bons que **ceux de la multifrontale native** du code.
Sur les gros problèmes, en activant les compressions BLR, on peut même gagner un facteur deux ou trois supplémentaires. Ce produit, *via* le parallélisme qu'il exhibe et ses fonctionnalités avancées (pivotage, pré/post-traitements, qualité de la solution…) facilite grandement le passage des études standards. De plus, Il reste souvent la **seule alternative viable** pour exploiter certaines modélisations/analyses (quasi-incompressible, X-FEM…) ou passer de très grosses études (en tant que solveur direct précis ou en tant que préconditionneur, cf. option 'LDLT_SP' de 'GCPC'/'PETSC').
Dans la première partie du document nous résumons la problématique générale de résolution de systèmes linéaires, puis nous abordons les grandes familles de solveurs directs creux et leurs déclinaisons dans les bibliothèques du domaine public.
Toutes choses qu'il faut avoir à l'esprit avant d'aborder, **dans la seconde partie**, le package MUMPS au travers de ses principales caractéristiques et de ses fonctionnalités avancées. Puis nous détaillons les aspects numériques, informatiques et fonctionnels de son intégration dans *Code_Aster*. Enfin, nous concluons par quelques résultats numériques.

Pour plus de détails et de conseils sur l'emploi des solveurs linéaires on pourra consulter les notices d'utilisation spécifiques [U4.50.01]/[U2.08.03]. Les problématiques connexes d'amélioration des performances (RAM/CPU) d'un calcul et, de l'utilisation du parallélisme, font aussi l'objet de notices détaillées: [U1.03.03] et [U2.08.06].

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

Table des Matières

1 Généralités sur les solveurs directs....4
1.1 Système linéaire et méthodes de résolution associées....4
1.2 Les librairies d'algèbre linéaire....6
1.3 Méthodes directes: le principe....7
1.4 Méthodes directes: les différentes approches....9
1.5 Méthodes directes: les principales étapes....11
1.6 Méthodes directes: les difficultés....12
2 Le package MUMPS....14
2.1 Historique....14
2.2 Principales caractéristiques....15
2.3 Zooms sur quelques points techniques....16
2.3.1 Pivotage....16
2.3.2 Raffinement itératif....17
2.3.3 Fiabilité des calculs....18
2.3.4 Gestion mémoire (In-Core versus Out-Of-Core)....20
2.3.5 Gestion des matrices singulières....21
2.3.6 Compression 'Block Low-Rank' (BLR)....22
3 Implantation dans Code_Aster....24
3.1 Contexte/synthèse....24
3.2 Deux types de parallélisme: centralisé et distribué....24
3.2.1 Principe....24
3.2.2 Différents modes de distribution....25
3.2.3 Équilibrage de charge....26
3.2.4 Retailler les objets Code_Aster....26
3.3 Gestion de la mémoire MUMPS et Code_Aster....27
3.4 Gestion particulière des double multiplicateurs de Lagrange....28
3.5 Périmètre d'utilisation....29
3.6 Paramétrage et exemples d'utilisation....29
3.6.1 Paramètres d'utilisation de MUMPS via Code_Aster....29
3.6.2 Monitoring....30
3.6.3 Exemples d'utilisation....31
4 Conclusion....33
5 Bibliographie....34
5.1 Livres/articles/proceedings/thèses....34
5.2 Rapports/compte-rendus EDF....34
5.3 Ressources internet....34
6 Historique des versions du document....35

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

7 Annexe n°1: Principe des compressions BLR dans MUMPS....36
7.1 Principe de la méthode multifrontale....36
7.2 Arbre d'élimination....37
7.3 Traitements algorithmiques....38
7.4 Gestion de la mémoire....39
7.5 Approximation low-rank....40
7.6 Multifrontale 'Block Low-Rank' (BLR)....41
7.7 Quelques éléments complémentaires....43

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

1 Généralités sur les solveurs directs

1.1 Système linéaire et méthodes de résolution associées

En simulation numérique de phénomènes physiques, **un coût calcul important provient souvent de la construction et de la résolution de systèmes linéaires**. La mécanique des structures n'échappe pas à la règle ! Le coût de la construction du système dépend du nombre de points d'intégration et de la complexité des lois de comportement, tandis que celui de la résolution est lié au nombre d'inconnues, aux modélisations retenues et à la topologie (largeur de bande, conditionnement). Lorsque le nombre d'inconnues explose, la seconde étape devient prépondérante[1] et c'est donc cette dernière qui va principalement nous intéresser ici. D'ailleurs, lorsqu'il est possible d'être plus performant sur cette phase de résolution, grâce à l'accès à une machine parallèle, nous verrons que cet atout pourra se propager à la phase de constitution du système proprement dite (calculs élémentaires et assemblages).
Ces **inversions de systèmes linéaires sont en fait omniprésentes dans les codes et souvent enfouies au plus profond d'autres algorithmes numériques**: schéma non-linéaire, intégration en temps, analyse modale…. On cherche, par exemple, le vecteur des déplacements nodaux (ou des incréments de déplacement) $\mathbf{u}$ vérifiant un système linéaire du type

$$\mathbf{Ku} = \mathbf{f} \tag{1.1-1}$$

avec $\mathbf{K}$ une matrice de rigidité et $\mathbf{f}$ un vecteur traduisant l'application de forces généralisées au système mécanique.

De manière générale, la **résolution de ce type de problème requiert un (plus) large questionnement** qu'il n'y parait:

- A-t-on accès à la matrice ou connaît-on simplement son action sur un vecteur ?
- Cette matrice est elle creuse ou dense ?
- Qu'elles sont ses propriétés numériques (symétrie, définie positive…) et structurelles (réelle/complexe, par bandes, par blocs..) ?
- Veux-t-on résoudre un seul système (1.1-1), plusieurs en simultané [2] ou de manière consécutive [3] ? Voire plusieurs systèmes différents et successifs dont les matrices sont très proches [4] ?
- Dans le cas de résolutions successives, peux-t-on réutiliser des résultats précédents afin de faciliter les prochaines résolutions (cf. technique de redémarrage, factorisation partielle) ?
- Quel est l'ordre de grandeur de la taille du problème, de la matrice et de sa factorisée par rapport aux capacités de traitements des CPU et des mémoires associées (RAM, disque) ?
- Veut on une solution très précise ou simplement une estimation (cf. solveurs emboîtés) ?
- A-t-on accès à des librairies d'algèbres linéaires (et à leurs pré-requis MPI, BLAS, LAPACK…) ou doit-on faire appel à des produits «maison» ?

Dans *Code_Aster* , on construit explicitement la matrice et on la stocke au format MORSE[5] . Avec la plupart des modélisations, la matrice est creuse (du fait de la discrétisation par éléments finis), potentiellement mal conditionnée[6] et souvent réelle, symétrique et indéfinie[7] . En non linéaire, en modal ou lors de chaînages thermo-mécaniques, on traite souvent des problèmes de type «multiples seconds membres». Les méthodes discrètes de contact-frottement tirent profit de facultés de factorisation partielle et la méthode par décomposition de domaines. D'autre part, *Code_Aster* utilise aussi des scénarios de résolutions simultanées (compléments de Schur du contact et de la sous-structuration…).

1 Pour *Code_Aster*, voir l'étude de 'profiling' conduite par N.Anfaoui[Anf03].
2 Même matrice mais plusieurs seconds membres indépendants; Cf. construction d'un complément de Schur.
3 Problème de type multiples seconds membres: même matrice mais plusieurs seconds membres successifs et interdépendants; Cf. algorithme de Newton sans réactualisation de la matrice tangente.
4 Problème de type multiples premiers membres: plusieurs matrices et seconds membres successifs et interdépendants, mais avec des matrices «spectralement» proches; Cf. algorithme de Newton avec réactualisation de la matrice tangente.
5 Dit encore SCR pour 'Symmetric Compressed Row storage' (en fait 'Column' dans *Code_Aster*).
6 En mécanique des structures le conditionnement $\eta(\mathbf{K})$ est connu pour être assez mauvais. Il peut varier, typiquement, de 10^5 à 10^{12} et le fait de raffiner le maillage, d'utiliser des éléments étirés ou des éléments structuraux a des conséquences dramatiques sur ce chiffre (cf. B.Smith. *A parallel implementation of an iterative substructuring algorithm for problems in 3D.* SIAM J.Sci.Comput., **14** (1992), pp406-423. §3.1 ou I.FRIED. *Condition of finite element matrices generated from nonuniform meshes*. AIAA J., **10** (1972), pp219-221.)
7 Le caractère indéfini plutôt que défini positif est du à l'adjonction de variables supplémentaires (dites «de Lagrange») pour imposer des conditions limites de Dirichlet simples ou généralisées[R3.03.01].

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

Quant aux tailles des problèmes, même si elles enflent d'année en année, elles restent modestes par rapport à la CFD : de l'ordre du million d'inconnues mais pour des centaines de pas de temps ou d'itérations de Newton.
Par ailleurs, d'un **point de vue «middleware et hardware»** , le code s'appuie désormais sur de nombreuses librairies optimisées et pérennes (MPI, BLAS, LAPACK, (PAR)METIS, (PT)SCOTCH, PETSc, MUMPS…) et s'utilise principalement sur des clusters de SMP (réseaux rapides, grande capacité de stockage RAM et disque). On cherche donc surtout à optimiser l'utilisation des solveurs linéaires dans cette optique.

Depuis 60 ans, deux types de techniques se disputent la suprématie dans le domaine, les solveurs directs et les solveurs itératifs (cf. [Che05][Dav03][Duf06][Gol96][Las98][Liu89][Meu99][Saa03]).
Les premiers sont robustes et aboutissent en un nombre fini d'opérations (théoriquement) connu par avance. Leur théorie est relativement bien achevée et leur déclinaison suivant moults types de matrices et d'architectures logicielles est très complète. En particulier, leur algorithmique multi-niveaux est bien adaptée aux hiérarchies mémoires des machines actuelles. Cependant, ils requièrent des capacités de stockage qui croissent rapidement avec la taille du problème ce qui limite l'extensibilité de leur parallélisme[8]. Même si ce parallélisme peut se décomposer en plusieurs strates indépendantes, démultipliant ainsi les performances.
En revanche, les **méthodes itératives** sont plus «scalables» lorsque l'on augmente le nombre de processeurs. Leur théorie regorge de nombreux «problèmes ouverts», surtout en arithmétique finie. En pratique, leur convergence en un nombre «raisonnable» d'itérations, n'est pas toujours acquise, elle dépend de la structure de la matrice, du point de départ, du critère d'arrêt… Ce type de solveurs a plus de mal à percer en mécanique des structures industrielles où l'on cumule souvent hétérogénéités, non-linéarités et jonctions de modèles qui gangrènent le conditionnement de l'opérateur de travail. D'autre part, elles ne sont pas adaptées pour résoudre efficacement les problèmes de type «multiples seconds membres». Hors ceux-ci sont très fréquents dans l'algorithmique des simulations mécaniques.

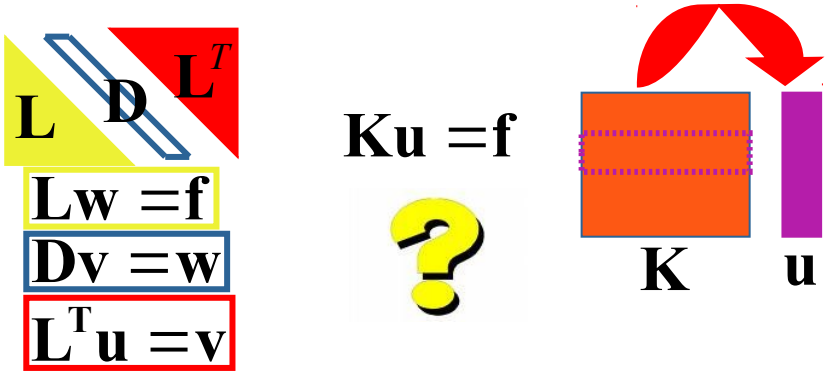


Figure 1.1-1._ Deux classes de méthodes pour résoudre un système linéaire: les directes et les itératives.

Contrairement à leurs homologues directs, il n'est pas possible de proposer LE solveur itératif qui va résoudre n'importe quel système linéaire. L'adéquation du type d'algorithme à une classe de problèmes se fait au cas par cas. Ils présentent, néanmoins, d'autres avantages qui historiquement leur ont donné droit de cité pour certaines applications. A gestion mémoire équivalente, ils en requièrent moins que les solveurs directs, car on a juste besoin de connaître l'action de la matrice sur un vecteur quelconque, sans devoir véritablement la stocker. D'autre part, on n'est pas soumis au «diktat» du phénomène de remplissage qui détériore le profil des matrices, on peut exploiter efficacement le caractère creux des opérateurs et contrôler la précision des résultats [9] . **Bref, l'utilisation de solveurs directs relève plutôt du domaine de la technique alors que le choix du bon couple méthode itérative/préconditionneur est plutôt un art !** En dépit de sa simplicité biblique sur le papier, la résolution d'un système linéaire, même symétrique défini positif, n'est pas «un long fleuve tranquille». Entre deux maux, remplissage/pivotage et préconditionnement, il faut choisir !

Remarques:

• *Une troisième classe de méthodes essaie de tirer partie des avantages respectifs des directes et des itératives: les méthodes de Décomposition de Domaine (DD)[R6.01.03].*

8 On parle aussi de «scalabilité» ou de passage à l'échelle.
9 Ce qui peut être très intéressant dans le cadre de solveurs emboîtés (par ex. Newton+GCPC), cf. V.Frayssé. *The power of backward error analysis*. HDR de l'Institut National Polytechnique de Toulouse (2000).

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

• Les deux grandes familles de méthodes doivent plus être vues comme complémentaires que comme concurrentes. On cherche souvent à les mixer: méthodes DD, préconditionneur par factorisation incomplète (cf. [R6.01.02] § 4.2) ou de type multigrille, procédure de raffinement itératif en fin de solveur direct…

1.2 Les librairies d'algèbre linéaire

Pour effectuer efficacement la résolution d'un système linéaire, **la question du recourt à une librairie ou à un produit externe est désormais incontournable**. Pourquoi ? Parce que cette stratégie permet:

- Des développements moins techniques, moins invasifs et beaucoup plus rapides dans le code hôte.
- D'acquérir, à moindre frais, un large périmètre d'utilisation tout en externalisant bon nombres des contingences associées (typologie du problème cf. §1.1, représentation des données, architecture de la machine cible…).
- De bénéficier du retour d'expérience d'une communauté d'utilisateurs variée et des compétences (très) pointues d'équipes internationales.

Ces librairies conjuguent en effet souvent efficacité, fiabilité, performance et portabilité:

- Efficacité car elles exploitent la localité spatiale et temporelle des données et jouent sur la hiérarchie mémoire (exemple des différentes catégories de BLAS).
- Fiabilité car elles proposent des outils pour estimer l'erreur commise sur la solution (estimation du conditionnement et des 'backward/forward errors') voire pour l'améliorer (pour les directs, équilibrage de la matrice et raffinement itératif).

Depuis **l'émergence dans les années 70/80 des premières librairies publics**[10] **et privées/constructeurs**[11] et de leurs communautés d'utilisateurs, l'offre s'est démultipliée. La tendance étant bien sûr de proposer des solutions performantes (vectoriel, parallélisme à mémoire centralisé puis distribué, parallélisme multiniveau *via* des threads) ainsi que des «toolkits» de manipulation d'algorithmes d'algèbre linéaire et des structures de données associées. Citons de manière non exhaustive: ScaLAPACK(Dongarra & Demmel 1997), SparseKIT(Saad 1988), PETSc(Argonne 1991), HyPre(LL 2000), TRILINOS(Sandia 2000)…



Figure 1.2-1._ Quelques «logos» de bibliothèques d'algèbre linéaire.

Remarque:

• Pour structurer plus efficacement leur utilisation et proposer des solutions «boîte noire», des macro-librairies ont récemment vue le jour. Elles regroupent un panel de ces produits auxquels elles rajoutent des solutions «maisons» : Numerical Platon (CEA-DEN), Arcane (CEA-DAM)…

Concernant plus spécifiquement les **méthodes directes de résolution de systèmes linéaires** , une cinquantaine de packages sont disponibles. On distingue les produits «autonomes» de ceux incorporés à une librairie, les publics des commerciaux, ceux traitant des problèmes denses et d'autres des creux. Certains ne fonctionnent qu'en mode séquentiel, d'autres supportent un parallélisme à mémoire partagée et/ou distribuée. Enfin, certains produits sont généralistes (symétrique, non symétrique, SPD, réel/complexe…) d'autres adaptés à un besoin/scénario bien précis.
On peut trouver une liste assez exhaustive de tous ces produits sur le site d'un des pères fondateurs de LAPACK/BLAS: Jack Dongarra[Don]. Le tableau ci-dessous (tableau 1.2-1) en est une version expurgée. Il ne reprend que les solveurs directs du domaine public et oublie de mentionner: CHOLMOD, CSPARSE, DMF, Oblio, PARASPAR, PARDISO, PaStiX (l'autre solveur direct français avec MUMPS), S+, SPRSBLKKT et

10 EISPACK(1974), LINPACK(1976), BLAS(1978) puis LAPACK(1992)…
11 NAG(1971), IMSL/ESSL(IBM 1971), ASL/MathKeisan(NEC), SciLib(Cray), MKL(Intel), HSL(Harwell)…

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

WSMP. Cette ressource internet recense aussi des packages implémentant des solveurs itératifs, des préconditionneurs, des solveurs modaux ainsi que de nombreux produits support (BLAS, LAPACK, ATLAS…).

DIRECT SOLVERS	License	Support	Real	Complex	F77	C	Seq	Dist	SPD	Gen
DENSE										
FLAME	LGPL	oui	X	X	X	X	X			
LAPACK	BSD	oui	X	X	X	X	X			
LAPACK95	BSD	oui	X	X	95		X			
NAPACK	BSD	oui	X		X		X			
PLAPACK	?	oui	X	X	X	X		M		
PRISM	?	non	X		X		X	M		
ScaLAPACK	BSD	oui	X	X	X	X		M/P		
Trilinos/Pliris	LGPL	oui	X	X		X et C++		M		
SPARSE										
DSCPACK	?	oui	X			X	X	M	X	
HSL	?	oui	X	X	X		X		X	X
MFACT	?	oui	X			X	X	M	X	
MUMPS	PD	oui	X	X	X	X	X	M	X	X
PSPASES	?	oui	X		X	X		M	X	
SPARSE	?	?	X	X		X	X		X	X
SPOOLES	PD	?	X	X		X	X	M		X
SuperLU	Own	oui	X	X	X	X	X	M		X
TAUCS	Own	oui	X	X		X	X		X	X
Trilinos/Amesos	LGPL	oui	X				X	M	X	X
UMFPACK	LGPL	oui	X	X		X	X			X
Y12M	?	oui	X		X		X		X	X

Tableau 1.2-1._ Extrait de la page web de Jack Dongarra[Don] sur les produits libres implémentant une méthode directe; 'Seq' pour séquentiel, 'Dist' pour parallèle ('M' OpenMP et 'P' MPI), 'SPD' pour symétrique définie positive et 'Gen' pour matrice quelconque.

Remarque:

• Une ressource internet plus détaillée mais focalisée sur les solveurs directs creux est maintenue par un autre grand nom du numérique: T.A.Davis[Dav], un des contributeurs de Matlab.

1.3 Méthodes directes: le principe

L'idée de base des méthodes directes est de **décomposer la matrice du problème $\mathbf{K}$ en un produit de matrices particulières** (triangulaires inférieure et supérieure, diagonale) plus faciles à «inverser». C'est ce qu'on appelle la **factorisation**[12] de la matrice de travail:

- *Si $\mathbf{K}$ est SPD*, elle admet la «factorisation de Cholesky» unique: $\mathbf{K}=\mathbf{L}\mathbf{L}^{\mathbf{T}}$ avec $\mathbf{L}$ triangulaire inférieure;
- *Si $\mathbf{K}$ est symétrique quelconque et régulière*, elle admet au moins une «factorisation $\mathbf{L}\mathbf{D}\mathbf{L}^{\mathbf{T}}$»: $\mathbf{P}\mathbf{K}=\mathbf{L}\mathbf{D}\mathbf{L}^{\mathbf{T}}$ avec $\mathbf{L}$ triangulaire inférieure à coefficients diagonaux égaux à l'unité, $\mathbf{D}$ une matrice diagonale[13] et $\mathbf{P}$ une matrice de permutation;
- *Si $\mathbf{K}$ est quelconque et régulière*, elle admet au moins une «factorisation $\mathbf{L}\mathbf{U}$»: $\mathbf{P}\mathbf{K}=\mathbf{L}\mathbf{U}$ avec $\mathbf{L}$ triangulaire inférieure à diagonale l'unité, $\mathbf{U}$ triangulaire supérieure et $\mathbf{P}$ une matrice de permutation;

12 Par analogie avec les factorisations polynomiales des petites classes…

13 Elle peut comporter aussi des blocs diagonaux 2×2.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

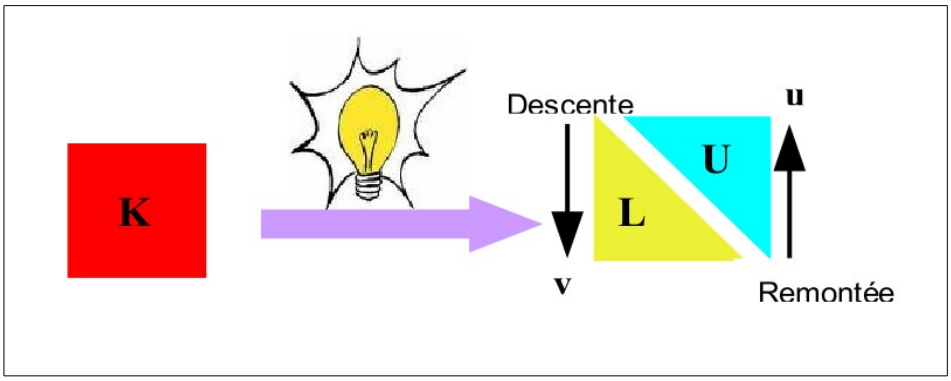


Figure 1.3-1._ Principe des méthodes directes.

Remarque:

• *Par exemple, la matrice symétrique et régulière* $\mathbf{K}$ *ci-dessous se décompose sous la forme* $\mathbf{L}\,\mathbf{D}\,\mathbf{L}^{\mathbf{T}}$ *suivante (sans avoir besoin ici de permutation* $\mathbf{P}=\mathbf{Id}$ *)*

$$\mathbf{K}:=\begin{bmatrix}10 & & sym\\ 20 & 45 & \\ 30 & 80 & 171\end{bmatrix}=\underbrace{\begin{bmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 3 & 4 & 1\end{bmatrix}}_{\mathbf{L}}\underbrace{\begin{bmatrix}10 & 0 & 0\\ 0 & 5 & 0\\ 0 & 0 & 1\end{bmatrix}}_{\mathbf{D}}\underbrace{\begin{bmatrix}1 & 2 & 3\\ 0 & 1 & 4\\ 0 & 0 & 1\end{bmatrix}}_{\mathbf{L}^{\mathbf{T}}} \qquad (1.3\text{-}1)$$

Une fois cette décomposition effectuée, la résolution du problème est grandement facilitée. Elle ne s'exprime plus que sous forme des résolutions linéaires les plus simples qui soient: à base de matrices triangulaires ou diagonales. Ce sont les fameuses «**descente-remontées»** (‘**forward/backward algorithms**’). Par exemple dans le cas d'une factorisation $\mathbf{L}\,\mathbf{U}$, le système (1.1-1) sera résolu par

$$\left.\begin{array}{l}\mathbf{Ku}=\mathbf{f}\\ \mathbf{PK}=\mathbf{LU}\end{array}\right\rangle\Rightarrow\begin{array}{l}\mathbf{Lv}=\mathbf{Pf}\ \ (\text{descente})\\ \mathbf{Uu}=\mathbf{v}\ \ (\text{remontée})\end{array} \qquad (1.3\text{-}2)$$

Dans le premier système diagonal inférieur (descente), on détermine le vecteur solution intermédiaire $\mathbf{v}$. Ce dernier sert alors de second membre au système diagonal supérieur (remontée) dont est solution le vecteur $\mathbf{u}$ qui nous intéresse.

Cette phase est peu coûteuse (en dense, de l'ordre de N^2 contre N^3 pour la factorisation[14] avec N la taille du problème) et peut donc être répétée de nombreuse fois en conservant la même factorisée. Ce qui est très utile lorsqu'on résout un problème de type **multiples seconds membres** ou lorsqu'on souhaite effectuer des **résolutions simultanées**.

Dans le **premier scénario** , la matrice $\mathbf{K}$ est fixée et on change successivement de second membre $\mathbf{f}_i$ pour calculer autant de solution $\mathbf{u}_i$ (les résolutions sont interdépendantes). Cela permet de mutualiser et donc d'amortir ce coût initial de la factorisation. Cette stratégie est abondamment utilisée, notamment dans *Code_Aster*: boucle non linéaire avec réactualisation périodique (ou pas de réactualisation) de la matrice tangente (par ex. opérateur *Aster* STAT_NON_LINE), méthodes de sous-espaces ou de la puissance inverse (sans accélération de Rayleigh) en calcul modal (CALC_MODES), chaînage thermo-mécanique avec des caractéristiques matériaux indépendantes de la température (MECA_STATIQUE), …

Dans le **second scénario**, on a connaissance de tous les $\mathbf{f}_i$ en même temps et on organise, par blocs, les phases de descente-remontée, pour calculer simultanément les solutions $\mathbf{u}_i$ indépendantes. On peut ainsi utiliser des routines d'algèbre linéaire de haut niveau plus efficaces, voire même jouer sur les consommations

14 En dense, Coppersmith et Winograd (1982) ont montré qu'on pouvait, au mieux, diminuer cette complexité algorithmique à CN^{α} avec $\alpha=2{,}49$ et C constante (pour des N grands).

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

mémoire en stockant le vecteur $\mathbf{f_i}$ en creux. Cette stratégie est (partiellement) utilisée dans *Code_Aster* , par exemple, dans la construction des compléments de Schur des algorithmes de contact-frottement ou pour la sous-structuration.

Remarque:

• Le produits MUMPS prévoie ces deux types de stratégie et propose même des fonctionnalités pour faciliter la construction et la résolution de complément de Schur.

Examinons maintenant le **processus de factorisation en lui-même**. Il est déjà clairement explicité dans les autres documentations théoriques de *Code_Aster* sur le sujet[R6.02.01][R6.02.02], ainsi que dans les références bibliographiques déjà citées[Duf06][Gol96][Las98]. Aussi nous ne le détaillerons pas. Précisons juste que c'est un processus itératif organisé schématiquement autour de trois boucles: l'une «dite en i » (sur les lignes de la matrice de travail), la seconde «en j » (resp. colonnes) et la troisième «en k » (resp. étapes de la factorisation). Elles construisent itérativement une nouvelle matrice $\tilde{\mathbf{A}}_{k+1}$ à partir de certaines données de la précédente, $\tilde{\mathbf{A}}_k$, *via* la formule classique de factorisation qui s'écrit formellement:

$$\begin{array}{c}\textbf{Boucles en } i,j,k \\ \tilde{\mathbf{A}}_{k+1}(i,j) := \tilde{\mathbf{A}}_k(i,j) - \dfrac{\tilde{\mathbf{A}}_k(i,k)\tilde{\mathbf{A}}_k(k,j)}{\tilde{\mathbf{A}}_k(k,k)}\end{array} \qquad (1.3\text{-}3)$$

Initialement le processus est activé avec $\tilde{\mathbf{A}}_0 = \mathbf{K}$ et à la dernière étape, on récupère dans la matrice carrée $\tilde{\mathbf{A}}_N$ les parties triangulaires ($\mathbf{L}$ et/ou $\mathbf{U}$) voire diagonale ($\mathbf{D}$) qui nous intéressent. Par exemple, dans le cas $\mathbf{L}\,\mathbf{D}\,\mathbf{L}^{\mathbf{T}}$:

$$\begin{array}{c}\textbf{Boucles en } i,j \\ \text{si } i<j: \ \mathbf{L}(i,j) = \tilde{\mathbf{A}}_N(i,j) \\ \text{si } i=j: \ \mathbf{D}(i,j) = \tilde{\mathbf{A}}_N(i,j)\end{array} \qquad (1.3\text{-}4)$$

Remarque:

• La formule (1.4-3) *contient en germe les problèmes inhérents aux méthodes directes: en stockage creux, le fait que le terme $\tilde{\mathbf{A}}_{k+1}(i,j)$ peut devenir non nul alors que $\tilde{\mathbf{A}}_k(i,j)$ l'est (notion de remplissage de la factorisée, 'fill-in', impliquant donc une renumérotation ou 'ordering'); la propagation d'erreur d'arrondis ou la division par zéro via le terme $\tilde{\mathbf{A}}_k(k,k)$ (notion de pivotage et d'équilibrage des termes de la matrice ou 'scaling').*

1.4 Méthodes directes: les différentes approches

L'ordre des boucles i, j et k n'est pas figé. On peut les permuter et effectuer les mêmes opérations mais dans un ordre différent. Cela définit ainsi six variantes $kij, kji, ikj \ldots$ qui vont manipuler différentes zones de la matrice courante $\tilde{\mathbf{A}}_k$: «zone des nouveaux termes calculés» *via* (1.3-2), «zone déjà calculée et utilisée» dans (1.3-2), «zone déjà calculée et inutilisée» et «zone non encore calculée». Par exemple, dans la variante jik , on a le schéma de fonctionnement suivant pour j fixé

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

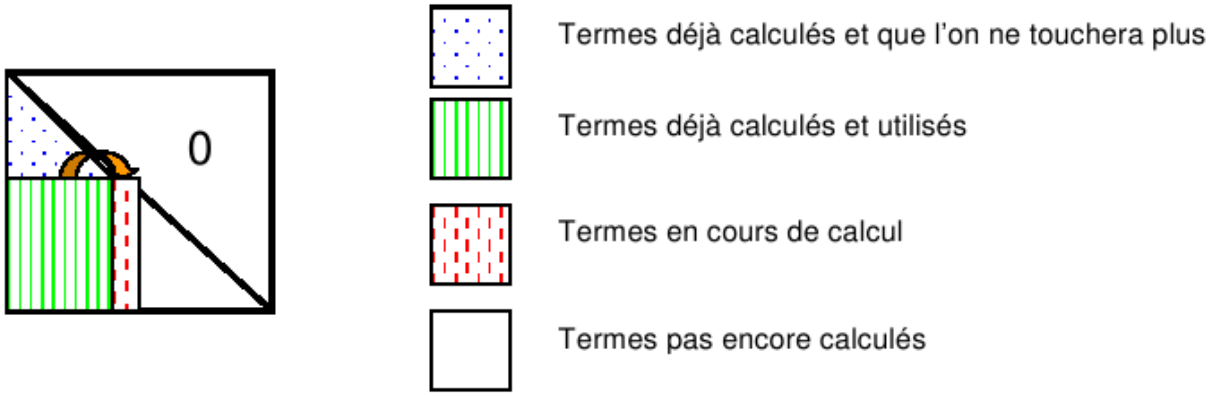


Figure 1.4-1._ Schéma de construction d'une factorisation « jik » ('right looking').

Remarques:

• La méthode $\mathbf{LDL^T}$ de Code_Aster (`SOLVEUR/METHODE='LDLT'`) est une factorisation « ijk », les multifrontales de C.Rose (`…='MULT_FRONT'`) et de MUMPS (`…='MUMPS'`) sont elles orientées colonnes (« kji »).

• Certaines variantes portent des noms particuliers: algorithme de Crout (« jki ») et celui de Doolitle (« ikj »).

• Dans les papiers on utilise souvent la terminologie anglo-saxonne désignant l'orientation des manipulations matricielles plutôt que l'ordre des boucles: 'looking forward method', 'looking backward method', 'up-looking', 'left-looking', ''right-looking', 'left-right-looking'…

Toutes **ces variantes se déclinent** encore suivant:

- Que l'on exploite certaines propriétés de la matrice (symétrie, définie-positivité, bande…) ou que l'on cherche le périmètre d'utilisation le plus large;
- Que l'on effectue des traitements scalaires ou par blocs;
- Que la décomposition en blocs soit déterminée par des aspects mémoires (cf. méthode $\mathbf{LDL^T}$ paginée de *Code_Aster*) ou plutôt liée aux indépendances des tâches ultérieures (*via* un graphe d'élimination cf multifrontale [R6.06.02] §1.3);
- Que l'on réintroduit des termes nuls dans les blocs pour faciliter l'accès aux données[15] et susciter des opérations algébriques très efficaces, souvent *via* des BLAS3[16] (cf. multifrontale de C.Rose et MUMPS);
- Que l'on groupe les contributions affectant un bloc de lignes/colonnes (approche 'fan-in', cf. PaStiX) ou qu'elles soient appliquées au plus tôt ('fan-out');
- Qu'en parallélisme, on cherche à ménager différents niveaux de séquences de tâches indépendantes, qu'on les ordonnance statiquement ou dynamiquement, qu'on recouvre du calcul par de la communication…
- Que l'on applique des pré et post-traitements pour réduire le remplissage et améliorer la qualité des résultats: renumérotation des inconnues, mise à l'échelle des termes de la matrice, pivotage partiel (ligne) ou total (ligne et colonne), scalaire ou par blocs diagonaux, raffinement itératif…

Pour **les regrouper on distingue souvent quatre catégories**:

- Les **algorithmes classiques**: Gauss, Crout, Cholesky, Markowitz (*Matlab, Mathematica, Y12M* …) ;
- Les méthodes **frontales** (*MA62* …);
- Les méthodes **multifrontales** (*MULT_FRONT Aster, MUMPS, SPOOLES, TAUCS, UFMPACK, WSMP* …)
- Les **supernodales** (*SuperLU, PaStiX, CHOLMOD, PARDISO* …).

15 Ce compromis creux/dense permet de diminuer les adressages indirects aux données et ainsi de mieux utiliser la hiérarchie mémoire des machines actuelles.

16 Le ratio «calcul/accès mémoire» des Blas niveau 3 (produit matrice/matrice) est N fois meilleur (avec N la taille du problème) que celui des autres niveaux de Blas. Il est aussi souvent supérieur à celui de routines «faites à la main» non optimisées sur ces aspects «localité des données/hiérarchie mémoire».

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

1.5 Méthodes directes: les principales étapes

Lorsqu'on traite des systèmes creux, la phase de factorisation numérique (1.3-3) ne s'applique pas directement à la matrice initiale $\mathbf{K}$, mais à **une matrice de travail** $\mathbf{K}_{\text{travail}}$ résultant d'une **phase de prétraitements** . Et ce **afin de réduire le remplissage, d'améliorer la précision des calculs** et donc d'optimiser les coûts ultérieurs en CPU et en mémoire. Grossièrement cette matrice de travail peut s'écrire sous la forme du produit matriciel suivant

$$\mathbf{K}_{travail} := \mathbf{P}_o \ \mathbf{D}_r \ \mathbf{K} \ \mathbf{Q}_c \ \mathbf{D}_c \ \mathbf{P}_o^T \tag{1.5-1}$$

dont nous allons décrire les différents éléments par la suite. On peut ainsi décomposer le fonctionnement d'un solveur direct en quatre étapes:

1) *Prétraitements et factorisation symbolique*: elle intervertie l'ordre des colonnes de la matrice de travail (*via* une matrice de permutation $\mathbf{Q_c}$) afin d'éviter les divisions par zéro du terme $\tilde{\mathbf{A}}_k(k,k)$ et de réduire le remplissage. De plus elle rééquilibre les termes afin de limiter les erreurs d'arrondi (*via* les matrices de mise à l'échelle $\mathbf{D_r}/\mathbf{D_c}$). Cette phase peut être aussi **cruciale pour l'efficacité algorithmique** (gain d'un facteur 10 parfois constaté) et la qualité des résultats (gain de 4 ou 5 décimales).
   Dans cette phase, on créé aussi les structures de stockage de la matrice factorisée creuse et des auxiliaires (pivotage dynamique, communication…) requis par les phases suivantes. De plus, on estime l'arbre de dépendance des tâches, leur répartition initiale selon les processeurs et les consommations mémoires totales prévues.

2) *L'étape de renumérotation*: elle interverti l'ordre des inconnues de la matrice (*via* la matrice de permutation $\mathbf{P_o}$) afin de réduire le remplissage qu'implique la factorisation. En effet, dans la formule (1.3-3) on voit que la factorisée ($\tilde{\mathbf{A}}_{k+1}(i,j) \neq 0$) peut contenir un nouveau terme non nul dans son profil alors que la matrice initiale n'en comportait pas ($\tilde{\mathbf{A}}_k(i,j) = 0$) . Du fait du terme $\dfrac{\tilde{\mathbf{A}}_k(i,k)\tilde{\mathbf{A}}_k(k,j)}{\tilde{\mathbf{A}}_k(k,k)}$ non nécessairement nul. En particulier, il est non nul lorsqu'on peut trouver des termes non nuls de la matrice initiale du type $\tilde{\mathbf{A}}_k(i,l) \text{ ou } \tilde{\mathbf{A}}_k(l,j)\ (l<i \text{ et } l<j)$. Ce phénomène peut conduire à des surcoûts mémoire et calcul très importants (la factorisée peut être 100 fois plus grosse que la matrice creuse initiale !).
   D'où l'idée de renuméroter les inconnues (et donc de permuter les lignes et les colonnes de $\mathbf{K}$) afin de freiner ce phénomène qui est le **vrai «talon d'achille» des directs** . Pour ce faire, on fait souvent appel à des produits externes ((PAR)METIS, (PT)SCOTCH, CHACO, JOSTLE, PARTY…) ou à des heuristiques embarquées avec les solveurs (AMD, RCMK…). Bien sûr, ces produits affichent des performances différentes suivant les matrices traitées, le nombre de processeurs… Parmi eux, **METIS et SCOTCH sont très répandus et «sortent souvent du lot»** (gain jusqu'à 50%).

3) *La phase de factorisation numérique* : elle met en œuvre la formule (1.3-3) *via* les méthodes entrevues au paragraphe §1.4 précédent. C'est la phase, de loin, **la plus coûteuse** qui va construire explicitement les factorisations creuses $\mathbf{L}\mathbf{L}^{\mathbf{T}}, \mathbf{L}\mathbf{D}\mathbf{L}^{\mathbf{T}}$ ou $\mathbf{L}\mathbf{U}$.

4) *La phase de résolution*: elle effectue les descentes-remontées (1.3-2) dont «jaillit» (enfin !) la solution $\mathbf{u}$. Elle est **peu coûteuse** et **mutualise éventuellement une factorisation numérique ultérieure** (multiples seconds membres, résolutions simultanées, redémarrage de calcul…).

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

Remarques:

• Les étapes 1 et 2 ne requièrent que la connaissance de la connectivité et du graphe de la matrice initiale. Donc finalement que des données stockables et manipulables sous forme d'entiers. Seules les deux dernières étapes utilisent des réels sur les termes effectifs de la matrice. Elles n'ont besoin des termes de la matrice que si les étapes de scaling sont enclenchées (calcul de $\mathbf{D}_r / \mathbf{D}_c$ *).*

• Les étapes 1 et 4 sont indépendantes tandis que les 1, 2 et 3, a contrario, sont liées. Suivant les produits/approches algorithmiques, on les agglomère différemment: 1 et 2 sont liés dans MUMPS, 1 et 3 dans SuperLu et 1, 2 et 3 dans UFMPACK… MUMPS permet d'effectuer séparément mais successivement les étapes 1+2, 3 et 4, voire de mutualiser leurs résultats pour effectuer divers séquences. Pour l'instant, dans Code_Aster, on alterne les séquences 1+2+3 et 4, 4, 4… et à nouveau 1+2+… (cf. chapitre suivant).

• Certains produits proposent de tester plusieurs stratégies dans une ou plusieurs étapes et choissent la plus adaptée: SPOOLES et WSMP pour l'étape 1, TAUCS pour l'étape 3 etc.

• Les outils de renumérotation de la première phase sont basés sur des concepts très variés : méthodes d'ingénieurs, techniques géométriques ou d'optimisation, théorie des graphes, théorie spectrale, méthodes tabou, algorithmes évolutionnaires, ceux mémétics, ceux basés dits de «colonies de fourmis», réseaux neuronaux… Tous les coups sont permis pour améliorer l'optimum local sous la forme duquel s'exprime la problématique du renuméroteur. Ces outils servent aussi souvent à partitionner/distribuer des maillages (cf. [R6.01.03] §6). Pour l'instant, Code_Aster utilise les renuméroteurs METIS/AM/AMD (pour `METHODE=' MULT_FRONT'` *et '* `MUMPS'`*), AMF/QAMD/PORD (pour '* `MUMPS'` *) et RCMK*[17] *(pour* `'GCPC', 'LDLT'` *et* `'PESTC'`*).*

• Quelque soit le solveur linéaire[18] *utilisé, Code_Aster effectue une phase de factorisation préliminaire (étape 0) pour décrire les inconnues du problème (lien degré de liberté physique ou tardif/numéro de ligne de la matrice via la structure de données* `NUME_DDL`*) et prévoir le stockage ad hoc du profil MORSE de la matrice .*

1.6 Méthodes directes: les difficultés

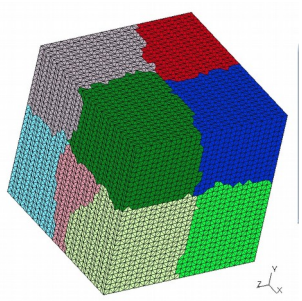
Parmi les difficultés que doivent surmonter les «méthodes directes creuses» on trouve:

- La **manipulation de structures de données complexes** qui optimisent le stockage (cf. profil de la matrice) mais qui complexifie l'algorithmique (cf. pivotage, OOC…). Cela contribue a abaisser le ratio «calcul/accès aux données».
- La **gestion efficace des données vis-à-vis de la hiérarchie mémoire** et la bascule IC/OOC[19]. Cette une question récurrente à beaucoup de problèmes, mais qui ici est prégnante du fait de la forte consommation calcul.
- La gestion du **compromis creux/dense** (pour les méthodes par fronts) vis-à-vis de la consommation mémoire, de la facilité d'accès aux données et de l'efficacité des briques élémentaires d'algèbre linéaire.
- Le choix de la **bonne renumérotation**: c'est un problème NP-complet ! Pour les problèmes de grandes tailles, on ne peut trouver en un temps «raisonnable» la renumérotation optimale. On doit se contenter d'une solution «locale».
- La gestion effective de la **propagation des erreurs d'arrondi** *via* le scaling, le pivotage et les calculs d'erreurs sur la solution (erreur directe/inverse[20] et conditionnement).
- La **taille de la factorisée qui est souvent le «goulet d'étranglement» n°1** . Sa répartition entre processeurs (*via* le parallélisme distribué) et/ou l'OOC ne permettent pas toujours de surmonter cet écueil (cf. figure 1.6-1).

17 Pour minilmiser le remplissage, la factorisation incomplète est utilisée comme préconditionneur de ces solveurs itératifs.
18 Parmi `'MULT_FRONT'/'LDLT'/'MUMPS'/'GCPC'/'PESTC'`.
19 IC pour In-Core (toutes les structures de données sont en RAM) et OOC pour Out-Of-Core (certaines sont basculées sur disque).
20 Référencées souvent sous le vocable anglo-saxon: 'forward/backward errors'.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

N =0.21M
nnz =8M (x38)
K $^{-1}$ =302M
(x38 METIS)





N =0.8M
nnz =28M (x35)
K $^{-1}$ =397M
(x15 METIS)

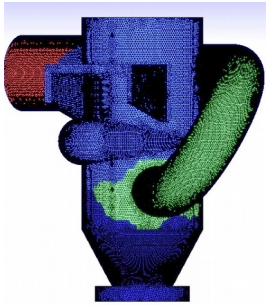


Figure 1.6-1._ Le «boulet» des solveurs directs creux: la taille de la factorisée.

Cette figure reprend deux exemples: un cas-test canonique (cube) et une étude industrielle (pompe RIS). Avec les notations suivantes: M pour millions de termes, N la taille du problème, nnz le nombre de termes non nuls de la matrice et $\mathbf{K}^{-1}$ celui de sa factorisée renumérotée *via* METIS. Le surfacteur lorsqu'on passe de l'une à l'autre est noté entre parenthèses.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

2 Le package MUMPS

2.1 Historique

Le package MUMPS implémente une multifrontale «massivement» parallèle (‘**MUltifrontal Massively Parallel sparse direct Solver** ’[Mum]) mise au point durant le Projet européen PARASOL (1996-1999) par les équipes de trois laboratoires: CERFACS, ENSEEIHT-IRIT et RAL (I.S.Duff, P.R.Amestoy, J.Koster et J.Y.L’Excellent…).

Depuis cette version finalisée (MUMPS 4.04 22/09/99) et public (libre de droit), une quarantaine d’autres versions ont été livrées. Ces développements corrigent des anomalies, étendent le périmètre d’utilisation, améliorent l’ergonomie et surtout, enrichissent les fonctionnalités. MUMPS est donc un produit pérenne, développé et maintenu par une dizaine de personnes appartenant à des entités académiques réparties entre Bordeaux, Lyon et Toulouse: IRIT, CNRS, CERFACS, INRIA/ENS Lyon et Université de Bordeaux I.

Figure 2.1-1._ Logos des principaux contributeurs à MUMPS [Mum].

Le produit est public et téléchargeable sur son site web: **http://graal.ens-lyon.fr/MUMPS**. On recense environ **1000 utilisateurs directs** (dont 1/3 Europe + 1/3 USA) sans compter ceux qui l’utilisent *via* les librairies qui le référence: PETSc, TRILINOS, Matlab et Scilab... Son site propose de la documentation (théorique et d’utilisation), des liens, des exemples d’application, ainsi qu’un forum de discussion (en anglais) traçant le retour d’expérience sur le produit (bugs, problèmes d’installation, conseils…).
Chaque année une dizaine de travaux algorithmiques/informatiques aboutissent à des améliorations du package (thèse, post-doc, travaux de recherche…). D’autre part il est utilisé régulièrement pour des études industrielles (EADS, CEA, BOEING, GéoSciences Azur, SAMTECH, *Code_Aster*…).

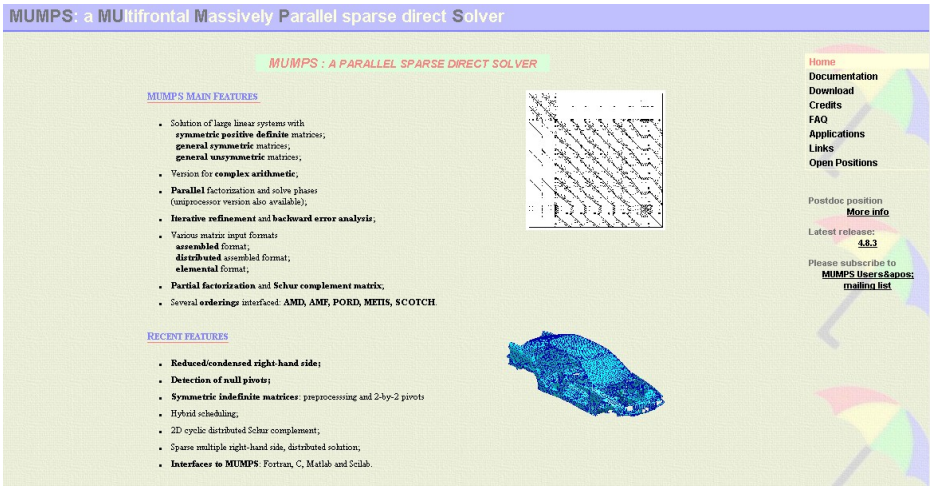


Figure 2.1-2._ La page d’accueil du site web de MUMPS [Mum].

Depuis 2015, **un consortium entre équipes académiques, industriels et éditeurs de logiciels s'est monté autour du produit** : le consortium MUMPS[MUC]. Il doit permettre de mieux assurer son développement, sa diffusion et sa pérennité.
Il géré par l'INRIA et regroupe fin 2015: EDF, ALTAIR, MICHELIN, LSTC, SIEMENS, ESI, TOTAL (membres) et CERFACS, INPT, INRIA, ENS Lyon et Université de Bordeaux (membres fondateurs).

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

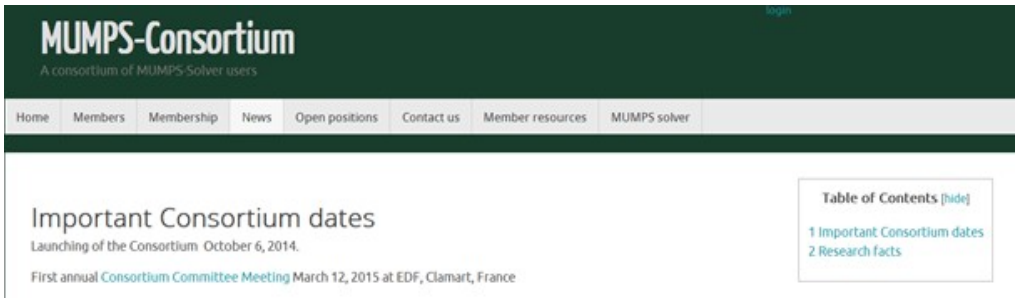


Figure 2.1-3._ La page d'accueil du site web dédié au consortium[MUC].

2.2 Principales caractéristiques

MUMPS implémente une **multifrontale** [ADE00][ADKE01][AGES06] effectuant une factorisation $\mathbf{LU}$ ou $\mathbf{LDL}^{\mathbf{T}}$ (cf. § 1.4). Ses principales caractéristiques sont:

- **Large périmètre d'utilisation**: SPD, symétrique quelconque, non symétrique, réel/complexe, simple/double précision, matrice régulière/singulière (tout ce périmètre est exploité dans *Code_Aster*);
- Admet **trois modes de distribution des données**: par élément, centralisé ou distribué (les deux derniers modes sont exploités dans *Code_Aster*);
- **Interfaçage** en Fortran (exploité), C, matlab et scilab.
- **Paramétrage par défaut** et possibilité de laisser le package choisir certaines de ses options en fonction du type de problème, de sa nature et du nombre de processeurs (exploité).
- **Modularité** (3 phases distinctes interchangeables cf. §1.5 et figure 2.2.1) et ouverture de certaines arcanes numériques de MUMPS. L'utilisateur (très) avancé peut ainsi sortir du produit le résultat de certains prétraitements (scaling, pivotage, renumérotation), les modifier ou les remplacer par d'autres et les réinsérer dans la chaîne de calculs propre à l'outil;
- Différentes **stratégies de résolutions**: multiples seconds membres, résolutions simultanées et compléments de Schur (seules les deux premières sont exploitées) ;
- Différents **renumeroteurs** embarqués ou externes: (PAR)METIS, AMD, QAMD, AMF, PORD, (PT)SCOTCH, 'fourni par l'utilisateur' (exploités sauf le dernier);
- **Fonctionnalités connexes**: détection de petits pivots/calcul de rang (exploités), calcul de noyaux (bientôt exploité), analyse d'erreur et amélioration de la solution (exploitée), calcul du critère d'inertie pour le test Sturm en calcul modal (exploité), compression low-rank (exploitée); usage du caractère creux du second membre et résolution simultanées de plusieurs systèmes linéaires (pas exploité), calcul de quelques termes de l'inverse (pas exploité), procédure de rédémarrage (pas encore exploité), manipulation des entiers longs (pas encore exploité) ?
- **Pré et post-traitements**: mise à l'échelle, permutation ligne/colonne et scalaire/bloc 2x2, raffinement itératif (exploités);
- **Parallélisme** : potentiellement à 2 niveaux (MPI+OpenMP), gestion asynchrone des flots de tâches/données et leur réordonnancement dynamique, recouvrement calcul/communication; Distribution des données associée à la distribution des tâches; Ce parallélisme ne commence qu'au niveau de la phase de factorisation, sauf si l'un des renumeroteurs parallèles (PARMETIS ou PTSCOTCH) a été choisis.
- **Mémoire** : déchargement sur disque ou non de la factorisée (modes In-Core ou Out-Of-Core) avec estimation préalable des consommations RAM par processeur dans les deux cas (exploité).

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

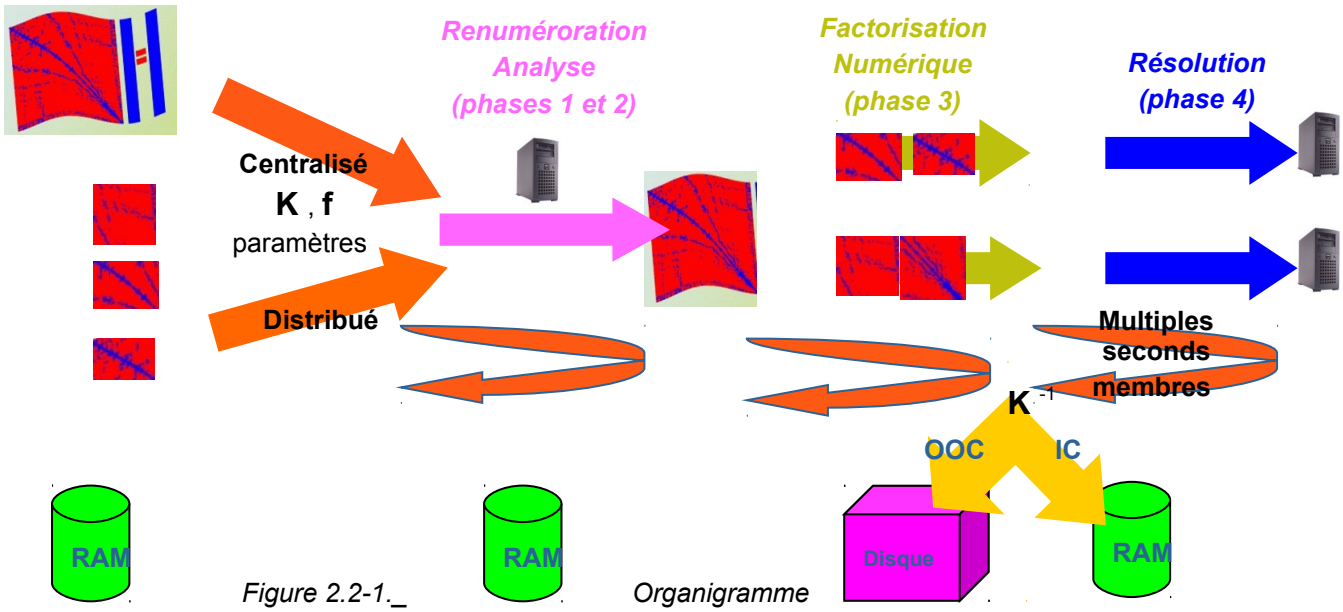


Figure 2.2-1._ Organigramme fonctionnel de MUMPS: ses trois étapes en parallèle centralisé/distribué et IC/OOC.

Remarque:

• En terme de parallélisme, MUMPS exploite deux niveaux (cf. [R6.01.03] § 2.6.1): l'un externe lié à l'élimination concurrente de fronts (via MPI), l'autre interne, au sein de chaque front (via des BLAS «threadées» ou autour des algorithmes de compression low-rank).

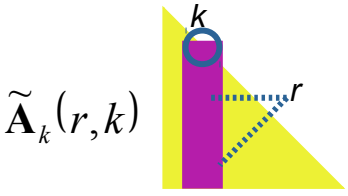
• La méthode multifrontale native de Code_Aster n'exploite que le second niveau et en parallélisme à mémoire partagée (via OpenMP). Donc sans recouvrement, réordonnancement dynamique et distribution des données entre les processeurs. Par contre, par ses connections fines avec Code_Aster, elle exploite toutes les facilités du gestionnaire mémoire JEVEUX (OOC, redémarrage, diagnostic…) et les spécificités de modélisation du code (éléments de structures, Lagranges).

2.3 Zooms sur quelques points techniques

2.3.1 Pivotage

La technique de pivotage consiste à choisir un terme $\tilde{\mathbf{A}}_k(k,k)$ adapté (dans la formule 1.4-3) pour éviter de diviser par un terme trop petit (ce qui amplifierait la propagation des erreurs d'arrondi lors du calcul des termes $\tilde{\mathbf{A}}_{k+1}(i,j)$ suivants). Pour ce faire, on permute des lignes (pivotage partiel) et/ou des colonnes (resp. total) pour trouver le dénominateur de (1.4-3) adapté. Par exemple, dans le cas d'un pivotage partiel, on choisit comme «pivot» le terme $\tilde{\mathbf{A}}_k(r,k)$ tel que

$$\tilde{\mathbf{A}}_k(r,k) \geq u \max_i |\tilde{\mathbf{A}}_k(i,k)| \quad \text{avec } u \in]0,1] \tag{2.3-1}$$



$\tilde{\mathbf{A}}_k(r,k)$

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

F igure 2.3-1._ Choix du pivot partiel à l'étape k .

D'où une amplification des erreurs d'arrondi au maximum de ($1+1/u$) à cette étape. **Ce qui est important ici ce n'est pas tant de choisir le terme le plus grand possible en valeur absolu** (u=1) **que d'éviter de choisir le plus petit !** L'inverse de ces pivots intervient aussi lors de la phase de descente-remontée, donc il faut ménager ces deux sources d'amplification d'erreurs en choisissant un u médian. MUMPS, comme beaucoup de package, propose par défaut u=0.01 (paramètre MUMPS `CNTL(1)`).
Pour pivoter on utilise généralement des termes diagonaux scalaires mais aussi des blocs de termes (des blocs diagonaux 2x2).
Dans MUMPS, deux types de pivotage sont mis en œuvre, l'un dit 'statique' (lors de la phase d'analyse), l'autre dit 'numérique' (resp. factorisation numérique). Ils sont paramétrables et activables séparément (cf. paramètres MUMPS `CNTL(1)`, `CNTL(4)` et `ICNTL(6)`). Pour des matrices SPD ou à diagonale dominante, ces facultés de pivotage peuvent être désactivées sans risque (le calcul y gagnera en rapidité), par contre, dans les autres cas de figure, il faut les initialiser pour gérer les éventuels pivots très petits ou nuls. Cela implique en général un surcroît de remplissage de la factorisée mais accroît la stabilité numérique.

Remarques:

• Cette fonctionnalité de pivotage rend MUMPS indispensable pour traiter certaines modélisations de Code_Aster (éléments quasi-incompressibles, formulations mixtes, X-FEM…). Du moins tant que d'autres solveurs directs incluant du pivotage ne seront pas disponibles dans le code.

• L'utilisateur Aster n'a pas accès directement au paramétrage fin de ces facultés de pivotage. Elles sont activées avec les valeurs par défaut. Il peut juste les débrancher partiellement en posant `SOLVEUR/PRETRAITEMENTS='SANS'` (par défaut=`'AUTO'`).

• Le surcroît de remplissage du au pivotage numérique doit s'ordonnancer au plus tôt dans MUMPS (dès la phase d'analyse). Et ce, en prévoyant arbitrairement un pourcentage de surconsommation mémoire par rapport au profil prévu. Ce chiffre doit être renseigné en pourcents dans le paramètre MUMPS `ICNTL(14)` (accessible à l'utilisateur Aster via le mot-clé `SOLVEUR/PCENT_PIVOT` initialisé par défaut à 20%). Par la suite, si cette évaluation s'avère insuffisante, suivant le type de gestion mémoire choisie (mot-clé `SOLVEUR/GESTION_MEMOIRE`), soit le calcul s'arrête en `ERREUR_FATALE`, soit on retente plusieurs fois une factorisation numérique en doublant à chaque fois la taille de cet espace réservé au pivotage.

• Certains produits restreignent leur périmètre/robustesse en ne proposant pas de stratégie de pivotage (SPRSBLKKT, MULT_FRONT_Aster…), d'autres se limitent à des pivots scalaires (CHOLMOD, PaStiX, TAUCS, WSMP…) ou proposent des stratégies particulières (méthode de perturbation+correction de Bunch-Kaufmann pour PARDISO, de Bunch-Parlett pour SPOOLES…).

2.3.2 Raffinement itératif

En fin de résolution, ayant obtenu la solution $\mathbf{u}$ du problème, on peut évaluer facilement son résidu $\mathbf{r} := \mathbf{Ku} - \mathbf{f}$. Connaissant déjà la factorisée de la matrice, ce résidu peut alors alimenter à peu de frais **le processus itératif d'amélioration suivant** (dans le cas général non symétrique):

$$\begin{aligned} &\text{Boucle en } i \\ &(1)\quad \mathbf{r}^i = \mathbf{f} - \mathbf{K}\mathbf{u}^i \\ &(2)\quad \mathbf{LU}\ \delta\mathbf{u}^i = \mathbf{r}^i \\ &(3)\quad \mathbf{u}^{i+1} \Leftarrow \mathbf{u}^i + \delta\mathbf{u}^i \end{aligned} \qquad (2.3\text{-}2)$$

Ce processus est «indolore»[21] puisqu'il ne coûte principalement que le prix des descente-remontées de l'étape (2). Il peut ainsi s'itérer jusqu'à un certain seuil ou jusqu'à un nombre d'itérations maximum. Si le calcul de résidu ne comporte pas trop d'erreur d'arrondi, c'est-à-dire si l'algorithme de résolution est plutôt fiable (cf.

21 C'est vrai lorsque MUMPS fonctionne en mode de gestion mémoire In-Core et en séquentiel. Par contre, lorsque les données sont distribuées entre les processeurs et entre les mémoires RAM et disque (parallélisme et Out-Of-Core sont activés), cette étape peut être un peu coûteuse.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

paragraphe suivant) et que le conditionnement du système matriciel est bon, ce processus de raffinement itératif[22] est très bénéfique sur la qualité de la solution.
Dans MUMPS ce processus est activable ou non (paramètre ICNTL(10)<0) et borné par un nombre d'itérations maximum N_{err} (ICNTL(10)). Le processus (2.3-2) se poursuit tant que le «résidu équilibré» $\mathbf{B}_{err}$ est supérieur à un seuil paramétrable *seuil* (CNTL(2), fixé par défaut à $\sqrt{\varepsilon}$ avec ε précision machine)

$$\mathbf{B}_{err} := \max_j \frac{|\mathbf{r}_j^i|}{\left(|\mathbf{K}||\mathbf{u}^i|+|\mathbf{f}|\right)_j} < seuil \tag{2.3-3}$$

ou qu'il ne décroît pas d'un facteur au moins 5 (non paramétrable). En général, une ou deux itérations suffisent. Si ce n'est pas le cas, c'est souvent révélateur d'autres problèmes: mauvais conditionnement ou erreur inverse (cf. paragraphe suivant).

Remarques:

• Pour l'utilisateur Code_Aster ces paramètres MUMPS ne sont pas directement accessibles. La fonctionnalité est activable ou non via le mot-clé POSTTRAITEMENTS .

• Cette fonctionnalité est présente dans de nombreux packages: Oblio, PARDISO, UFMPACK, WSMP…

2.3.3 Fiabilité des calculs

Pour estimer la qualité de la solution d'un système linéaire[ADD89][Hig02], MUMPS propose des outils numériques déduits de la théorie de **l'analyse inverse des erreurs d'arrondi initiée par Wilkinson** (1960). Dans cette théorie, les erreurs d'arrondi dues à plusieurs facteurs (troncature, opération en arithmétique finie..) sont assimilées à des perturbations sur les données initiales. Cela permet de les comparer à d'autres sources d'erreurs (mesure, discrétisation…) et de les manipuler plus facilement *via* trois indicateurs obtenus en post-traitement:

- *Le conditionnement* $cond(\mathbf{K},\mathbf{f})$: il mesure la **sensibilité du problème aux données** (problème instable, mal formulé/discrétisé…). C'est-à-dire, le facteur multiplicatif que la manipulation des données va opérer sur le résultat. Pour l'améliorer, on peut essayer de changer la formulation du problème ou d'équilibrer les termes de la matrice, en dehors de MUMPS ou *via* MUMPS (SOLVEUR/PRETRAITEMENTS='OUI' dans *Code_Aster*).
- *L'erreur inverse* $be(\mathbf{K},\mathbf{f})$ ('backward error'): elle mesure la **propension de l'algorithme de résolution à transmettre/amplifier les erreurs d'arrondi**. Un outil est dit «fiable» lorsque ce chiffre est proche de la précision machine. Pour l'améliorer, on peut essayer de changer d'algorithme de résolution ou de modifier une ou plusieurs de ses étapes (dans *Code_Aster* on peut jouer sur les paramètres SOLVEUR/TYPE_RESOL, PRETRAITEMENTS et RENUM).
- *L'erreur directe* $fe(\mathbf{K},\mathbf{f})$ ('forward error'): elle est le produit des deux chiffres précédents et fournit un **majorant de l'erreur relative sur la solution**.

$$\frac{\|\delta\mathbf{u}\|}{\|\mathbf{u}\|} < \underbrace{cond(\mathbf{K},\mathbf{f})\times be(\mathbf{K},\mathbf{f})}_{fe(\mathbf{K},\mathbf{f})} \tag{2.3-4}$$

On peut donner une représentation graphique (cf. figure 2.3-2) de ces notions en exprimant **l'erreur inverse** comme l'écart entre «les données initiales et les données perturbées», tandis que **l'erreur directe** mesure l'écart entre «la solution exacte et la solution réellement obtenue» (celle du problème perturbé par les erreurs d'arrondi).

22 On parle aussi «d'amélioration itérative» ('iterative refinement').

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

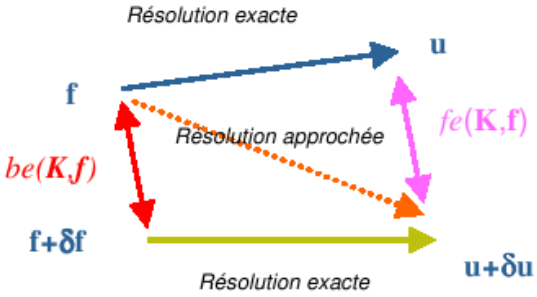


Figure 2.3-2._ Représentation graphique de la notion d'erreurs directe et inverse.

Dans le cadre des systèmes linéaires, l'erreur inverse se mesure *via* le **résidu équilibré**

$$be(\mathbf{K},\mathbf{f}) := \max_{j\in J} \frac{|\mathbf{f}-\mathbf{K}\mathbf{u}|_j}{(|\mathbf{K}||\mathbf{u}|+|\mathbf{f}|)_j} \tag{2.3-5}$$

On ne peut pas toujours l'évaluer sur tous les indices ($J \neq [1,N]_N$). En particulier lorsque le dénominateur est très petit (et le numérateur non nul), on lui préfère la formulation (avec J^* tel que $J \cup J^* = [1,N]_N$)

$$be^*(\mathbf{K},\mathbf{f}) := \max_{j\in J^*} \frac{|\mathbf{f}-\mathbf{K}\mathbf{u}|_j}{(|\mathbf{K}||\mathbf{u}|)_j + \|\mathbf{K}_{j.}\|_\infty \|\mathbf{u}\|_\infty} \tag{2.3-6}$$

où $\boldsymbol{K}_{j.}$ représente la $j^{ième}$ ligne de la matrice $\mathbf{K}$. A ces deux indicateurs, on associe deux estimations du conditionnement matricel (l'un lié aux lignes retenues dans l'ensemble J et l'autre à son complémentaire J^*): $cond(\mathbf{K},\mathbf{f})$ et $cond^*(\mathbf{K},\mathbf{f})$ .La théorie nous fournit alors les résultats suivant:

- **La solution approchée u est la solution exacte du problème perturbé**

$$\begin{aligned} &(\mathbf{K}+\delta\mathbf{K})\,\mathbf{u} = (\mathbf{f}+\delta\mathbf{f}) \\ &\text{avec } \delta\mathbf{K}_{ij} \le \max(be, be^*)|\mathbf{K}_{ij}| \\ &\text{et } \delta\mathbf{f}_i \le \max\left(be.|\mathbf{f}_i|, be^*.\|\mathbf{K}_{i.}\|_\infty \|\mathbf{u}\|_\infty\right) \end{aligned} \tag{2.3-7}$$

- On a la majoration suivante (*via* l'erreur directe $fe(\mathbf{K},\mathbf{f})$) sur **l'erreur relative en solution**

$$\frac{\|\delta\mathbf{u}\|}{\|\mathbf{u}\|} < \underbrace{cond\times be + cond^*\times be^*}_{fe(\mathbf{K},\mathbf{f})} \tag{2.3-8}$$

En pratique, on scrute surtout cette dernière estimation $fe(\mathbf{K},\mathbf{f})$ et ses composantes. Son ordre de grandeur indique *grosso modo* le nombre de décimales «vraies» de la solution calculées. Pour les problèmes mal conditionnés, une tolérance de 10^{-3} n'est pas rare, mais elle doit être prise au sérieux car ce type de pathologie peut sérieusement perturber un calcul.

Même dans le cadre très précis de la résolution de système linéaire, il existe de nombreuses façons de définir la sensibilité aux erreurs d'arrondis du problème considéré (c'est-à-dire son conditionnement). Celle retenue par MUMPS et, qui fait référence dans le domaine (cf. Arioli, Demmel et Duff 1989), est indissociable de la 'backward error' du problème. La définition de l'un n'a pas de sens sans celle de l'autre. Il ne faut donc pas confondre ce type de conditionnement avec la notion de conditionnement matriciel classique.
D'autre part, **le conditionnement fourni pas MUMPS prend en compte le SECOND MEMBRE du système ainsi que le CARACTERE CREUX de la matrice**. En effet, ce n'est pas la peine de tenir compte d'éventuelles erreurs d'arrondis sur des termes matriciels nuls et donc non fournis au solveur ! Les degrés de liberté correspondant ne «se parlent pas» (vu de la lorgnette élément fini). Ainsi, ce conditionnement MUMPS respecte la physique du problème discrétisé. Il ne replonge pas le problème dans l'espace trop riche des matrices pleines.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

Ainsi, le chiffre de conditionnement affiché par MUMPS est beaucoup moins pessimiste que le calcul standard que peut fournir un autre produit (Matlab, Python...). Mais martelons, que ce n'est que son produit avec la 'backward error', appelée 'forward error', qui a un intérêt. Et uniquement, dans le cadre d'une résolution de système linéaire *via* MUMPS.

Remarques:

• Cette analyse de la qualité de la solution n'est pas limitée aux solveurs linéaires. Elle se décline aussi, par exemple, pour les solveurs modaux[Hig02].

• Dans MUMPS, les estimateurs fe, be, be^*, $cond$ *et* $cond^*$ *sont accessibles via, respectivement, les variables* `RINFO(9/7/8/10` *et* `11)`*. Ces post-traitements sont un peu coûteux (~jusqu'à 10% du temps calcul) et donc désactivables (via* `ICNTL(11)`*).*

• Pour l'utilisateur Code_Aster ces paramètres MUMPS ne sont pas directement accessibles. Ils sont affichés dans un encart spécifique du fichier de message (libellé «monitoring MUMPS») si on renseigne le mot-clé `INFO=2` *dans l'opérateur . D'autre part, cette fonctionnalité n'est activée que si il choisit d'estimer et de tester la qualité de sa solution via le paramètre* `SOLVEUR/RESI_RELA` *. Suivant les opérateurs Aster, ce paramètre est par défaut débranché (valeur négative) ou fixé à* 10^{-6} *. Lorsqu'il est activé (valeur positive), on teste si l'erreur directe* $fe(\mathbf{K},\mathbf{f})$ *est bien inférieure à* `RESI_RELA` *. Si cela n'est pas le cas, le calcul s'arrête en* `ERREUR_FATALE` *en précisant la nature du problème et les valeurs incriminées.*

• L'activation de cette fonctionnalité n'est pas indispensable (mais souvent utile) lorsque la solution recherchée est elle-même corrigée par un autre processus algorithmique (algorithme de Newton, schéma de Newmark): bref, dans les opérateurs linéaires `THER_LINEAIRE, MECA_STATIQUE, STAT_NON_LINE, DYNA_NON_LINE ...`

• Ce type de fonctionnalité semble peu présent dans les librairies: LAPACK, Nag, HSL…

2.3.4 Gestion mémoire (In-Core versus Out-Of-Core)

On a vu que l'inconvénient majeur (cf. §1.7) des méthodes directes réside dans la taille de la factorisée. **Pour permettre de passer en mémoire vive des systèmes plus grands, MUMPS propose de décharger cet objet sur disque**: c'est le mode **Out-Of-Core** (mot-clé `GESTION_MEMOIRE='OUT_OF_CORE'`) par opposition au mode **In-Core** (mot-clé `GESTION_MEMOIRE='IN_CORE'`) où toutes les structures de données résident en RAM (cf. figures 2.2-1 et 2.3-3). Ce mode d'économie de la RAM est complémentaire de la distribution de données qu'induit naturellement le parallélisme. La plus-value de l'OOC est donc surtout prégnante pour des nombres de processeurs modérés (<32 processeurs).
D'autre part, l'équipe MUMPS a été très attentive au surcoût CPU engendré par cette pratique. En retravaillant dans l'algorithmique du code les manipulations des entités déchargées, ils ont pu limiter au strict minimum ces surcoûts (quelques pourcents et surtout dans la phase de résolution).



Figure 2.3-3._ Deux types de gestion mémoire standards:
entièrement en RAM (`'IN_CORE'`) et RAM/disque (`'OUT_OF_CORE'`).

Ces deux modes de gestion mémoire sont «sans filet». Aucune correction ne sera opérée ultérieurement en cas de problème. Si on ne sait *a priori* pas lesquels de ces deux modes choisir et si on veux limiter, autant que faire se peut, les problèmes dus à des défauts de place mémoire, on peut choisir le mode automatique:

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

GESTION_MEMOIRE='AUTO'. Des heuristiques internes à *Code_Aster* gèrent alors toutes seules les contingences mémoire de MUMPS en fonction de la configuration informatique (machine, parallélisme) et des difficultés numériques du problème.
Dans le même ordre d'idée, une option du même mot-clé, GESTION_MEMOIRE='EVAL', permet de **calibrer les besoins d'un calcul** en affichant dans le fichier message les ressources mémoires requises par le calcul *Code_Aster*+MUMPS.

```
************************************************************************
 - Taille du système linéaire: 500000

 - Mémoire RAM minimale consommée par Code_Aster                        : 200 Mo
 - Estimation de la mémoire Mumps avec GESTION_MEMOIRE='IN_CORE'        :3500 Mo
 - Estimation de la mémoire Mumps avec GESTION_MEMOIRE='OUT_OF_CORE'    : 500 Mo
 - Estimation de l'espace disque pour Mumps avec GESTION_MEMOIRE='OUT_OF_CORE':2000 Mo

  ===> Pour ce calcul, il faut donc une quantité de mémoire RAM au minimum de
        - 3500 Mo si GESTION_MEMOIRE='IN_CORE',
        -  500 Mo si GESTION_MEMOIRE='OUT_OF_CORE'.
 En cas de doute, utilisez GESTION_MEMOIRE='AUTO'.
************************************************************************
```

Figure 2.3-4._Extrait de fichier de message avec GESTION_MEMOIRE='EVAL'.

Remarques:

• Les paramètres MUMPS ICNTL(22)/ICNTL(23) permettent de gérer ces options mémoire. L'utilisateur Aster les active indirectement via le mot clé SOLVEUR/GESTION_MEMOIRE.

• Le déchargement sur disque est entièrement contrôlé par MUMPS (nombre de fichiers, fréquence déchargement/rechargement…). On renseigne juste l'emplacement mémoire: c'est tout naturellement le répertoire de travail de l'exécutable Aster propre à chaque processeur (%OOC_TMPDIR='.'). Ces fichiers sont automatiquement effacés par MUMPS lorsqu'on détruit l'occurrence MUMPS associée. Cela évite donc un engorgement du disque lorsque différents systèmes sont factorisés dans une même résolution.

• D'autres stratégies d'OOC seraient envisageables voire sont déjà codées dans certains packages (PaStiX, Oblio, TAUCS…). On pense en particulier au fait de pouvoir moduler le périmètre des objets déchargés (cf. phase d'analyse parfois coûteuse en RAM) et de pouvoir les réutiliser sur disque lors d'une autre exécution (cf. POURSUITE au sens Aster ou factorisation partielle).

2.3.5 Gestion des matrices singulières

Un des gros point fort du produit est sa **gestion des singularités**. Il est non seulement capable de **détecter les singularités numériques**[23] d'une matrice et d'en synthétiser l'information pour un usage externe (calcul de rang, avertissement à l'utilisateur, affichage d'expertise..), mais en plus, malgré cette difficulté, il calcule une **solution «régulière**[24]**»** voire tout ou partie du **noyau associé**.

Ces nouveaux développements étaient un des livrables de l'ANR SOLSTICE[SOL]. Nous les avions demandés à l'équipe MUMPS (en partenariat avec l'équipe Algo du CERFACS) pour rendre ce produit iso-fonctionnel par rapport aux autres solveurs directs de *Code_Aster*.

Et en pratique, comment MUMPS procède t'il ?
A gros traits, lors de la construction de la matrice factorisée, il détecte les lignes comportant des pivots[25] très petits (par rapport au un critère CNTL(3)[26]). Il les répertorie dans le vecteur PIVNUL_LIST(1:INFOG(28)) et,

23 Les singularités dites numériques sont déterminées à une précision numérique près, contrairement à la singularité dite exacte ou vraie.

24 C'est une solution possible du problème du moment que le second membre $\mathbf{f} \in ker((\mathbf{K}^T)^T)$. Ce qui dans notre cas symétrique revient à $\mathbf{f}$ élément de l'espace image.

25 Il s'agit, en toute rigueur, de la norme infinie de la ligne de la matrice de travail comportant le pivot.

26 Par défaut on le fixe à 10^{-8} (en double précision) et 10^{-4} (en simple) car ces chiffres représentent (empiriquement) une perte d'au moins la moitié du niveau de précision si on poursuit quand même la factorisation.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

suivant le cas de figure, soit il les remplace par une valeur pré-fixée (*via* CNTL(5)[27]), soit il les stocke à part. Le bloc ainsi constitué (de plus petite taille) subira ultérieurement un algorithme QR *ad hoc*.
Et pour finir, les itérations de raffinement itératifs viennent compléter cet écheveau. Comme elles n'utilisent cette factorisée «retouchée» uniquement que comme préconditionneur, et qu'elles bénéficient, par contre, de l'information exacte du produit matrice-vecteur, elles ramènent[28] la solution «biaisée» dans le bon chemin !

Remarques:

• Les paramètres MUMPS ICNTL(13)/ICNTL(24)/ICNTL(25) et CNTL(3)/CNTL(5) permettent d'activer ces fonctionnalités. Ils ne sont pas modifiables par l'utilisateur. Par prudence, on garde la fonctionnalité activée en permanence.
• Cette fonctionnalité peut aussi s'avérer utile en calcul modal (filtrage des modes rigides).

2.3.6 Compression 'Block Low-Rank' (BLR)

Cette technique de compression vise à faciliter les grosses études en réduisant leurs coûts en temps et en mémoire. Elle est **complémentaire du parallélisme et de la panoplie algorithmique/fonctionnelle** du produit**. Son périmètre d'utilisation est quasi-complet.** On n'a pas à choisir entre le parallélisme, tel ou tel raffinement numérique et ces compressions BLR ! Toutes ces fonctionnalités sont compatibles et leurs gains souvent se cumulent[29].
A la manière des formats mp3, zip ou pdf de nos usages domestiques et bureautiques, ces compressions permettent de réduire, avec peu de pertes, les étapes coûteuses de MUMPS[30]. Et cette approximation ne perturbe généralement pas la précision et le comportement des calculs mécaniques englobants.
Elle n'est cependant intéressante que sur des problèmes de grandes tailles (N au moins > **2.10 ^{6} ddls**). Car comme ces compressions impliquent un surcoût, il ne faut comprimer que les blocs de données suffisamment grands et donc susceptibles de compenser rapidement ce surcoût.
Les **gains constatés** sur quelques études *Code_Aster* varient de **20%** à **80%** (cf. figures 7.3-5/7.3-6). Ils augmentent avec la taille du problème et son caractère massif.

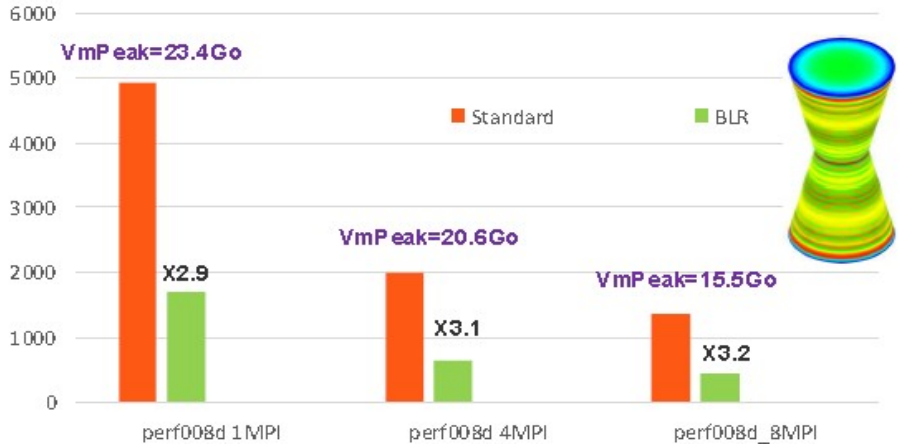


Figure 7.3-5: Exemple de gains procurés par les compressions low-rank sur le cas test de performance perf008d (paramètres par défaut, gestion mémoire en OOC, N=2M, NNZ=80M, Facto_METIS4=7495M, conditionnement=10^7). On trace, en fonction du nombre de processus MPI activés, les temps elapsed consommés par toute l'étape de résolution de système linéaire dans Code_Aster v13.1, son pic mémoire RAM, ainsi que le facteur d'accélération procuré par BLR.

27 Cette valeur doit être assez grande pour limiter l'impact de cette modification sur le reste de la factorisation. Dans *Code_Aster*/Code_Carmel3D, on la fixe à $10^6 \|\mathbf{K}_{travail}\|$.

28 C'est le même mécanisme que pour le pivotage statique.

29 Par contre ces gains varient en fonction du contexte numérico-informatique: renuméroteur, nombre de processus MPI...

30 Pour l'instant la compression ne concerne que certaines étapes de la factorisation numérique (pas les descente-remontées). Ces gains doivent donc, *a minima*, compenser un léger surcoût dans l'étape d'analyse préliminaire ainsi que les coûts de compression/décompression du début et de la fin de l'étape de factorisation numérique. Pour l'instant il ne s'agit que de gains en temps (pas de gains en pic RAM).

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

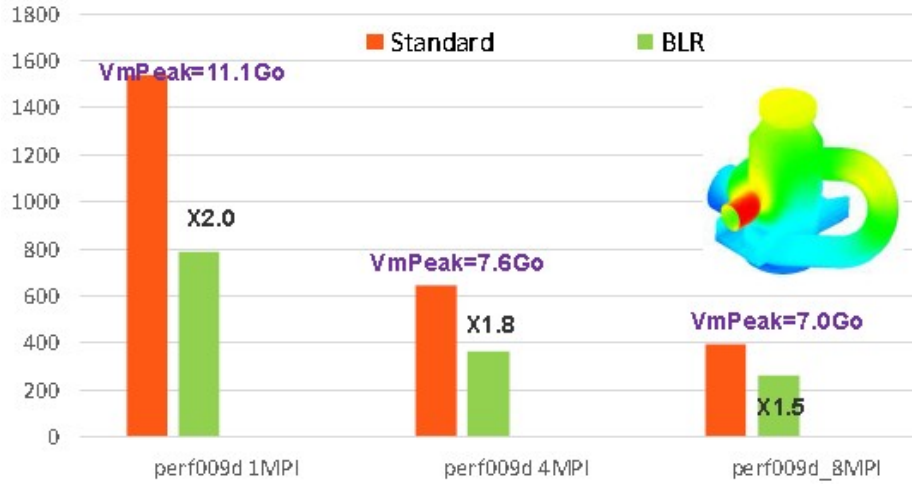


Figure 7.3-6: Exemple de gains procurés par les compressions low-rank sur le cas test de performance perf009d (paramètres par défaut, gestion mémoire en OOC, N=5.4M, NNZ=209M, Facto_METIS4=5247M, conditionnement=10^8). On trace, en fonction du nombre de processus MPI activés, les temps elapsed consommés par toute l'étape de résolution de système linéaire dans Code_Aster v13.1, son pic mémoire RAM, ainsi que le facteur d'accélération procuré par BLR.

A grands traits, **cette stratégie cherche à compresser** les gros blocs de données les plus manipulés par la multifrontale de MUMPS : les **gros blocs denses de son arbre d'élimination**. Cette technique repose sur l'hypothèse (souvent vérifiée empiriquement) qu'il est possible de renuméroter[31] les variables au sein de ces blocs denses afin de dégager une structure matricielle plus avantageuse : **décomposer ces blocs comme produit de deux matrices denses beaucoup plus petites**.

L'objectif est donc de décomposer les grosses matrices denses $\mathbf{A}$ (de cette arbre d'élimination) , par exemple de taille m x n , sous la forme

$$\mathbf{A} = \mathbf{U}.\mathbf{V}^T + \mathbf{E} \tag{7.3-9}$$

avec $\mathbf{U}$ et $\mathbf{V}$ des matrices beaucoup plus petites (respectivement mxk et kxn avec $k < min(m,n)$) et $\mathbf{E}$ une matrice mxn «négligeable» ($\|\mathbf{E}\| < \varepsilon$).

Lors des manipulations ultérieures MUMPS fait alors l'approximation

$$\mathbf{A} \approx \mathbf{U}.\mathbf{V}^T \tag{7.3-10}$$

en pariant que celle-ci aura peu d'impact sur la qualité du résultat (grâce au raffinement itératif ou aux algorithmes non linéaires englobants) et sur les 'outputs' connexes du solveur (détection de singularité, calcul de déterminant, critère de Sturm).

Et c'est la plupart du temps vérifié tant que le paramètre de compression est assez petit ($\varepsilon < 10^{-9}$). Si celui est plus grand, l'approximation ne peut plus être négligée et MUMPS ne peut plus être utilisé en tant que solveur direct «exacte»: seulement en tant que solveur direct «relaxé» (en non linéaire) ou en tant que préconditionneur (par exemple pour le GCPC de Code_Aster out les solveurs de Krylov de PETSc).

Cette fonctionnalité maintenant disponible dans l**es versions consortium** du produit résulte d'un **partenariat EDF-INPT (2010-13)** autour de la **travaux de thèse de C.Weisbecker**[CW13/15]. Celle-ci a été récompensée par un des prix L.Escande de l'année 2014. Pour plus de détails sur les aspects techniques de ces travaux on pourra consulter le document de thèse proprement dit ou le résumé fourni à l'annexe n°1 de ce document.

Remarque:

• Suivant les versions de MUMPS, différents paramètres permettent de gérer cette stratégie de compression. L'utilisateur Code_Aster les active indirectement via les mots-clés `ACCELERATION/LOW_RANK_SEUIL`.

31 Suivant certains critères propres au BLR.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

3 Implantation dans *Code_Aster*

3.1 Contexte/synthèse

Pour améliorer les performances des calculs réalisés, la **stratégie retenue par *Code_Aster*** [Dur08], comme par la plupart des grands codes généralistes en mécanique des structures, consiste notamment à diversifier son panel de solveurs linéaires[32] afin de mieux cibler les besoins et les contraintes des utilisateurs: machine locale, cluster ou centre de calcul; encombrement mémoire et disque; temps CPU; étude industrielle ou plus exploratoire…
Coté parallélisme et solveur linéaire, une voie est particulièrement prospectée[33]:

- le «**parallélisme numérique**» de librairies de solveurs externes telles que MUMPS et PETSc, éventuellement complété par un «parallélisme informatique» (interne au code) pour les calculs élémentaires et les assemblages matriciels/vectoriels;

Nous nous intéressons ici au premier scénario au travers de MUMPS. **Ce solveur externe est «plugé» dans *Code_Aster* et accessible aux utilisateurs depuis la v8.0.14**. Il nous permet ainsi de bénéficier, «à moindre frais», du Rex d'une large communauté d'utilisateurs et des compétences très pointues d'équipes internationales. Le tout en conjuguant efficacité, performance, fiabilité et large périmètre d'utilisation.
Ce travail a d'abord été réalisé en exploitant le mode séquentiel In-Core du produit. En particulier, grâce à ses facultés de pivotage, il rend de précieux services en traitant de nouvelles modélisations (éléments quasi-incompressibles, X-FEM…) qui peuvent s'avérer problématiques pour les autres solveurs linéaires.
Depuis, **MUMPS est quotidiennement utilisé sur des études** [GM08][Tar07][GS11]. Notre Rex s'est bien sûr étoffé et nous entretenons une **relation partenariale active avec l'équipe de développement de MUMPS** (notamment *via* l'ANR SOLSTICE[SOL] et une thèse en cours). D'autre part, son intégration dans *Code_Aster* bénéficie d'un enrichissement continu: parallélisme centralisé IC[Des07] (depuis la v9.1.13), parallélisme distribué IC[Boi07] (depuis la v9.1.16) puis mode IC et OOC[BD08] (depuis la v9.3.14) .
En mode parallèle distribué, l'usage de MUMPS procure des **gains en CPU** (par rapport à la méthode par défaut du code) **de l'ordre de la douzaine sur 32 processeurs** de la machine *Aster*. Sur des cas très favorables ce résultat peut être bien meilleur et, pour des «études frontières», MUMPS reste parfois la seule alternative viable (cf. internes de cuve [Boi07]).
Quant aux consommations RAM, on a vu aux chapitres précédents que c'est la principale faiblesse des solveurs directs. Même en mode parallèle, où l'on a pourtant naturellement une distribution des données entre les processeurs, ce facteur peut s'avérer handicapant. Pour surmonter ce problème il possible d'activer dans *Code_Aster*, une fonctionnalité récente de MUMPS (développée dans le cadre de l'ANR pré-citée): l'«Out-Of-Core» (OOC), pendant du «In-Core» (IC) par défaut. Elle permet de réduire ce goulot d'étranglement en déchargeant sur disque bon nombre de données. Grâce à l'OOC, on peut ainsi **s'approcher des consommations RAM de la multifrontale native** de *Code_Aster* (même en séquentiel), voire descendre en dessous en conjuguant les efforts du parallélisme et de ce déchargement sur disque. Les premiers essais montrent un gain en RAM entre l'OOC et l'IC d'au moins 50% (voire plus sur des cas favorables) pour un surcoût en CPU limité (<10%).
Le solveur MUMPS permet donc, non seulement de résoudre des problèmes numériquement difficiles, mais, inséré dans un processus de calcul Aster déjà partiellement parallèle, il en démultiplie les performances. Il procure au code un cadre parallèle performant, générique, robuste et grand public. Il facilite ainsi le passage des études standards (< million de degrés de liberté) et rend accessible au plus grand nombre le traitement de gros cas (~ plusieurs millions de degrés de liberté).

3.2 Deux types de parallélisme: centralisé et distribué

3.2.1 Principe

MUMPS est un solveur linéaire parallélisé. Ce parallélisme peut être activé de plusieurs manières notamment suivant les aspects séquentiels ou parallèles du code qui l'utilise. Ainsi, ce **parallélisme peut être limité aux flots de données/traitements internes à MUMPS**, ou, *a contrario* **s'intégrer à un flot de**

32 Cette recherche continue d'amélioration des performances ne se réduit évidemment pas qu'aux seuls solveurs linéaires. Le code propose bon nombre d'outils pour répondre aux mêmes objectifs: distribution de calculs indépendants, X-FEM, amélioration du contact-frottement, des solveurs EDO/modaux/non-linéaires, maillage adaptatif, éléments finis de structure…

33 Cf. [R6.01.03] pour une vision détaillée des stratégies de parallélisme potentielles et de celles effectivement mises en œuvre dans le code.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

données/traitements parallèles déjà organisés en amont du solveur, dès les phases de calculs élémentaires de *Code_Aster*. Le premier mode ('CENTRALISE') a pour lui la robustesse et un plus large périmètre d'utilisation, le second ('GROUP_ELEM'/'MAIL_***' et 'SOUS_DOMAINE') est moins générique mais plus efficace.
Car les phases souvent les plus coûteuses en temps CPU d'une simulation sont: la construction du système linéaire (purement *Code_Aster* , découpée en trois postes: factorisation symbolique, calculs élémentaires et assemblages matriciels/vectoriels) et sa résolution (dans MUMPS, cf. §1.6: renumérotation + analyse, factorisation numérique et descente-remontée). Le premier mode de parallélisation ne profite que du parallélisme des étapes 2 et 3 de MUMPS, alors que les trois autres parallélisent aussi les calculs élémentaires et les assemblages de *Code_Aster (*cf. figure 2.2-1 et 3.2-1*)* .

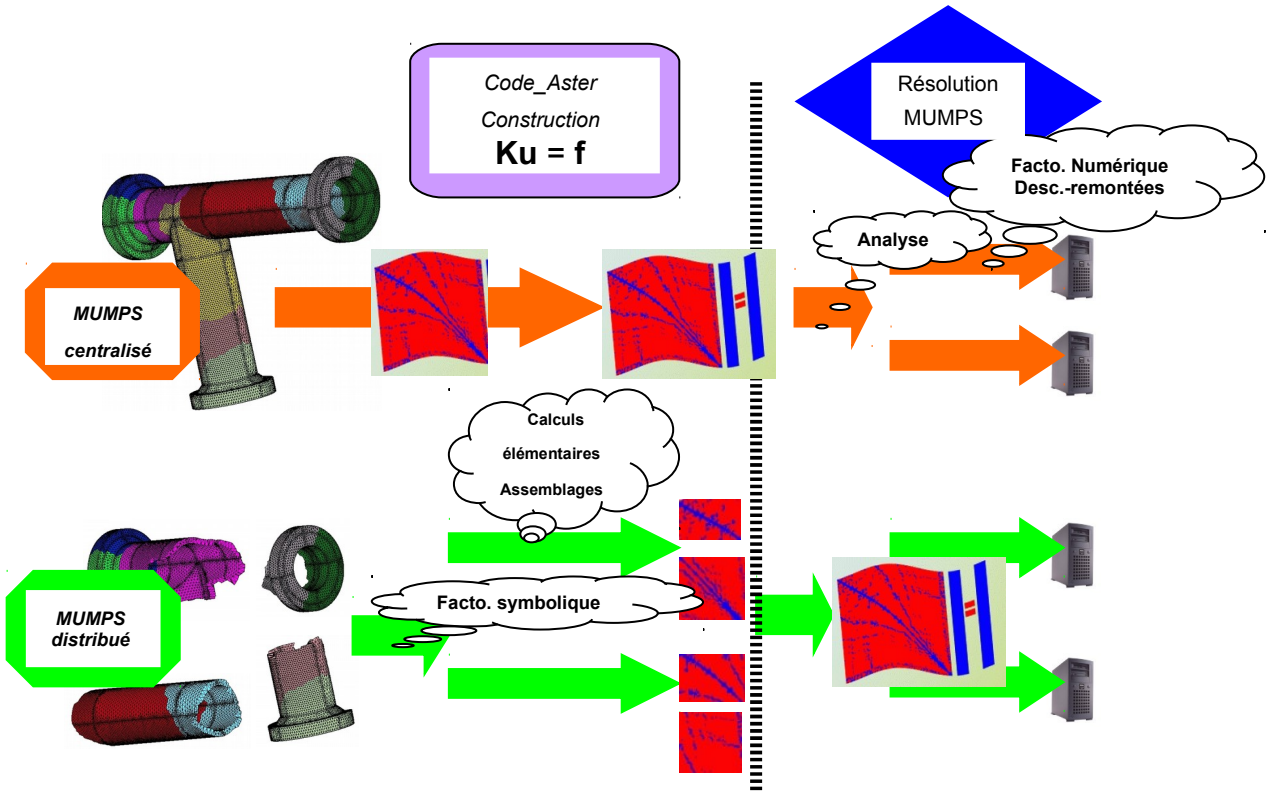


Figure 3.2-1._ Flots de données/traitements parallèles de MUMPS centralisé/distribué.

3.2.2 Différents modes de distribution

On décrit dans ce paragraphe les différents choix de distribution des calculs élémentaires aux processeurs participant au calcul. Cette distribution se paramètre lors de la construction du modèle élément fini.
Un premier choix consiste à ne pas répartir les calculs élémentaires entre processeurs. Si l'on souhaite distribuer les calculs élémentaires, on peut s'appuyer sur une répartition des éléments finis ou bien une répartition des mailles (indépendamment des éléments finis portés par ces mailles) . On peut enfin combiner répartition par élément fini et par maille. Détaillons chaque mode:

- **CENTRALISE : Les mailles ne sont pas distribuées** (comme en séquentiel). Les calculs élémentaires ne sont pas parallélisés. Le parallélisme ne commence qu'au niveau de MUMPS. Chaque processeur construit et fournit au solveur linéaire l'intégralité du système à résoudre. Ce mode d'utilisation est utile pour les tests de non-régression. Dans tous les cas de figure où les calculs élémentaires représentent une faible part du temps total (par ex. en élasticité linéaire), cette option peut être suffisante.
- **'GROUP_ELEM' :** Un second type de répartition consiste à constituer des groupes homogènes d'éléments finis (par type d'élément fini) puis à distribuer les calculs élémentaires entre les processeurs pour équilibrer au mieux la charge (en terme de nombre de calculs élémentaires de chaque type). Chaque processeur alloue la matrice entière mais n'effectue et n'assemble que les calculs élémentaires qui lui ont été attribués.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

- **'MAIL_DISPERSE/MAIL_CONTIGU':**
  On effectue une distribution des mailles du modèle soit par **paquets de mailles contiguës** ('MAIL_CONTIGU'), soit par **distribution cyclique** ('MAIL_DISPERSE'). Cette distribution est indépendante du type d'élément fini porté par les mailles. Par exemple, avec un modèle comportant 8 mailles et pour un calcul sur 4 processeurs, on a les répartitions de charge suivantes:

Mode de distribution	*Maille 1*	*Maille 2*	*Maille 3*	*Maille 4*	*Maille 5*	*Maille 6*	*Maille 7*	*Maille 8*
MAIL_CONTIGU	Proc. 0	Proc. 0	Proc. 1	Proc. 1	Proc. 2	Proc. 2	Proc. 3	Proc. 3
MAIL_DISPERSE	Proc. 0	Proc. 1	Proc. 2	Proc. 3	Proc. 0	Proc. 1	Proc. 2	Proc. 3

  Chaque processeur alloue la matrice entière mais n'effectue et n'assemble que les calculs élémentaires qui lui ont été attribués.

- **'SOUS_DOMAINE' (défaut)** : Ce type de distribution est une distribution hybride des calculs élémentaires qui s'appuie sur un e répartition des mailles (à partir d' un partitionnement du maillage global en sous-domaines) puis sur une répartition des éléments finis par type à l'intérieur de chaque sous-domaine. Chaque processeur alloue la matrice entière mais n'effectue et n'assemble que les calculs élémentaires qui lui ont été attribués.

3.2.3 Équilibrage de charge

La distribution par mailles est très simple mais peut conduire à des déséquilibres de charge car elle ne tient pas explicitement compte des mailles spectatrices, des mailles de peau (cf. figure 3.2-2 sur un exemple comportant 8 mailles volumiques et 4 mailles de peau), de zones particulières (non linéarités…). La distribution par sous-domaines est plus souple et peut s'avérer plus efficace en permettant d'adapter son flot de données à sa simulation.
Une autre cause de déséquilibrage peut provenir des conditions de Dirichlet par dualisation (DDL_IMPO, LIAISON_***…). Par soucis de robustesse, leur traitement est affecté seulement au processeur maître. Cette surcharge de travail, souvent négligeable, introduit cependant dans certains cas, un déséquilibrage plus marqué. L'utilisateur peut le compenser en renseignant un des mot-clés CHARGE_PROC0_MA/SD. Ce traitement différencié concerne en fait tous les cas de figures impliquant des mailles dites «tardives» (Dirichlet *via* des multiplicateurs de Lagrange mais aussi force nodale, contact méthode continue…).
Pour plus de détails sur les spécifications informatiques et les implications fonctionnelles de ce mode de parallélisme on pourra consulter les documentations [U2.08.03] et [U4.50.01].

Remarques:

• Sans l'option MATR_DISTRIBUEE (cf. paragraphe suivant), les différente stratégies sont équivalentes en terme d'occupation mémoire. On tarit le plus tôt possible le flot de données et d'instructions. Il s'agit de traiter sélectivement des blocs matriciels/vectoriels du problème global, que MUMPS va rassembler.

• En mode distribué, chaque processeur ne manipule que des matrices partiellement remplies. Par contre, afin d'éviter d'introduire trop de communications MPI dans le code (critères d'arrêt, résidu…), ce scénario n'a pas été retenu pour les seconds membres. Leur construction est bien parallélisé, mais en fin d'assemblage, les contributions de tous les processeurs sont sommées et envoyées à tous. Ainsi tous les processeurs connaissent entièrement les vecteurs impliqués dans le calcul.

• De même, la matrice est pour l'instant dupliquée : dans l'espace JEVEUX (RAM ou disque) et dans l'espace F90 de MUMPS (RAM). A terme, du fait du déchargement sur disque de la factorisée, elle va devenir un objet dimensionnant de la RAM. Il faudra donc la construire directement via MUMPS.

3.2.4 Retailler les objets *Code_Aster*

En mode parallèle, lorsqu'on distribue les données JEVEUX en amont de MUMPS, on ne redécoupe pas forcément les structures de données concernées. Avec l'option MATR_DISTRIBUEE='NON', tous les objets distribués sont alloués et initialisés à la même taille (la même valeur qu'en séquentiel). Par contre, chaque processeur ne va modifier que les parties d'objets JEVEUX dont il a la charge. Ce scénario est particulièrement

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

adapté au mode parallèle distribué de MUMPS (mode par défaut) car ce produit regroupe en interne ces flots de données incomplets. Le parallélisme permet alors, outre des gains en temps calcul, de réduire la place mémoire requise par la résolution MUMPS mais pas celle nécessaire à la construction du problème dans JEVEUX.
Ceci n'est pas gênant tant que l'espace RAM pour JEVEUX reste très inférieur à celui requis par MUMPS. Comme JEVEUX stocke principalement la matrice et MUMPS, sa factorisée (généralement des dizaines de fois plus grosse), le goulet d'étranglement RAM du calcul est théoriquement sur MUMPS. Mais dès qu'on utilise quelques dizaines de processeurs en MPI et/ou qu'on active l'OOC, comme MUMPS distribue cette factorisée par processeur et décharge ces morceaux sur disque, la «balle revient dans le camp de JEVEUX».
D'où l'option MATR_DISTRIBUEE qui retaille la matrice, au plus juste des termes non nuls dont a la responsabilité le processeur. L'espace JEVEUX requis diminue alors avec le nombre de processeurs et descend en dessous de la RAM nécessaire à MUMPS. Les résultats de la figure 3.2-2 illustrent ce gain en parallèle sur deux études: une Pompe RIS et une modèle de cuve de l'étude « Epicure ».

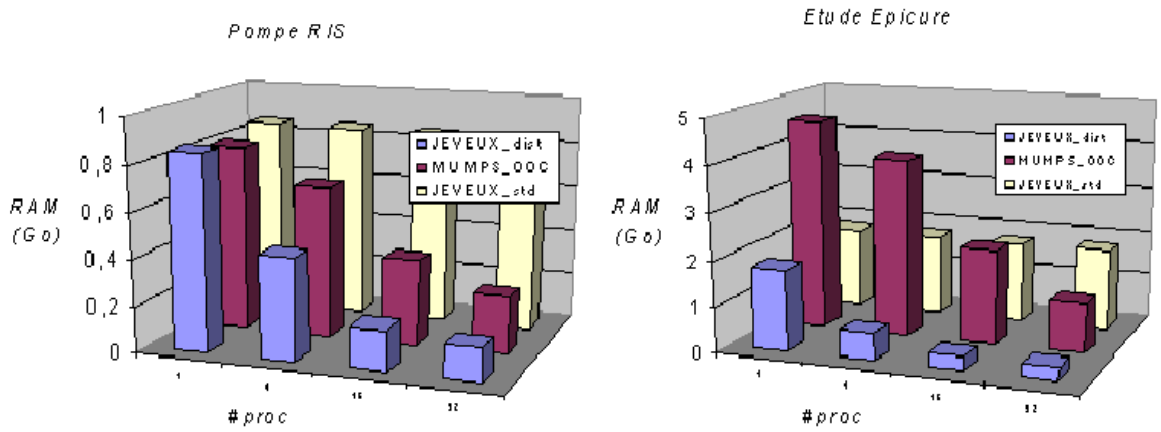


Figure 3.2-2._ Evolution des consommations RAM (en Go) en fonction du nombre de processeurs, de Code_Aster v11.0 (JEVEUX standard MATR_DISTRIBUE='NON' et distribué, resp. 'OUI') et de MUMPS OOC. Résultats effectués sur une Pompe RIS et sur la cuve de l'étude Epicure.

Remarques:

- *On traite ici les données résultant d'un calcul élémentaire (RESU_ELEM et CHAM_ELEM) ou d'un assemblage matriciel (MATR_ASSE). Les vecteurs assemblés (CHAM_NO) ne sont pas distribués car les gains mémoire induits seraient faibles et, d'autre part, comme ils interviennent dans l'évaluation de nombreux critères algorithmiques, cela impliquerait trop de communications supplémentaires.*
- *En mode MATR_DISTRIBUE, pour faire la jointure entre le bout de MATR_ASSE local au processeur et la MATR_ASSE globale (que l'on ne construit pas), on rajoute un vecteur d'indirection sous la forme d'un NUME_DDL local.*

3.3 Gestion de la mémoire MUMPS et *Code_Aster*

Pour activer ou désactiver les facultés OOC de MUMPS (cf. figure 3.3-1)**,** l'utilisateur renseigne le mot-clé SOLVEUR/GESTION_MEMOIRE='IN_CORE'/'OUT_OF_CORE'/'AUTO' (défaut). Cette fonctionnalité est bien sûr cumulable avec le parallélisme d'où une plus grande variété de fonctionnement pour éventuellement s'adapter aux contingences d'exécution: «séquentiel IC ou OOC», «parallélisme centralisé IC ou OCC», «parallélisme distribué par sous-domaines IC ou OOC»...
Pour un petit cas linéaire, le mode «séquentiel IC» suffit; pour un plus gros cas toujours en linéaire, le mode «parallèle centralisé IC» (ou mieux OOC) amène véritablement un gain en CPU et en RAM; en non linéaire, avec réactualisation fréquente de la matrice tangente, le mode «parallèle distribué OOC» est conseillé.
Pour plus de détails sur les spécifications informatiques et les implications fonctionnelles de ce mode de gestion de la mémoire MUMPS on pourra consulter les documentations [BD08] et [U2.08.03/U4.50.01].

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

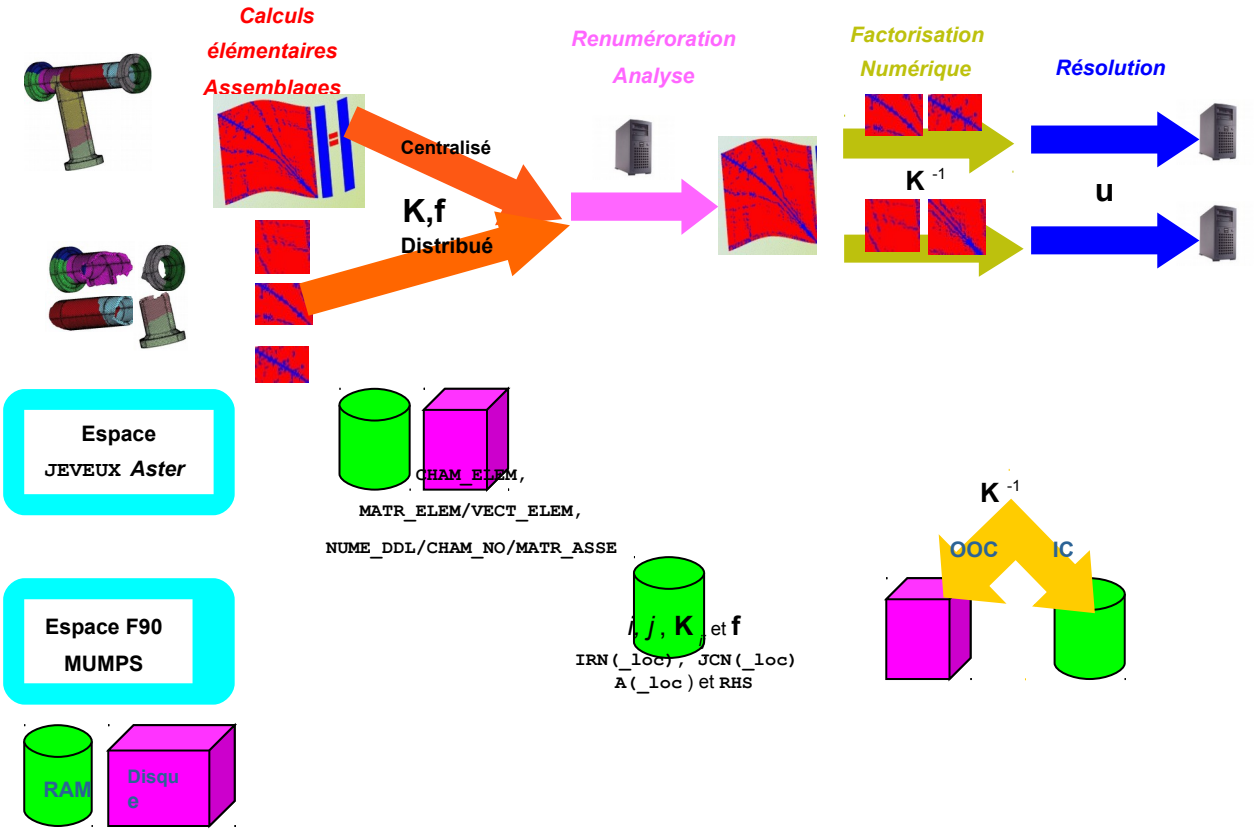


Figure 3.3-1._ Schéma fonctionnel du couplage Code_Aster/MUMPS (avec une rénuméroteur séquentiel) vis-à-vis des principales structures de données et de l'occupation mémoire (RAM et disque).

3.4 Gestion particulière des double multiplicateurs de Lagrange

Historiquement, les solveurs linéaires directs de *Code_Aster* ('MULT_FRONT' et 'LDLT') ne disposaient pas d'algorithme de pivotage (qui cherche à éviter les accumulations d'erreurs d'arrondis par division par des termes très petits). Pour contourner ce problème, la prise en compte des conditions limites par des multiplicateurs de Lagrange (AFFE_CHAR_MECA/THER...) a été modifiées en introduisant des doubles-multiplicateurs de Lagrange. Formellement, on ne travaille pas avec la matrice initiale $\mathbf{K}_0$

$$\mathbf{K}_0=\begin{bmatrix}\mathbf{K} & \text{blocage}\\ \text{blocage} & \mathbf{0}\end{bmatrix}\begin{matrix}\mathbf{u}\\ \text{lagr}\end{matrix}$$

mais avec sa forme doublement dualisée $\mathbf{K}_2$

$$\mathbf{K}_2=\begin{bmatrix}\mathbf{K} & \text{blocage} & \text{blocage}\\ \text{blocage} & -\mathbf{1} & \mathbf{1}\\ \text{blocage} & \mathbf{1} & -\mathbf{1}\end{bmatrix}\begin{matrix}\mathbf{u}\\ \text{lagr}_1\\ \text{lagr}_2\end{matrix}$$

D'où un surcoût mémoire et calcul.
Comme MUMPS dispose de facultés de pivotage, ce choix de dualisation des conditions limites peut être remis en cause. En initialisant le mot-clé ELIM_LAGR à 'LAGR2', **on ne tient plus compte que d'un Lagrange, l'autre étant spectateur**[34]. D'où une matrice de travail K_1 simplement dualisée

34 Pour maintenir la cohérence des structures de données et garder une certaine lisibilité/maintenabilité informatique, il est préférable de «bluffer» le processus habituel en passant de K_2 à K_1, plutôt qu'au scénario optimal K_0.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

$$\mathbf{K}_1 = \begin{bmatrix} \mathbf{K} & \text{blocage} & \mathbf{0} \\ \text{blocage} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & -\mathbf{1} \end{bmatrix} \begin{matrix} \mathbf{u} \\ \text{lagr}_1 \\ \text{lagr}_2 \end{matrix}$$

plus petite car les termes extra-diagonaux des lignes et des colonnes associées à ces multiplicateurs de Lagrange spectateurs sont alors initialisées à zéro. *A contrario*, avec la valeur 'NON', MUMPS reçoit les matrices dualisées usuelles.

Pour les **problèmes comportant de nombreux multiplicateurs de Lagrange (jusqu'à 20% du nombres d'inconnues totales)**, l'activation de ce paramètre est souvent payante (matrice plus petite). Mais lorsque **ce nombre explose (>20%)**, ce procédé peut-être contre-productif. Les gains réalisés sur la matrice sont annulés par la taille de la factorisée et surtout le nombre de pivotages tardifs que MUMPS doit effectuer. Imposer ELIM_LAGR='NON' peut être alors très intéressant (gain de 40% en CPU sur le cas-test mac3c01).

3.5 Périmètre d'utilisation

A priori , tous les opérateurs/fonctionnalités utilisant la résolution d'un système linéaire sauf une configuration de calcul modal[35]. Pour plus de détails on pourra consulter les documentations Utilisateur [U4.50.01].

3.6 Paramétrage et exemples d'utilisation

Récapitulons le paramétrage principal permettant de piloter MUMPS dans *Code_Aster* et illustrons son utilisation *via* un cas-test officiel (mumps05b) et une géométrie d'étude (pompe RIS). Pour plus d'informations on pourra consulter les documentations Utilisateur associées [U2.08.03/U4.50.01], les notes EDF [BD08][Boi07] [Des07] ou les cas-tests utilisant MUMPS.

3.6.1 Paramètres d'utilisation de MUMPS *via Code_Aster*

Opérande	Mot-clé	Valeur par défaut	Détails/conseils	Réf.
SOLVEUR/ METHODE= 'MUMPS'				
Paramètres fonctionnels	TYPE_RESOL	'AUTO' ('NONSYM' si la matrice est non symétrique, 'SYMGEN' sinon)	'AUTO', 'NONSYM', 'SYMGEN' et 'SYMDEF'. Paramètre permettant de préciser la nature du problème à traiter.	§1
	PCENT_PIVOT	10%	Surcoût mémoire prévu pour les pivotages.	§2.3
	ELIM_LAGR	'LAGR2'	'LAGR2'/'NON'.	§3.4
	RESI_RELA	-1 (non linéaire) 10^{-6} (linéaire)	Paramètre en lien avec le mot-clé POSTTRAITEMENTS. Si ce paramètre est positif, MUMPS effectue des itérations de raffinement itératif et examine la qualité de la solution. Si l'erreur relative en solution est inférieures à cette valeur, Aster s'arrête en	§2.3

35 Résolution d'un problème quadratique avec CALC_MODES + OPTION='SEPARE'/'AJUSTE'. Car cette option requiert un accès direct à la diagonale de la matrice factorisée. Or MUMPS ne permet pas, pour l'instant, de connaître les termes précis de la factorisée réellement obtenue. Par contre, et c'est son cadre d'utilisation privilégié, il les utilise efficacement et avec robustesse pour résoudre un système linéaire et/ou pour calculer un post-traitement (critère de Sturm, calcul de déterminant…).

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

Opérande	Mot-clé	Valeur par défaut	Détails/conseils	Réf.
			ERREUR_FATALE. **CAS PARTICULIER :** POSTTRAITEMENTS='MINI'.	
Paramètres numériques	PRETRAITEMENTS	'AUTO'	'AUTO' et 'SANS'.	§1.6 § 2.3
	RENUM	'AUTO'	'AUTO', 'AMD', 'AMF', 'QAMD', 'PORD', '(PT)SCOTCH' et '(PAR)METIS'. Dans le premier cas de figure MUMPS choisit le meilleur renumeroteur disponible, dans les autres, on lui impose. Si ce renumeroteur n'est pas disponible: ERREUR_FATALE ou ALARME et on le remplace par un autre du même type.	§ 1.6
	FILTRAGE_MATRICE /MIXER_PRECISION		Options pour «relaxer» les résolutions effectuées *via* MUMPS.	[U4.50.01]
	POSTTRAITEMENTS	'AUTO'	'AUTO', 'MINI', FORCE' et 'SANS'.	§2.3
	ACCELERATION/LOW_RANK_SEUIL	'AUTO'/0.0	Utile principalement sur de gros problèmes (N> 2.10^6 ddls). Utiliser en complément POSTTRAITEMENTS='MINI'.	§2.3 et annexe n°1
Mémoire	GESTION_MEMOIRE	'AUTO'	'IN_CORE', 'OUT_OF_CORE', 'AUTO' ou 'EVAL'.	§3.3
	MATR_DISTRIBUEE	'NON'	'OUI' ou 'NON'.	§3.2

Tableau 3.6-1._ Récapitulatif du paramétrage spécifique de MUMPS dans Code_Aster.

3.6.2 Monitoring

En positionnant le **mot-clé INFO à 2 et en utilisant le solveur MUMPS** , l'utilisateur peut faire afficher dans le fichier de message un monitoring synthétique des différentes phases de la construction et de la résolution du système linéaire: répartition par processeur du nombre de mailles, des termes de la matrice et de sa factorisée, l'analyse d'erreur (si demandée) et un bilan de leur éventuel déséquilibrage. A **ce monitoring orienté CPU, on rajoute quelques informations sur les consommations RAM de MUMPS**: par processeur, estimation (d'après la phase d'analyse) des besoins en RAM en IC, en OOC et la valeur effectivement utilisée avec rappel de la stratégie choisie par l'utilisateur.Les temps consommés pour chacune des étapes du calcul suivant les processeurs peuvent apparaître aussi. Ils sont gérés par un mécanisme plus global qui n'est pas spécifique à MUMPS (cf. §4.1.2 [U1.03.03] ou la documentation Utilisateur de l'opérateur DEBUT/POURSUITE).

```
****************************************************************
  <MONITORING MUMPS >
  TAILLE DU SYSTEME    803352
 CONDITIONNEMENT/ERREUR ALGO  2.2331D+07 3.3642D-15
 ERREUR SUR LA SOLUTION  7.5127D-08
  RANG     NBRE MAILLES     NBRE TERMES K     LU FACTEURS
  N    0 :           54684          7117247        117787366
  N    1 :           55483          7152211         90855351
…
   EN % : VALEUR RELATIVE ET DESEQUILIBRAGE MAX
          :  1.45D+01   2.47D+00   2.38D+00          1.50D+01   4.00D+01   2.57D+01
          :  1.40D-01  -1.09D+00  -5.11D+00         -9.00D-02   1.56D+00  -4.16D-01

   MEMOIRE RAM ESTIMEE ET REQUISE              PAR MUMPS EN Mo(FAC_NUM + RESOL)
  RANG ASTER : ESTIM IN-CORE | ESTIM OUT-OF-CORE | RESOL. OUT-OF-CORE
  N    0 :            1854             512              512
  N    1 :            1493             482              482
…
#1 Resolution des systemes lineaires          CPU (USER+SYST/SYST/ELAPS):    105.68      3.67     59.31
#1.1 Numerotation, connectivite de la matrice CPU (USER+SYST/SYST/ELAPS):      3.26      0.04      3.26
#1.2 Factorisation symbolique                 CPU (USER+SYST/SYST/ELAPS):      3.13      1.20      4.11
#1.3 Factorisation numerique (ou precond.)    CPU (USER+SYST/SYST/ELAPS):     45.22      0.83     23.48
#1.4 Resolution                               CPU (USER+SYST/SYST/ELAPS):     54.07      1.60     28.46
#2 Calculs elementaires et assemblages        CPU (USER+SYST/SYST/ELAPS):      3.44      0.03      3.42
```

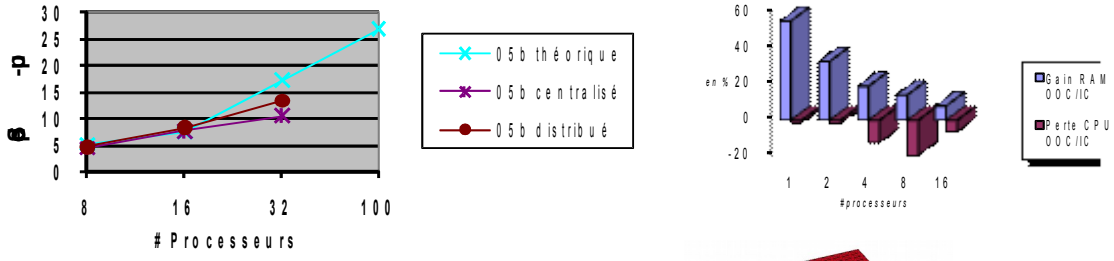

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

```
#2.1 Routine calcul                          CPU (USER+SYST/SYST/ELAPS):      2.20      0.01      2.20
#2.1.1 Routines te00ij                       CPU (USER+SYST/SYST/ELAPS):      2.07      0.00      2.06
#2.2 Assemblages                             CPU (USER+SYST/SYST/ELAPS):      1.24      0.02      1.22
#2.2.1 Assemblage matrices                   CPU (USER+SYST/SYST/ELAPS):      1.22      0.02      1.21
#2.2.2 Assemblage seconds membres            CPU (USER+SYST/SYST/ELAPS):      0.02      0.00      0.01
```

Figure 3.5-1._Extrait de fichier de message en INFO =2.

3.6.3 Exemples d'utilisation

Concluons ce chapitre par deux séries de tests illustrant les **écarts de performance suivant le cas de figure et le critère observé** (cf. figures 3.5-2/3). Le cas-test canonique du cube en linéaire se parallélise très bien. En centralisé (resp. en distribué), plus de 96% (resp. 98%) des phases de construction et d'inversion du système linéaire sont parallélisés. Soit un speed-up théorique proche de 25 (resp. 50). En pratique, sur les nœuds parallèles de la machine centralisée *Aster*, on obtient d'assez bonnes accélérations: speed-up effectif de 14 sur 32 processeurs au lieu des 17 théoriques.



$N = 0.7\,M / nnz = 27\,M$

%parallèle centralisé/distribué: 96/98

Speed-ups théoriques cent./dist. <25/50
32 proc (x1): ~3min
Conso. RAM IC: 4Go (1 proc)/1.3Go (16)
Conso. RAM OOC: 2Go (1 proc)/1.2Go (16)

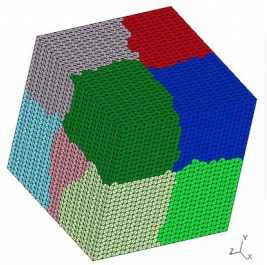


Figure 3.5-2._ Calcul mécanique linéaire (op. MECA_STATIQUE) sur le cas-test officiel du cube (mumps05b). On construit et résout un seul système linéaire. Simulation effectuée sur la machine centralisée Aster avec Code_Aster v11.0. Consommations RAM Aster+MUMPS mesurées.

Sur l'étude non linéaire de la pompe, les gains que l'on peut espérer sont plus faibles. Compte-tenu de la phase d'analyse séquentielle de MUMPS, seulement 82% des calculs sont parallèles. D'où des speed-ups théoriques et effectifs appréciables mais plus modestes. D'un point de vue mémoire RAM, la gestion OOC de MUMPS procure des gains intéressants dans les deux cas, mais plus marqués pour la pompe: en séquentiel, gain IC vs OOC de l'ordre de 85%, contre 50% pour le cube. En augmentant le nombre de processeurs, la distribution de données qu'induit le parallélisme rogne progressivement ce gain. Mais il reste prégnant sur la pompe jusqu'à 16 processeurs et disparaît quasiment avec le cube.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

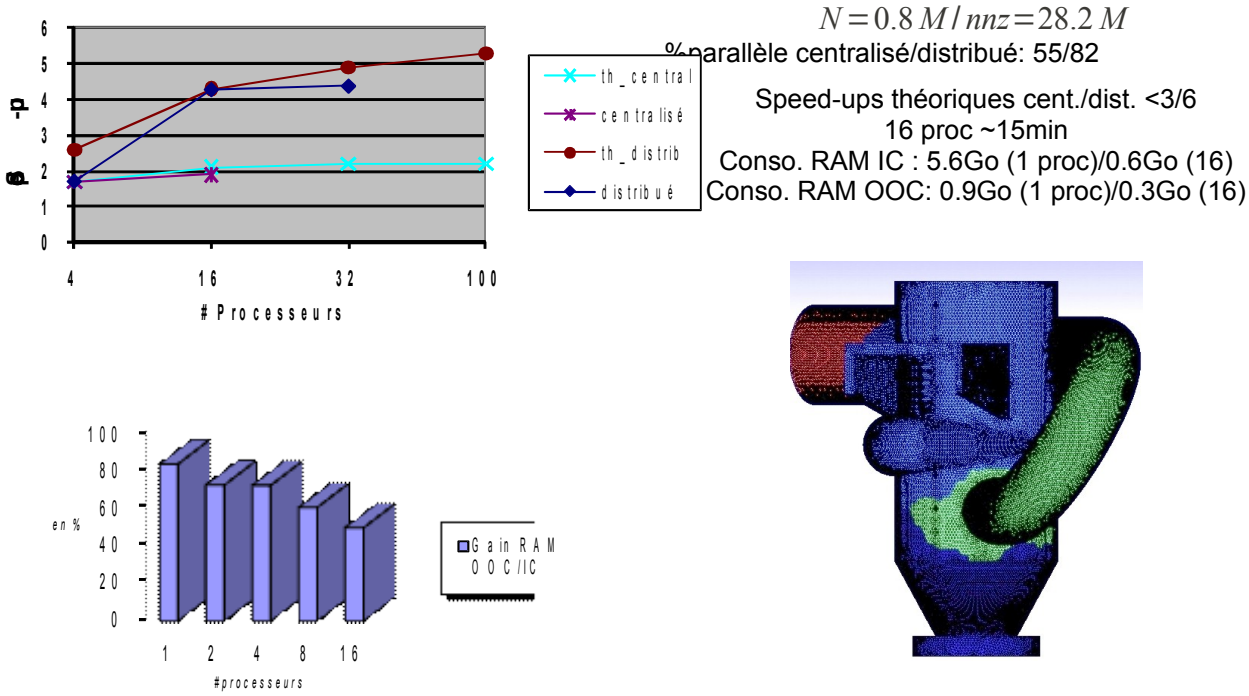


Figure 3.5-3._ Calcul mécanique non linéaire (op. STAT_NON_LINE) sur une géométrie plus industrielle (pompe RIS). On construit et résout 12 systèmes linéaires (3 pas de temps x 4 pas de Newton). Simulation effectuée sur la machine centralisée Aster avec Code_Aster v11.0. Consommations RAM MUMPS estimées.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

4 Conclusion

Dans le cadre des simulations thermo-mécaniques avec *Code_Aster*, **l'essentiel des coûts calcul provient souvent de la construction et de la résolution des systèmes linéaires**. Depuis 60 ans, deux types de techniques se disputent la suprématie dans le domaine, les solveurs directs et ceux itératifs. *Code_Aster*, comme bon nombre de codes généralistes, a fait le choix d'une offre diversifiée dans le domaine. Avec toutefois **une orientation plutôt solveurs directs creux**. Ceux-ci sont plus adaptés à ses besoins que l'on peut résumer sous le triptyque «robustesse/problèmes de type multiples seconds membres/parallélisme modéré». Le code s'appuyant désormais sur de nombreux «middlewares» optimisés et pérennes (MPI, BLAS, LAPACK, METIS…) et s'utilisant principalement sur des clusters de SMP (réseaux rapides, grande capacité de stockage RAM et disque), on cherche à optimiser le traitement des solveurs linéaires dans cette optique.

Compte-tenu de la technicité requise[36] et d'une offre internationale riche[37], **pour effectuer efficacement ces résolutions, la question du recourt à un produit externe est désormais incontournable**. Cela permet d'acquérir, à moindre frais, une fonctionnalité souvent efficace, fiable, performante et bénéficiant d'un large périmètre d'utilisation. On peut ainsi bénéficier du retour d'expérience d'une large communauté d'utilisateurs et des compétences (très) pointues d'équipes internationales.
Ainsi ***Code_Aster* a fait le choix d'intégrer la multifrontale parallèle du package MUMPS**. Ceci en complément, notamment, de sa multifrontale «maison». Mais si celle-ci bénéficie d'une adaptation de longue haleine aux modélisations *Aster,* elle reste moins riche en fonctionnalités (pivotage, pré/post-traitements, qualité de la solution…) et moins performantes en parallèle (pour des consommations RAM du même ordre). Pour exploiter certaines modélisations (éléments quasi-incompressibles, X-FEM…) ou passer des «études frontières» (cf. internes de cuves), ce couplage «*Code_Aster*+MUMPS» devient parfois la seule alternative viable.

Depuis, son intégration dans *Code_Aster* bénéficie d'un enrichissement continu et **MUMPS** (`SOLVEUR/METHODE='MUMPS'`) **est quotidiennement utilisé sur des études** . Notre Rex s'est bien sûr étoffé et nous entretenons une **relation partenariale active avec la «core-team» de MUMPS** (notamment *via* l'ANR SOLSTICE et une thèse).
En mode **parallèle**, l'usage de MUMPS procure des **gains en CPU** (par rapport à la méthode par défaut du code, la multifrontale «maison») **de l'ordre de la douzaine sur 32 processeurs** de la machine *Aster*. Sur des cas plus favorables ou en exploitant un deuxième niveau de parallélisme ou les compressions BLR, ce gain CPU peut-être bien meilleur.

Le solveur MUMPS permet donc, non seulement de résoudre des problèmes numériquement difficiles, mais, inséré dans un processus de calcul *Aster* déjà partiellement parallèle, il en démultiplie les performances. **Il procure au code un cadre parallèle performant, générique, robuste et grand public**. Il facilite ainsi le passage des études standards (<million de degrés de liberté) et rend accessible au plus grand nombre le traitement de gros cas (~ plusieurs millions de degrés de liberté).

36 Pour donner un ordre de grandeur, le package MUMPS fait plus de 10^5 lignes (F90/C).
37 Rien que dans le domaine public, on recense des dizaines de packages, librairies, «macro-librairies»…

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

5 Bibliographie

5.1 Livres/articles/proceedings/thèses…

[ADD89] M.Arioli, J.Demmel et I.S .Duff. Solving sparse linear systems with sparse backward error. *SIAM journal on matrix analysis and applications* . 10, 165:190 (**1989**).
[ADE00] P.R.Amestoy, I.S.Duff et J.Y.L'Excellent. *Multifrontal parallel distributed symmetric and unsymmetric solvers*. Comput. Methods in Appl. Mech. Eng. 184, 501:520 (**2000**).
[ADKE01] P.R.Amestoy, I.S.Duff, J.Koster et J.Y.L'Excellent. *A fully asynchronous multifrontal solveur using distributed dynamic scheduling*. SIAM journal of matrix analysis and applications, 23, 15:41 (**2001**).
[AGES06] P.R.Amestoy, A.Guermouche, J.Y.L'Excellent et S.Pralet. *Hybrid scheduling for the parallel solution of linear systems*. Parallel computing. 32, 136:156 (**2006**).
[Che05] K.Chen. *Matrix preconditioning techniques and applications*. Ed. Cambridge University Press (**2005**).
[CW13] C.Weisbecker. *Improving Multifrontal Solvers by Means of Algebraic Block Low-Rank Representations*. PhD thesis of Toulouse University (**2013**). 2013 Leopold Escande thesis award.
[CW15] P.Amestoy, C.Ashcraft, O.Boiteau, A.Buttari, J.Y.L'Excellent and C.Weisbecker. *Improving Multifrontal Methods by Means of Block Low-Rank Representations*. SIAM J.Sci. Comput., 37 (3), A1451-1474 (**2015**).
[Dav06] T.A.Davis. *Direct methods for sparse linear systems*. Ed. SIAM (**2006**).
[Duf06] I.S.Duff et al. *Direct methods for sparse matrices* Ed. Clarendon Press (**2006**).
[Gol96] G.Golub & C.Van Loan. *Matrix computations*. Ed. Johns Hopkins University Press (**1996**).
[Hig02] N.J.Higham. *Accuracy and stability of numerical algorithms*. Ed. SIAM (**2002**).
[Las98] P.Lascaux & R.Théodor. *Analyse numérique matricielle appliquée à l'art de l'ingénieur*. Ed. Masson (**1998**).
[Liu89] J.W.H.Liu. *Computer solution of large sparse positive definite systems*. Prentice Hall (**1981**).
[Meu99] G.Meurant. *Computer solution of large linear systems*. Ed. Elsevier (**1999**).
[Saa03] Y.Saad. *Iterative methods for sparse matrices*. Ed. PWS (**2003**).

5.2 Rapports/compte-rendus EDF

[Anf03] N.Anfaoui. *Une étude des performances de Code_Aster: proposition d'optimisation*. Stage de DESS de mathématiques appliquées de PARIS VI (**2003**).
[BD08] O.Boiteau et C.Denis. *Activation des fonctionnalités «Out-Of-Core» de MUMPS dans Code_Aster*. Compte-rendu EDF R&D CR-I23/08/047 (**2008**).
[Boi07] O.Boiteau. *Intégration de MUMPS parallèle distribué dans Code_Aster*. Note EDF R&D HI-I23/07/03167 (**2007**).
[Boi13] O.Boiteau. *Solveur linéaire MUMPS : chantiers logiciels dans Code_Aster, thèse de C.Weisbecker sur les compressions low-rank et partenariat EDF/INPT*. Note EDF R&D H-I23-2013-03942 (**2013**).
[Boi15] O.Boiteau. *Partenariats et animations scientifiques dans le domaine des bibliothèques HPC d'algèbre linéaire : consortium MUMPS, mini-symposium SOLVER et semestre CIMI*. Note EDF R&D H-I23-2015-04879 (**2015**).
[Boi16] O.Boiteau. *MUMPS : dernières avancées du produit et actualités du consortium*. Note EDF R&D 6125-1106-2016-14581 (2016).
[Des07] T.DeSoza. *Evaluation et développement du parallélisme dans Code_Aster*. Stage de Master ENPC (**2007**) et note EDF R&D HT-62/08/01464 (**2008**).
[Dur08] C.Durand et al. *HPC avec Code_Aster: états des lieux et perspectives*. Note EDF R&D HT-62/08/0139 (**2008**).
[GM08] S.Géniaut et F.Meissonnier. *Faisabilité d'une étude de nocivité de fissure dans une vanne MP par la méthode X-FEM et avec la plate-forme SALOME*. Compte-rendu EDF R&D CR-AMA/08/0255 (**2008**).
[GS11] V.Godard et N.Sellenet. *Calcul HPC avec Code_Aster: état des lieux et perspectives*. CR-AMA-11.042 (**2011**).
[Tar07] N.Tardieu. *GCP+MUMPS, une solution simple pour la résolution de problèmes avec contact en parallèle*. Compte-rendu EDF R&D CR-AMA/07/0257 (**2007**).
[SOL] O.Boiteau. Suivi de l'ANR SOLSTICE. Comptes-rendus EDF R&D (**2007-2010**).

5.3 Ressources internet

[Dav] T.A.Davis. Pointeur sur les packages de solveurs creux directs: http://www. cise.ufl.edu/research/sparse/codes/.
[Don] J.Dongarra. Pointeur sur les packages de solveurs: http://www.netlib.org/utk/people/JackDongarra/la-sw.html .
[MaMa] Site web de MatrixMarket: http://math.nist.gov/MatrixMarket/index.html.
[Mum] Site web officiel de MUMPS: http://graal.ens-lyon.fr/MUMPS.
[MUC] Site web officiel du consortium MUMPS : http://mumps-consortium.org.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

6 Historique des versions du document

Version Aster	Auteur(s) ou contributeur(s), organisme	Description des modifications
9.4	O.BOITEAU EDF R&D SINETICS	Texte initial
V10.4	O.BOITEAU EDF R&D SINETICS	Corrections formelles dues au portage .doc/.odt; Mise à jour sur le parallélisme; **Prise en compte des rqs de l'équipe MUMPS**; Rajout de nouveaux mot-clés (ELIM_LAGR2, LIBERE_MEMOIRE, MATR_DISTRIBUEE); **Amaigrissement de la partie conseil/périmètre d'utilisation maintenant dévolue à la note U2.08.03**.
V11.3	O.BOITEAU EDF R&D SINETICS	Rajout du nouveau mot-clé GESTION_MEMOIRE à la place de OUT_OF_CORE et LIBERE_MEMOIRE. Rajout du paragraphe sur la prise en compte de systèmes singuliers.
V13.1	O.BOITEAU EDF R&D SINETICS	Mise à jour et suppression de quelques paragraphes obsolètes. Rajout des éléments sur le consortium et sur BLR.
V13.3	O.BOITEAU EDF R&D SINETICS	Nouveaux éléments liés à l'usage des threads et aux renuméroteurs parallèles PARMETIS et PTSCOTCH.
V13,4	O.BOITEAU EDF R&D/PERICLES	Nouveau mot-clé ACCELERATION.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

7 Annexe n°1: Principe des compressions BLR dans MUMPS

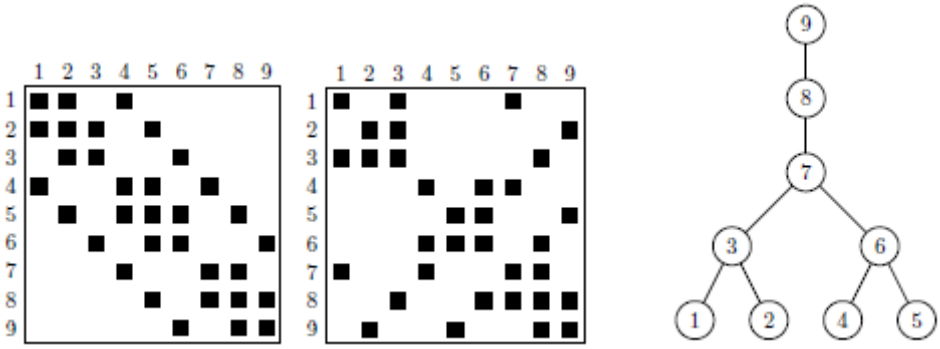
Ce paragraphe résume les différents aspects techniques des travaux de **thèse de Clément Weisbecker** sur les compressions low-rank[CW15][Boi13]. Ces travaux ont été poursuivis et améliorés par l'équipe MUMPS et ils sont en partie disponibles dans la version consortium[MUC] du produit.
Pour plus de détails on pourra consulter le document de thèse proprement dit [CW13]. Les figures réutilisées ici proviennent d'ailleurs, soit de ce document, soit du jeu de slides de sa soutenance.

7.1 Principe de la méthode multifrontale

La **méthode multifrontale est une classe de méthode directe** utilisée pour **résoudre des systèmes linéaires**. C'est cette méthode qui est implantée dans le produit MUMPS et c'est elle qui est en général utilisée dans les grands codes commerciaux en mécanique des structures (ANSYS, NASTRAN…). Même si il existe d'autres stratégies (supernodale…) et qu'elles commencent à être implémentées dans des packages publics (SuperLU, PastiX…), ce travail de thèse s'est exclusivement focalisé sur cette stratégie standard qu'est la multifrontale. Néanmoins beaucoup de ses résultats peuvent sans doute s'étendre ou s'adapter à ces autres méthodes de résolution.

La **méthode multifrontale**, mise au point par I.S.Duff et J.K.Reid (1983), consiste notamment à utiliser la théorie des graphes[38] pour construire, à partir d'une matrice creuse, un **arbre d'élimination organisant efficacement les calculs** (cf. figures 7.1.1). Les carrés noirs matérialisent les termes matriciels non nuls. Pour manipuler, «par graphes interposés», ce type de données on impose usuellement la règle suivante: un point du graphe représente une inconnue et, une arête entre deux points, un terme matriciel non nul.
Ainsi dans l'exemple ci-dessous, les variables 1 et 2 doivent être reliées par une arête (dans la numérotation initiale). Généralement, la **numérotation initiale de la matrice n'est pas optimale**. Afin de réduire l'encombrement mémoire de la factorisée[39], le nombre de flop des manipulations ultérieures et afin d'essayer de garantir un bon niveau de précision du résultat, **on renumérote cette matrice initiale**. On voit ainsi que le nombre de termes supplémentaires ('fill-in') créés par la factorisation passe de 16 à 10.

C'est à partir de cette matrice renumérotée que l'on construit l'arbre d'élimination propre à la multifrontale. **Cette vision arborescente a la grand mérite d'organiser concrètement les tâches:** on détermine précisément leur **dépendance** ou non (pour ménager du parallélisme), leur **précédence** (pour opérer des regroupements afin d'optimiser les performances BLAS); on peut **prévoir certaines consommations** (en temps et en mémoire) voire essayer de limiter les problèmes numériques.
Ainsi dans l'exemple ci-dessous, le traitement de la variable 1 n'a aucune incidence sur les variables 2, 4 et 5. Par contre, il va impacter les variables 3 et 7 qui occupent les niveaux supérieurs de l'arbre (ils sont ses ascendants).



38 Notamment les travaux fondateurs de R.S.Schreiber.
39 Pour limiter ce qu'on appelle le phénomène de «remplissage» ('fill-in') qui fait que la matrice factorisée comporte beaucoup plus de termes non nuls que le matrice initiale (communément x25 fois plus dans nos études). Ce point crucial cherche à restreindre l'encombrement mémoire du stockage de cette factorisée mais aussi les coûts en flop de ses nombreuses manipulations ultérieures (descente-remontées).

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

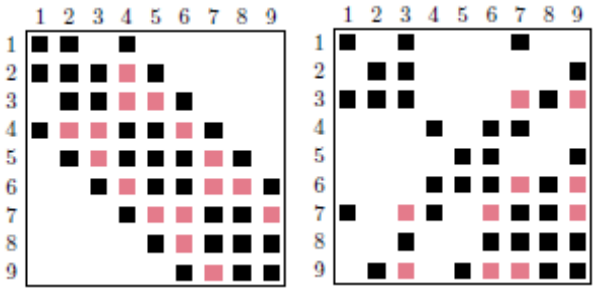


Figure 7.1.1_ Première ligne, de gauche à droite : une matrice creuse initiale, la même matrice réordonnée et l'arbre d'élimination associé à cette dernière. Deuxième ligne, de gauche à droite : les factorisées correspondant aux matrices initiale et renumérotée.

La **deuxième idée forte de la méthode multifrontale** est de réunir un maximum de variables (on parle d'**amalgamation**) afin de **constituer des «fronts» (denses)** dont le traitement numérique sera beaucoup plus efficace (*via* des routines de type BLAS).
Quitte à remplir le sous-bloc matriciel correspondant par des vrais zéros et à les manipuler tel quels[40]. C'est par exemple ce qui est fait dans l'arbre de la figure 7.1.2 entre les variables amalgamées 7, 8 et 9. Il existe de nombreuses techniques d'amalgamation basée sur des critères de graphe, des aspects numériques ou des considérations informatiques (par ex. parallélisme distribué).

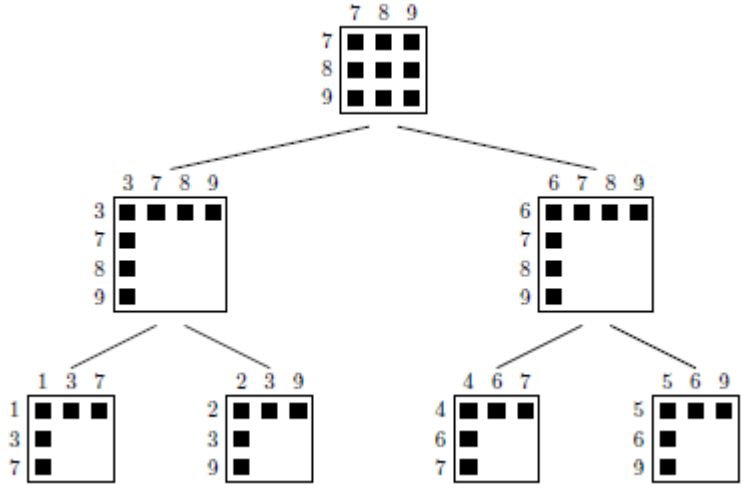


Figure 7.1.2_ Arbre d'élimination avec ses blocs matriciels et un choix de « fronts ».

Remarque:

• Par soucis de simplification, on n'abordera pas dans ce résumé les autres traitements numériques qui interviennent souvent dans le processus: mise à l'échelle des termes ('scaling'), permutation des colonnes et pivotage statique/dynamique. Comme le « diable est souvent dans les détails », ce sont pourtant ces traitements connexes qui ont complexifié les travaux numériques et les développements informatiques de Clément. Ils sont essentiels au bon déroulement de beaucoup de nos simulations industrielles avec Code_Aster .

7.2 Arbre d'élimination

Le traitement algorithmique de l'arbre commence du bas (au niveau des «feuilles») vers le haut (vers «la racine»). **Au sein de chaque feuille ou de chaque «nœud» de branche on associe un front dense**. Dans un front on distingue deux types de variables :

40 Sans compter le sur-remplissage induit. C'est un peu contradictoire avec l'étape de renumérotation précédente, mais en général, cette amalgamation est très bénéfique pour l'ensemble du processus.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

• les **'Fully Summed' (FS)** qui comme leur nom l'indique sont complètement traitées et ne vont plus être mises à jour;
• les '**Non Fully Summed' (NFS)** qui elles attendent toujours des contributions d'autres branches de l'arbre.

Le premier type de variables peut être complètement «**éliminé**»: les termes de la matrice factorisée les concernant (lignes de $\mathbf{U}$ et colonnes de $\mathbf{L}$) sont calculés et stockés[41] une fois pour toute. Le **second type produit un bloc de contributions**[42] (noté $\mathbf{CB}$ pour 'Contribution Blocks') qui va s'additionner au niveau supérieur de l'arbre avec d'autres CBs associés aux mêmes variables (cf. figure 7.2.1).

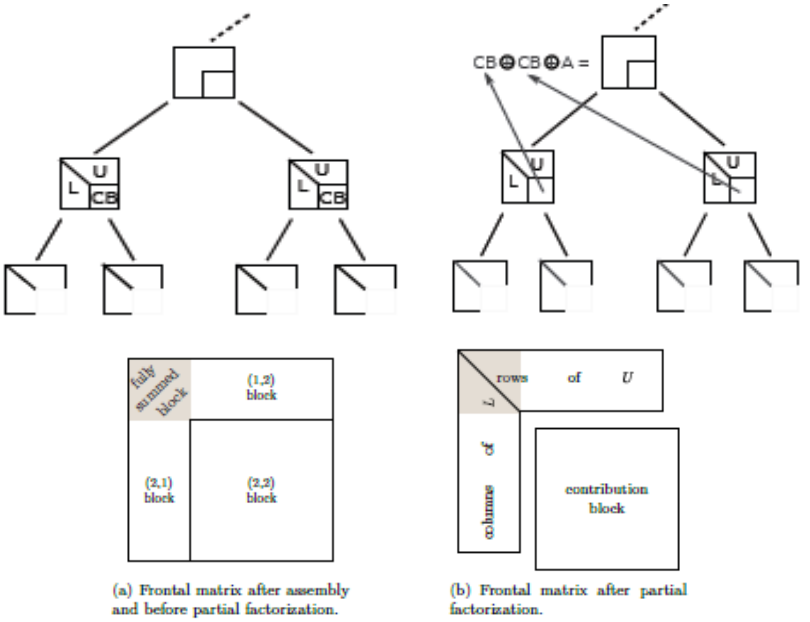


Figure 7.2.1_ Structure générale d'un « front » avant et après son traitement.

A leur tour, certaines variables associées à ces blocs vont être éliminées et d'autres, au contraire, vont continuer à participer activement au processus en fournissant à nouveaux des CBs. Et ainsi de suite… jusqu'à la racine de l'arbre. A ce dernier stade, il n'y a plus que des variables éliminées et plus aucun CB !

Ainsi dans les fronts associés aux feuilles de gauche de la figure 7.2.1, les variables 1 et 2 sont FS, tandis que les variables 3, 7 et 9 sont NFS. Ces dernières fournissent des CBs qui vont être cumulés dans le front du niveau supérieur regroupant les variables 3, 7, 8 et 9. Ce dernier va « soulager » le processus algorithmique de la variable FS 3 (elle va être éliminée), tandis que les NFS 7, 8 et 9 vont continuer leur cheminement dans l'arbre.

7.3 Traitements algorithmiques

Dans le cas d'une matrice dense générale $\mathbf{A}$, la factorisation de Gauss conduit à construire itérativement les termes de $\mathbf{L}$ et ceux de $\mathbf{U}$ suivant un algorithme du type 1.1 (cf. figure 7.3.1). Ici, pour simplifier, on ne tient ni compte des termes nuls (ou trop petits), ni d'aspects numériques type pivotage, permutation ou scaling.
A chaque étape k correspond un pivot (supposé non nul) qui va conduire la mise à jour du bloc $\mathbf{A}_{k+1:n,k+1:n}$ sous-jacent, ainsi que la construction de la colonne et de la ligne correspondante de $\mathbf{L}$ et de $\mathbf{U}$.

Cet algorithme requiert donc deux types d'opérations:
• Une **étape de factorisation** (notée '*Factor*') qui utilise le pivot diagonal $a_{k,k}$ pour construire la k-ième partie de la factorisée et ainsi éliminer la k-ième variable (elle sera dite FS).

41 En RAM puis sur disque si l'OOC est activé.
42 Objet de calcul géré uniquement en RAM.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

- Une **étape de mise à jour** (notée '*Update*') qui construit le bloc de contribution (CB).

La complexité algorithmique de l'ensemble et son coût mémoire sont respectivement en $\vartheta(n^3)$ et en $\vartheta(n^2)$. Ces seuls chiffres illustrent l'impact de cette étape sur les performances de nos simulations (même si celles-ci sont effectuées en 'sparse').

Algorithm 1.1 Dense LU factorization.

1: ▶ **Input:** a square matrix A of size n; $A = [a_{ij}]_{i=1:n,j=1:n}$
2: ▶ **Output:** A is replaced with its LU factors
3:
4: **for** $k = 1$ **to** $n-1$ **do**
5: Factor: $a_{k+1:n,k} \leftarrow \frac{a_{k+1:n,k}}{a_{kk}}$
6: Update: $a_{k+1:n,k+1:n} \leftarrow a_{k+1:n,k+1:n} - a_{k+1:n,k} \times a_{k,k+1:n}$
7: **end for**

Algorithm 1.3 Dense Block LU factorization.

1: ▶ **Input:** a NB × NB-block matrix A of size n; $A = [A_{I,J}]_{I=1:NB,J=1:NB}$
2: ▶ **Output:** A is replaced with its LU factors
3:
4: **for** $K = 1$ **to** NB **do**
5: Factor: $A_{K,K} \leftarrow L_{K,K}U_{K,K}$
6: Solve (compute U): $A_{K,K+1:NB} \leftarrow L_{K,K}^{-1} \cdot A_{K,K+1:NB}$
7: Solve (compute L): $A_{K+1:NB,K} \leftarrow A_{K+1:NB,K} \cdot U_{K,K}^{-1}$
8: Update: $A_{K+1:NB,K+1:NB} \leftarrow A_{K+1:NB,K+1:NB} - A_{K+1:NB,K} \cdot A_{K,K+1:NB}$
9: **end for**

Algorithmes 7.3.1_ Exemples d'algorithmes de factorisation denses, en scalaire et par blocs.

Afin d'optimiser les coûts on travaille **généralement sur des blocs et non sur des scalaires** (cf. algorithme 1.3). Les données à rassembler en mémoire RAM, au même instant, sont de plus petites tailles et les opérations algébriques vectorielles et matricielles sont beaucoup plus efficaces (*via* des routines de type BLAS optimisées).
Les types d'opérations à effectuer restent inchangés:

- Factorisations locales de blocs diagonaux ('*Factor*'),
- Descente-remontées (notée '*Solve*') pour construire effectivement les blocs colonnes/lignes de la factorisée,
- Mise à jour ('*Update*') par blocs de la sous-matrice.

Dans les arbres d'élimination présentés précédemment, c'est **ce type d'opération qui est conduit au sein de chaque front**, puis entre chaque front et son « père ». La compression low-rank va avoir pour objectif de réduire leur complexité algorithmique ainsi que leur empreinte mémoire (pic RAM et consommation disque).

7.4 Gestion de la mémoire

Sur ce dernier point, souvent le plus crucial pour nos études, t**out le problème réside dans la «gestion différée des CBs»**. Lors de l'étape k de l'algorithme 1.3, on doit garder en RAM non seulement le front «actif», mais aussi les CBs en attente de traitement et les (éventuels) facteurs non déchargés sur disque (si l'OOC est activé). Ils constituent la mémoire active du processus (zone bleue et verte de la figure 7.4.1).

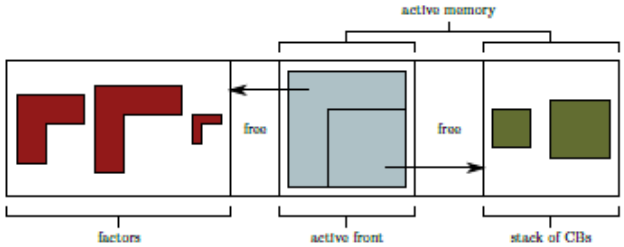


Figure 7.4.1_ Gestion mémoire des différents éléments manipulés par la multifrontale.

Cette **mémoire active** est gérée comme une pile ('stack'). Elle **fluctue constamment**. Elle croit lorsqu'un front est chargé. Puis elle décroit au fur et à mesure que des CBs associés à ce front sont consommés. Enfin elle recroit avec les nouveaux facteurs et le nouveau CB résultant de ce front.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

Les nouveaux facteurs sont ensuite éventuellement recopiés sur disque. **L'empreinte mémoire de ces facteurs** (zone rouge à gauche du graphique) est plus simple à analyser : **elle ne fait qu'enfler au fur et à mesure de la remontée de l'arbre !**

Dans tous les cas, **ces deux zones mémoire ne sont pas faciles à prédire *a priori***. Elles dépendent notamment de la renumérotation, de la construction de l'arbre d'élimination mais aussi des prétraitements numériques. C'est une des tâches effectuées par l'étape d'analyse de MUMPS. Elle est utile à la fois à la multifrontale pour allouer efficacement ses zones mémoire, mais aussi pour le code appelant afin de gérer au mieux ses propres objets internes[43]. La gestion optimisée de ces éléments se complique encore du fait des traitements numériques qui sont effectués dynamiquement en cours de factorisation: par exemple, l'organisation du pivotage et la distribution des données en parallèle.

C'est le pic de cette mémoire active qui constitue l'enjeu essentiel de cette thèse. Il est souvent très supérieur à la taille de la factorisée. Pourtant celle-ci est déjà souvent en 500N dans nos études actuelles (avec N la taille du problème). Ainsi le traitement d'un matrice comportant 1 million d'inconnues requiert au moins 500*1*12[44]=6Go de RAM (en réel double précision). Et ce chiffre a tendance à augmenter avec la taille du problème (on passe de 500N à 1000N voire plus).
On peut donc atteindre des pics RAM que même la distribution parallèle des données combinée à l'OOC ne pourra pas gérer efficacement[45]. **Donc toute réduction significative de ce pic serait un gain très appréciable pour nos grosses études actuelles et futures**.

7.5 Approximation low-rank

Une matrice dense $\mathbf{A}$, de taille mxn, est dite « de rang faible » ('low-rank') d'ordre k (<$min(m,n)$) lorsqu'elle peut se décomposer sous la forme

$$\mathbf{A} = \mathbf{U}.\mathbf{V}^T + \mathbf{E} \tag{7.5-1}$$

avec $\mathbf{U}$ et $\mathbf{V}$ des matrices beaucoup plus petites (respectivement mxk et kxn) et $\mathbf{E}$ une matrice mxn «négligeable» ($\|\mathbf{E}\| < \varepsilon$). Cette notion de «rang numérique» ne doit pas être confondue avec la notion de rang algébrique

Lors des manipulations ultérieures on fait alors l'approximation

$$\mathbf{A} \approx \mathbf{U}.\mathbf{V}^T \tag{7.5-2}$$

en pariant (souvent sous contrôle), qu'elle aura peu d'incidence sur le processus global. Cette approximation est donc dépendante du paramètre de compression ε.
C'est ce paramètre que l'on va faire varier, en pratique, pour adapter les gains du low-rank à la situation. Par exemple:

- avec $\varepsilon \leq 10^{-9}$ on a compression avec une perte de précision légère et pilotable. On peut bien sûr continuer à utiliser la multifrontale en tant que **solveur direct**. Pour retrouver une précision identique au cas full-rank, une ou deux itérations de raffinement itératif sont suffisants[46].
- avec $\varepsilon > 10^{-9}$ la compression est meilleure mais la perte de précision peut être importante[47]. On peut alors utiliser la factorisée pour construire un **préconditionneur,** plus ou moins «frustre», accélérant un solveur itératif de Krylov.

D'autre part, hormis des gains évidents en terme de stockage, la manipulation de telles matrices peut s'avérer très avantageuse: le rang de la somme est inférieur (au pire) à la somme des rangs et le rang du produit est

43 Cf. mot-clé *Code_Aster* SOLVEUR/GESTION_MEMOIRE.
44 Le chiffre multiplicatif 12 provient des 8 octets que consomme le stockage du terme matriciel réel et des 4 de son indice de ligne.
45 Goulet d'étranglement de quelques traitements séquentiels, détérioration des speed-ups et coûts des I/O.
46 Parfois en non linéaire la correction apportée par le processus de Newton est suffisante pour assurer la convergence globale du calcul. Il prend une ou deux itérations de Newton supplémentaires mais on évite ainsi les itérations de raffinement itératif en post-traitement de chacune des descente-remontées.
47 Plus embêtant encore est le fait que les «outputs» usuels du solveur direct peuvent être faussés et donc inutilisables par un algorithme englobant: détection de singularité, calcul de déterminant, critère de Sturm...

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

inférieur au minimum des rangs. Une fois décomposées, **la manipulation de matrices low-rank peut donc être (relativement) contrôlée afin d'optimiser la compression du résultat**.
Ainsi le produit de deux matrices dense de taille nxn et de rang k, avec $k << n$, réduit la une complexité algorithmique de l'opération de $\vartheta(n^3)$ à $\vartheta(kn^2)$; son empreinte mémoire passant elle de $\vartheta(n^2)$ à $\vartheta(kn)$.

Une matrice transformée sous forme low-rank est dite **«rétrogradée»** (‘**demoted**’) ou **compressée**. La transformation inverse, aux approximations près, **«promeut»** la matrice (‘**promoted**’) ou la **décompresse**.
Pour garder la même terminologie lorsqu'on manipule de manière standard une matrice (sans exhiber une décomposition du type (7.5-1)), on dit que celle-ci est de **rang plein (‘full-rank’)**.

La «décomposition low-rank» d'une matrice peut être effectuée de différentes manières:
- **SVD**,
- Rank-Revealing QR (**RRQR**),
- Adaptative Cross Approximation (**ACA**),
- **Echantillonnage aléatoire**.

Dans cette thèse Clément utilise principalement la deuxième solution, moins précise qu'une classique SVD mais bien moins coûteuse. Le tout étant que cette compression soit mutualisée entre différents traitements et qu'elle permette une compression maximale. Idéalement il faudrait donc que le rang obtenu vérifie une condition du type :

$$k(m+n) << mn \qquad (7.5\text{-}3)$$

Une partie du travail de Clément a consisté à mettre au point des heuristiques adaptées à la multifrontale afin d'essayer de respecter le plus souvent possible ce critère.

7.6 Multifrontale 'Block Low-Rank' (BLR)

L'objectif de cette thèse est d'exploiter ce type de compression en décomposant sous forme low-rank les plus gros fronts de l'arbre d'élimination d'une multifrontale. Car ce sont ces sous-blocs matriciels denses qui génèrent le plus de flop et qui grèvent le pic RAM (du fait de tous les CBs qu'ils vont agglomérer). En général ces gros fronts se situent dans les derniers niveaux de l'arbre d'élimination.

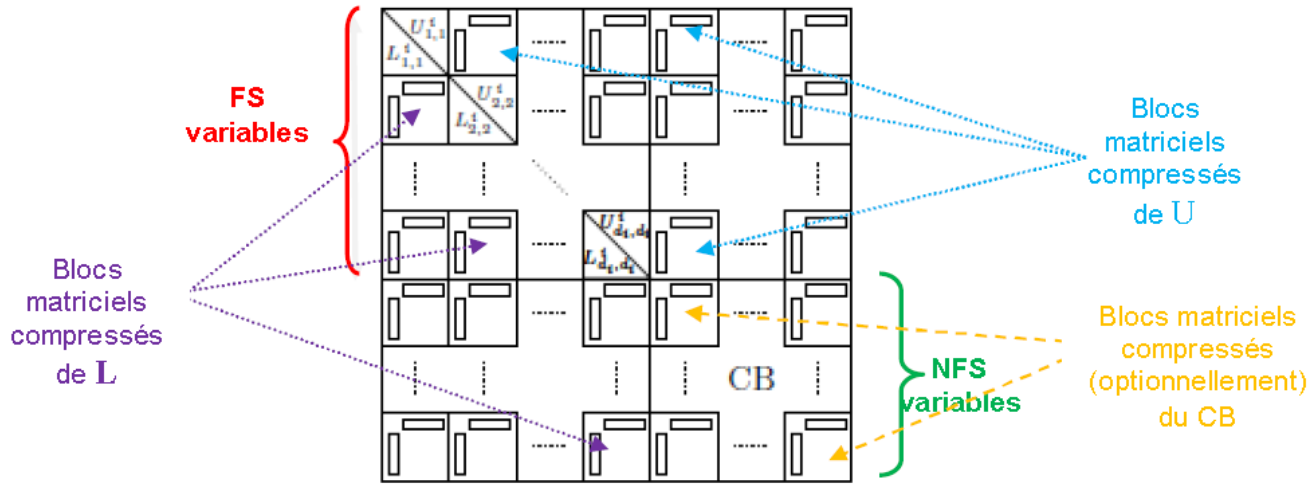


Figure 7.6.1_ Structure d'un front compressé après sa factorisation.

Comme illustré ci-dessous (cf. figure 7.6.1) on va donc, premièrement, **décomposer en colonnes et en lignes de sous-blocs** (décomposition par ‘panels’), les blocs matriciels correspondant aux **quatre types de termes des fronts** (suivant la terminologie de la figure 7.2.1):
- FSxFS (‘block (1,1)’),
- FSxNFS (‘block (1,2)’),
- NFSxFS (‘block (2,1)’),

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

- NFSxNFS (‘block (2,2)’).

Puis chacun de ces sous-blocs va être compressé en low-rank suivant la formule (7.5-1). Du moins lorsque cette compression est licite[48]. Sauf les sous-blocs diagonaux de la sous-partie FSxFS qui restent en full-rank (pour optimiser leurs manipulations ultérieures).

On **adapte la taille des panels** de manière à ce qu’elle soit :

- **Suffisamment grande** pour tirer parti de la performance des noyaux BLAS (invariant de boucle, minimisation des défauts de caches et gestion optimisée de la hiérarchie mémoire);
- **Pas trop grande** pour ménager de la souplesse dans les manipulations numériques (pivotage, scaling) et informatiques (parallélisme distribué) ultérieures;
- **De taille moyenne** afin d’optimiser les coûts des communications MPI (latence *versus* bande-passante).
- **De taille moyenne** afin de ménager de la souplesse dans le regroupement de variables qui va suivre (le ‘clustering’). Cette réorganisation devant générer un maximum de compression.
- **Pas trop grande** pour ne pas coûter trop cher en SVD ou en RRQR.

Mais tout le problème réside dans le fait que ces sous-blocs matriciels n’ont aucune raison d’être low-rank ! Même si ils sont de grosse taille, les parties de fronts ‘demoted’ peuvent s’avérer de rang plein. Les coûts des SVDs ou des RRQRs ne seront alors pas compensés et l’objectif de compression s’avérera compromis !
En s’inspirant des travaux déjà effectués par d’autres techniques de compression (matrices hiérarchique de type H, H2, HSS/SSS…), il a été mis au point des critères pour décomposer les variables composant un front et les renuméroter afin qu’elles génèrent des sous-blocs matriciels low-ranks. Ce critère est basé sur la constatation empirique suivante :

Plus deux blocs de variables sont distants, plus le rang numérique du bloc matriciel qu’il implique devient faible.

Cette **condition dite «d’admissibilité»** a été testée sur certains problèmes modèles issus de discrétisation d’EDP elliptique et elle est illustrée à la figure 7.6.2. Avec la terminologie de la théorie des graphes elle peut se reformuler sous la forme suivante :

$$diam(\sigma) + diam(\tau) << dist(\sigma,\tau) \qquad (7.6\text{-}1)$$

avec $diam$ et $dist$, les notions usuelles de diamètre et de distance dans le graphe associé au front traité.

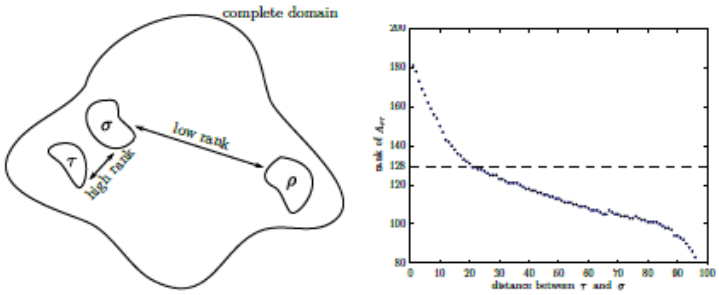


Figure 7.6.2_ Evolution du rang «numérique» de deux groupes de variables de taille homogène en fonction de la distance (au sens des graphes) qui les sépare.

Ainsi, sur l’exemple de la figure ci-dessous (cf. fig. 7.6.3), on obtient avec le découpage des variables du front en tranches un gain en nombre de termes de 17%. C’est-à-dire que, en moyenne, chaque sous-bloc admet un rang numérique k tel que $k(m+n) = 0.83mn$.

Tandis qu’avec le découpage en damier, le gain atteint 47%: $k(m+n) = 0.57mn$. Cet écart est dû à la plus grande régularité du second sur le premier. Les sous-blocs homogènes en forme de carré ont des diamètres rapidement plus faibles que la distance entre les sous-blocs. Avec leur forme allongée, ce n’est évidemment pas le cas des sous-blocs résultant du découpage en tranche !

48 Lorsque ces sous-blocs répondent à certains critères fonctionnels. Par exemple ils doivent être de taille suffisante pour qu'on tente une compression. D'autre part les gains de la compression doivent être suffisamment importants.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

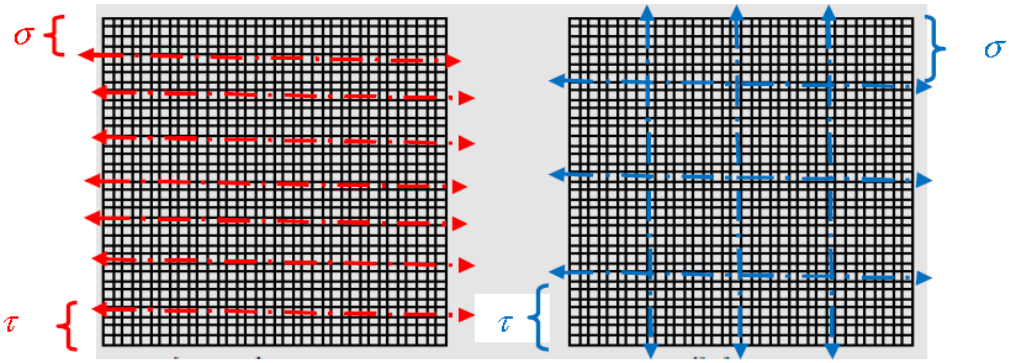


Figure 7.6.3_ Exemple de deux types de clustering sur un front dense: en tranche et en damier.

7.7 Quelques éléments complémentaires

C'est donc sur un critère d'admissibilité de ce type que MUMPSt va baser son clustering des variables composant un front. Il a beaucoup travaillé pour rendre cette «opération clé» de la méthode compatible avec les contraintes numérico-informatiques de la multifrontale et de l'outil MUMPS ; tout en veillant à ménager un maximum de compression et en limitant son surcoût (en temps).

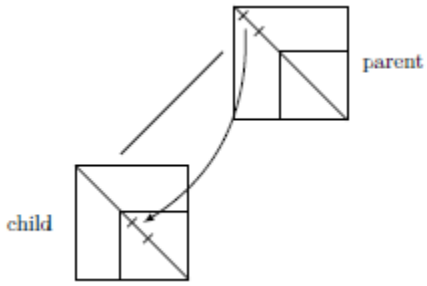


Figure 7.7.1_ Clustering de type «hérité» entre les variables NFS d'un front et celles FS d'un front antérieur.

Exemple de sophistication: **ce clustering est opéré séparément sur les deux types de variables, les FS et les NFS**. Comme en parcourant l'arbre d'élimination, les variables NFS d'un front deviennent les FS des fronts des niveaux supérieurs, on décompose cette tâche. D'autre part on ne recommence pas complètement ces découpages à chaque front. On mutualise certaines informations afin de limiter les surcoûts (en temps). Cet **algorithme particulier de clustering est appelé « hérité » ('inherit')** par opposition à l'**algorithme standard dit «explicite» ('explicit')** pour lequel on recommence les deux clusterings à chaque manipulation de nouveau front.

Le surcoût de la variante « explicite » de clustering peut être prohibitif sur les très gros problèmes (plusieurs fois le coût de la totalité de la phase d'analyse !) alors que le coût du clustering optimisé reste raisonnable : seulement quelques pourcents de cette phase d'analyse. Les gains low-rank des deux variantes sont très proches par contre l'implémentation de la version optimisée est plus complexe.
Enfin ces clusterings sont effectués par des **outils standards, METIS ou SCOTCH**. Mais on ne les applique pas directement sur les variables des fronts mais sur des halos les englobant. Cette astuce permet d'orienter convenablement ces découpages afin qu'ils produisent des groupes de nœuds homogènes contigües.

D'autre part, pour être plus précis, le traitement d'un front comporte cinq étapes (déjà évoquées dans les algorithmes de la fig. 7.3.1). On commence par **découper les variables en deux niveaux emboîtés de panels:**

- le premier niveau, le plus grand, celui qui est à l'extérieur (appelé '**outer panel**' ou '**BLR panel**'), résulte du clustering low-rank;
- tandis que le second, celui qui est à l'intérieur du précédent ('**inner panel**' ou '**BLAS panel**') regroupe les variables par sous-paquets de 32, 64 ou 96 variables contigües afin de nourrir plus efficacement les noyaux de calculs (et réduire le coût des communications MPI et des I/Os).

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

Puis, on a deux boucles emboîtées : la première sur les panels BLR et la seconde sur les panels BLAS. Pour un panel BLAS donné, on effectue :
- l'étape de Factorisation (**F**),
- l'étape de Solve (**S**),
- l'Upgrade Interne (**UI**) entre tous les sous-panels suivants du panel BLR,
- l'Upgrade Externe (**UE**) entre tous les panels BLR suivants.

L'ordre standard des opérations peut donc être noté grossièrement **FSUU**. Suivant le niveau auquel on fait intervenir la Compression low-rank (**C**) on distingue alors 4 variantes :
1. FSUUC,
2. FSUCU,
3. **FSCUU,**
4. FCSUU.

task	operation type	dense	low-rank
Factor (F)	$B = LU^T$	$(2/3)r^3$	$(2/3)r^3$
Compress (C)	$C = XY^T$	kr^2	—
Solve (S)	$D = X(Y^T L^{-1})$	r^3	kr^2
Update (U)	$D = D - X_1(Y_1^T X_2)Y_2^T$	$2r^3$	$2kr^2$

(r=block size, k=rank)

Tableau 7.7.2_ Comparaison des coûts des différentes étapes en dense (full-rank) et en low-rank.

C'est la variante n°3, FSCUU, qui est industrialisée les versions consortium de MUMPS.
Compte-tenu des coûts relatifs de chacune des opérations (cf. tableau 7.7.2) et des **contraintes de robustesse de l'outil, c'est cette variante qui a été privilégiée**. Elle permet de réduire significativement les coûts globaux tout en ménageant un maximum de précision dans les deux premières étapes (F et S). Car celles-ci sont cruciales dans la gestion de plusieurs ingrédients numériques sophistiqués (scaling, pivotage, détection de singularité, calcul de déterminant, test de Sturm…). Dans un premier temps, on a donc préféré les impacter le moins possible par cette compression. Aussi celle-ci est effectuée juste après. Et ses gains (et ses approximations éventuelles) n'impacteront ainsi que les updates internes et externes (UI et UE).

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)