Notice d'utilisation des solveurs linéaires

1 But

Les **solveurs linéaires sont en fait omniprésents dans le déroulement des opérateurs de *Code_Aster*** car ils sont souvent enfouis au plus profond d'autres algorithmes numériques : schéma non linéaire, intégration en temps, analyse modale etc. Ils en consomment souvent la majeure partie du temps CPU et de la mémoire. Le choix et le paramétrage complet du solveur linéaire requis s'effectue *via* le mot-clé facteur SOLVEUR. Il est présent dans la plupart des commandes de calcul (STAT_NON_LINE, THER_LINEAIRE, CALC_MODES, ...).
Cette **notice d'utilisation est complémentaire de la documentation utilisateur** du mot-clé SOLVEUR[U4.50.01]. Elle se veut un maillon intermédiaire entre la « simple » documentation de mot-clé et celle théorique (chaque solveur a une documentation de Référence).

La première partie de cette note donne une **vision d'ensemble** des différents solveurs linéaires disponibles, de leur périmètre d'utilisation et de leur performances moyennes en terme de robustesse et de consommations CPU/mémoire. Le chapitre suivant regroupe quelques **conseils** pour utiliser au mieux les paramètres de SOLVEUR en fonction des cas de figure: la taille du système, ses propriétés numériques, les consommations CPU et mémoire... Pour finir, on détaille certains paramètres de SOLVEUR afin d'aider l'utilisateur dans un **usage avancé** de cette fonctionnalité du code.
Les **problématiques connexes** d'amélioration des performances (RAM/CPU) d'un calcul et de l'utilisation du parallélisme sont aussi brièvement abordées. Elles font l'objet de notices spécifiques, respectivement, [U1.03.03] et [U2.08.06].

Table des Matières

2 Les différents solveurs linéaires disponibles

Ces **solveurs linéaires sont en fait omniprésents dans le déroulement des opérateurs** de *Code_Aster* car ils sont souvent enfouis au plus profond d'autres algorithmes numériques : schéma non linéaire, intégration en temps, analyse modale etc. Ils en consomment souvent la majeure partie du temps CPU et de la mémoire. Le choix et le paramétrage complet du solveur linéaire requis s'effectue *via* le mot-clé facteur SOLVEUR. Il est présent dans la plupart des commandes de calcul (STAT_NON_LINE, THER_LINEAIRE, CALC_MODES, ...).

Ce mot-clé permet de choisir entre les deux classes de solveurs : les directs et les itératifs. Concernant les **directs**, on dispose du classique algorithme de «Gauss» (SOLVEUR/METHODE='LDLT'), d'une factorisation multifrontale ('MULT_FRONT') et d'une externe ('MUMPS'). Pour les **itératifs**, il est possible de faire appel à un gradient conjugué ('GCPC') ou à certains outils de la librairie publique PETSc ('PETSC').

Seuls MULT_FRONT, MUMPS, PETSC sont **parallélisés**. Le premier en OpenMP, les autres en MPI. Mais tous les solveurs sont compatibles avec un traitement parallèle des calculs élémentaires et des assemblages. Et ce, que ces traitements soient initiés juste avant l'utilisation du solveur linéaire proprement dit, ou, dans un autre opérateur (par exemple de pré/post-traitements).

Détaillons un peu le fonctionnement de chacun d'entre eux:

Solveurs directs

/'MUMPS'

Solveur direct de type multifrontale avec pivotage. Ce solveur est obtenu en appelant le **produit externe MUMPS** développé par CERFACS/IRIT/INRIA/CNRS (voir copyright plus bas). Le stockage matriciel hors solveur est MORSE. En entrée du solveur, on fait la conversion au format interne de MUMPS: i, j, K_{ij}, centralisée ou distribuée.
Pour *Code_Aster*, **son intérêt principal réside dans sa capacité à pivoter** lignes et/ou colonnes de la matrice lors de la factorisation en cas de pivot petit. Cette possibilité est utile (voire indispensable) pour les modèles conduisant à des matrices non définies positives (hors conditions aux limites); Par exemple, les éléments "mixtes" ayant des degrés de liberté de type "Lagrange" (éléments incompressibles...).
Informatiquement, ce solveur pose deux problèmes: il nécessite un compilateur fortran90 et il faut partager la mémoire entre JEVEUX et la bibliothèque MUMPS.
Cette **méthode est parallélisée en mémoire distribuée** (MPI) et peut être exécutée sur plusieurs processeurs (*via* l'interface Astk menu Options/Optionsdelancement/mpi_nbcpu&mpi_nbnoeud). On peut aussi profiter du parallélisme en mémoire partagée des librairies d'algèbre linéaire de bas niveau (BLAS).
En parallèle, MUMPS distribue naturellement ses données (matrice, factorisée...) entre les processeurs. Ce qui permet de réduire l'occupation mémoire nécessaire, par processeur, pour lancer le calcul. Cette consommation RAM peut encore plus être réduite *via* le paramètre GESTION_MEMOIRE='OUT_OF_CORE' de MUMPS. En terme de mémoire, le goulet d'étranglement peut alors se trouver au niveau de l'espace JEVEUX. Pour réduire ce dernier, on peut distribuer la matrice *Aster* (MATR_ASSE) *via* l'option MATR_DISTRIBUEE.

/'MULT_FRONT' **Solveur direct de type multifrontale développé en interne EDF R&D**. Le stockage matriciel est MORSE (ou 'CSC' pour 'Compressed Sparse Column') et donc proscrit tout pivotage. Cette méthode est parallélisée en mémoire partagée (OpenMP) et peut être exécutée sur plusieurs processeurs (*via* l'interface Astk menu Options/Options de lancement/ncpus). La matrice initiale est stockée dans un seul objet JEVEUX et sa factorisée est répartie sur plusieurs, donc elle peut être déchargée partiellement et automatiquement sur disque.

/'LDLT' **Solveur direct avec factorisation de Crout** par blocs (sans pivotage) **développé en interne EDF R&D**. Le stockage matriciel hors solveur est MORSE. En entrée du solveur, on fait la conversion au format interne de LDLT: 'ligne de ciel' ('SKYLINE'). On a une pagination de la mémoire complètement paramétrable (la matrice est décomposée en blocs gérés en mémoire de façon indépendante et déchargés sur disque au fur et à mesure) qui permet de passer de gros cas mais qui se paye par des accès disques coûteux.
En outre, ce solveur permet de ne factoriser que partiellement la matrice. Cette possibilité est "historique". Elle permet de factoriser la matrice en plusieurs "fois" (plusieurs travaux) voire de modifier à la volée les dernières lignes de cette factorisée. Aujourd'hui, on n'imagine pas bien l'intérêt de cette fonctionnalité hormis pour certaines méthodes (dite discrètes) de contact-frottement où l'on a, à dessein, placé dans les dernières lignes de la matrice les termes concernant les noeuds susceptibles d'être en contact. Ainsi, au fur et à mesure des itérations d'appariement, les relations entre ces noeuds changeant, on efface puis recalcule que ces dernières contributions de la factorisée. C'est un exemple typique où l'usage astucieux d'un algorithme assez frustre peu amener des gains majeurs (en temps).

Solveurs itératifs

/'GCPC' **Solveur itératif de type gradient conjugué avec préconditionnement $ILU(k)$ ou basé sur une factorisée simple précision (via MUMPS).**
Le stockage de la matrice est alors MORSE. Avec le Cholesky incomplet, la matrice initiale et sa factorisée incomplète sont stockées, chacune, dans un seul objet JEVEUX.
Avec la factorisée simple précision, le préconditionneur est beaucoup plus coûteux (en CPU/RAM), mais il est effectué en simple précision et son calcul peut être mutualisé pendant plusieurs résolutions (problème de type multiples seconds membres, par ex. STAT_NON_LINE).

/'PETSC' **Solveurs itératifs issus de la librairie externe PETSc** (Laboratoire Argonne). Le stockage matriciel hors solveur est MORSE. En entrée du solveur, on fait la conversion au format interne de PETSc: 'CSR' pour 'Compressed Sparse Row'. PETSc alloue des blocs de lignes contigus pas processeur. Cette **méthode est parallélisée en mémoire distribuée** (MPI) et peut être exécutée sur plusieurs processeurs (*via* l'interface Astk menu Options/ Options de lancement /mpi_nbcpu&mpi_nboeud).

Attention : les solveurs PETSC et MUMPS étant incompatibles en séquentiel, on privilégie généralement MUMPS. Pour utiliser PETSC, il faut donc souvent lancer une version parallèle de *Code_Aster* (quitte à ne solliciter qu'un seul processeur).

	Périmètre Solveur	Robustesse	CPU	Mémoire	Détails
Direct					
MULT_FRONT	Solveur universel. A déconseiller pour les modélisations	+++	Séq: ++ //: ++ (speed-	+ OOC[1]	Solveur de référence. Sur 4 proc.

	Périmètre Solveur	Robustesse	CPU	Mémoire	Détails
	nécessitant du pivotage (EF mixtes de X-FEM, incompressible...).		up~2)		
MUMPS CENTRALISE ou DISTRIBUE	Solveur universel.	+++	Séq: ++ //: +++ (sp~12)	--- IC + OOC	Plutôt en linéaire Sur 16/32 proc.
LDLT	Solveur universel (mais très lent sur de gros cas). A déconseiller pour les modélisations nécessitant du pivotage (EF mixtes de X-FEM, incompressible...) . Factorisation partielle possible (contact discret).	+++	Séq: +	+ OOC	Plutôt les petits cas ou les cas de taille moyenne pouvant tirer partie de factorisation partielle.
Itératif					
GCPC	Problèmes symétriques réels sauf ceux requérant obligatoirement une détection de singularité (calcul modal, flambement).	- (LDLT_INC) + (LDLT_SP)	Séq: +++	+++ (LDLT_INC) ++ (LDLT_SP)	Ne pas trop augmenter le niveau de préconditionnement. Parfois très efficace (thermique…) surtout en non linéaire avec LDLT_SP.
PETSC	Idem GCPC mais compatible avec le non symétrique.	- (LDLT_INC) + (LDLT_SP)	Séq: +++ //: +++ (sp~4)	++ IC	Algorithmes plutôt robustes: GMRES.

Figure 2.-1: Synoptique des solveurs linéaires disponibles dans Code_Aster.

Remarque:

• Pour être complètement exhaustif, on peut préciser que quelques (rares) opérations numériques sont opérées avec un paramétrage solveur «figé ». Celui-ci est donc inaccessible aux utilisateurs (sans surcharge logiciel). Parmi celles-ci, on trouve certaines résolutions des méthodes discrètes de contact-frottement (réservées à LDLT), les recherche de modes rigides en modal, les calculs de modes d'interfaces (réservées à MUMPS ou à LDLT)... A chaque fois des raisons fonctionnelles ou de performance expliquent ce choix peu transparent.

1 OOC pour 'Out-Of-Core'. C'est-à-dire qu'on va libérer de la mémoire RAM en déchargeant sur disque une partie des objets. Cela permet de suppléer au swap système et de traiter des problèmes bien plus gros. Suivant l'algorithmique, ces accès disques supplémentaires peuvent être pénalisant. Le mode de gestion opposé est «l'In-Core» (IC). Tous les objets informatiques restent en RAM. Cela limite la taille des problèmes accessibles (au swap système près) mais privilégie la vitesse.

3 Quelques conseils d'utilisation

Si on souhaite[2] changer de solveur linéaire ou adapter son paramétrage, plusieurs questions doivent être abordées: Quel est le type de problème que l'on souhaite résoudre ? Qu'elles sont les propriétés numériques des systèmes linéaires rencontrés ? etc.
On liste ci-dessous et, de manière non exhaustive, plusieurs questions qu'il est intéressant de se poser lorsqu'on cherche à optimiser les aspects solveurs linéaires. Bien sûr, certaines questions (et réponses) sont cumulatives et peuvent donc s'appliquer simultanément.

En bref:

La méthode par défaut reste la multifrontale interne MULT_FRONT. Mais pour pleinement bénéficier des gains CPU et RAM que procure le **parallélisme**, ou pour résoudre un **problème numériquement souvent difficile** (X-FEM, incompressibilité, THM), on préconise l'utilisation du produit externe **MUMPS**.
Pour résoudre un problème de **très grande taille** (> 10^6 degrés de liberté) le recourt à PETSC + MUMPS parallèles peut permettre de passer le calcul malgré les limitations mémoire de la machine. Ce dernier n'apparaît pas explicitement, il est utilisé en « sous-main » en tant que préconditionneur de PETSC *via* l'option PRE_COND='LDLT_SP'.
Pour aller plus loin dans les **économies en place mémoire** , on peut aussi dégrader le préconditionnement (le calcul sera probablement plus long) en choisissant un des autres préconditionneurs de PETSC (' LDLT_INC '...).
En **non-linéaire, pour gagner en temps** , on peut aussi jouer sur plusieurs paramètres de relaxation (SYME en non symétrique) ou d'interactions « solveur non linéaire/solveur linéaire » (REAC_PRECOND , NEWTON_KRYLOV ..).
Pour des problèmes non linéaires **bien conditionnés** (thermique...), le recours à un MUMPS « relaxé » (MIXER_PRECISION/FILTRAGE_MATRICE) peut amener des gains mémoire très sensibles. De même, en linéaire comme en non linéaire, avec PETSC sans préconditionneur (PRE_COND='SANS').

Pour plus de détails et de conseils sur l'emploi des solveurs linéaires on pourra consulter la notice d'utilisation [U4.50.01] et les documentations de référence associées [R6...]. 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].

Quel est le type de problème à résoudre ?

- Résolution de nombreux systèmes avec la même matrice (problème de type multiples seconds membres[3]) ⇒ solveurs MULT_FRONT ou MUMPS (si possible en désactivant RESI_RELA et avec GESTION_MEMOIRE='IN_CORE').
- Calcul paramétrique standard en linéaire ⇒ un premier essai avec MUMPS en laissant les paramètres par défaut, puis tous les autres runs en débranchant RESI_RELA.
- Calcul paramétrique en non linéaire avec une matrice bien conditionnée ⇒ un premier essai avec MUMPS en jouant sur les paramètres de relaxation (FILTRAGE_MATRICE/MIXER_PRECISION ou SYME si on est en non symétrique). Puis, si un point de fonctionnement optimisé (consommations CPU/RAM) a été dégagé, l'utiliser pour tous les autres runs.
  On peut aussi essayer MUMPS mais, cette fois, comme préconditionneur simple précision du GCPC (LDLT_SP). L'intérêt est alors de ne le réactualiser que périodiquement (REAC_PRECOND).

Quelles sont ses propriétés numériques ?

2 Ou si, tout simplement, il faut jouer sur ce paramètre car le calcul ne passe pas sur la plate-forme logicielle choisie, ou avec des consommations en temps et en mémoire incompatibles avec les contraintes de l'étude.
3 C'est le cas, par exemple, en chaînage thermo-mécanique lorsque les caractéristiques matériaux ne dépendent pas de la température ou, en non linéaire, lorsqu'on réactualise peu souvent la matrice tangente.

- Système linéaire bien conditionné[4] (<10^4) ⇒ solveurs MUMPS+MIXER_PRECISION ou GCPC/PETSC.
- Système linéaire difficile (mauvais conditionnement, éléments finis mixtes, prédominances de Lagranges...) ⇒ solveur MUMPS.

A-t-on besoin d'une solution très précise ?

- On peut se contenter d'une solution très approximée[5] ⇒ solveurs GCPC ou PETSC avec un RESI_RELA=10^{-3}.
- On veut une solution précise ou, au moins, un diagnostic sur sa qualité et sur les difficultés numériques du système à résoudre ⇒ solveur MUMPS en activant NPREC et RESI_RELA (en INFO=2).

Comment optimiser les consommations temps/mémoire RAM du solveur linéaire ?

- Temps ⇒ solveur MUMPS en désactivant l'OOC (GESTION_MEMORE='IN_CORE') voire RESI_RELA. Calcul parallèle distribué éventuellement couplé au parallélisme OpenMP des BLAS (cf. doc. U2 sur la parallélisme).
- Mémoire ⇒ solveur MUMPS en activant GESTION_MEMOIRE='OUT_OF_CORE'/MATR_DISTRIBUE et en mode parallèle distribué (ou solveur itératif si le problème est bien conditionné).
  En non linéaire, si la matrice est bien conditionnée et/ou non symétrique, on peut aussi jouer sur les paramètres de relaxation de MUMPS (FILTRAGE_MATRICE, MIXER_PRECISION et SYME) ou utiliser un solveur itératif (GCPC/PETSC+LDLT_SP). On trouvera plus de détails dans la discussion du §7.2.6.
- Mémoire ⇒ solveur PETSC en activant MATR_DISTRIBUE et en mode parallèle distribué.

Est-ce un problème frontière de très grande taille (> 3.10^6 degrés de liberté) ?

- Solveur robuste ⇒ solveur MUMPS avec les optimisations mémoires précédentes.
- Solveurs de la «dernière chance» ⇒ solveurs itératifs (avec un niveau de préconditionnement faible).

Comment optimiser les performances globales de mon calcul ?

- Notice d'utilisation [U1.03.03].

Comment effectuer, calibrer et optimiser un calcul parallèle ?

- Notice d'utilisation [U2.08.06].

4 Soit la typologie du calcul renseigne sur cette information (on sait par exemple que c'est souvent le cas de la thermique), soit on l'obtient par ailleurs (par exemple, en faisant un test préliminaire avec MUMPS et en activant le mot-clé RESI_RELA+INFO=2).

5 Calculs non linéaires convexe, calcul prospectif ou calcul dont la qualité des solutions mécaniques d'intérêt est maîtrisée par ailleurs.

4 Préconisations sur les produits externes

Pour résoudre les nombreux systèmes linéaires qu'il produit, *Code_Aster* peut s'appuyer sur des produits externes. Ce papier documente l'utilisation de ces produits dans le cadre de *Code_Aster* en se référant à une utilisation et une installation classique. Les versions préconisées sont:

- Renuméroteurs/partitionneurs `METIS 4.0` et `SCOTCH 5.1.10`,
- Solveur direct `MUMPS 4.10.0` (la `4.9.2` est encore temporairement acceptée sauf calcul de déterminant, mais la 4.10.0 est le version de référence, maintenue et supportée),
- Solveur itératif `PETSc 3.2`.

Ci-joint le Copyright du produit MUMPS.

```
This version of MUMPS is provided to you free of charge. It is public
domain, based on public domain software developed during the Esprit IV
European project PARASOL (1996-1999) by CERFACS, ENSEEIHT-IRIT and RAL.
Since this first public domain version in 1999, the developments are
supported by the following institutions: CERFACS, CNRS, INPT(ENSEEIHT)-
IRIT, and INRIA.

Current development team includes Patrick Amestoy, Alfredo Buttari,
Abdou Guermouche, Jean-Yves L'Excellent, Bora Ucar.

Up-to-date copies of the MUMPS package can be obtained
from the Web pages:
http://mumps.enseeiht.fr/  or  http://graal.ens-lyon.fr/MUMPS


 THIS MATERIAL IS PROVIDED AS IS, WITH ABSOLUTELY NO WARRANTY
 EXPRESSED OR IMPLIED. ANY USE IS AT YOUR OWN RISK.


User documentation of any code that uses this software can
include this complete notice. You can acknowledge (using
references [1] and [2]) the contribution of this package
in any scientific publication dependent upon the use of the
package. You shall use reasonable endeavours to notify
the authors of the package of this publication.

 [1] P. R. Amestoy, I. S. Duff, J. Koster and  J.-Y. L'Excellent,
 A fully asynchronous multifrontal solver using distributed dynamic
 scheduling, SIAM Journal of Matrix Analysis and Applications,
 Vol 23, No 1, pp 15-41 (2001).

 [2] P. R. Amestoy and A. Guermouche and J.-Y. L'Excellent and
 S. Pralet, Hybrid scheduling for the parallel solution of linear
 systems. Parallel Computing Vol 32 (2), pp 136-156 (2006).
```

5 Liens avec le parallélisme

Toute simulation *Code_Aster* peut bénéficier des gains[6] de performance que procure le calcul sur plusieurs processeurs. Du moment qu'il effectue des calculs élémentaires/assemblages et/ou des résolutions de systèmes linéaires. Les gains peuvent être de deux ordres: sur le temps de calcul et sur l'espace RAM/disque. minimal requis.
Comme la plupart des codes généralistes en mécanique des structures, *Code_Aster* propose différentes stratégies pour s'adapter à l'étude et à la plate-forme de calcul:

1/ Parallélisme informatique

1a/ Lancement de *rampes de calculs indépendants* (calculs paramétriques, tests...); Parallélisme *via* un script shell; Gain en CPU. **Lancement standard *via* Astk**.

1b/ Distribution des *calculs élémentaires et des assemblages matriciels* et vectoriels dans les pré/post-traitements et dans les constructions de système linéaire. Parallélisme MPI; Gain en CPU et même gain en mémoire avec MUMPS ou PETSC + MATR_DISTRIBUE **Lancement standard *via* Astk.**

1c/ Distribution des *calculs d'algèbre linéaire* effectués par des *Blas* multithreadées. Gain en CPU. **Utilisation avancée**.

2/ Parallélisme numérique

2a/ *Solveur direct* *MULT_FRONT*; Parallélisme OpenMP; Gain en CPU. **Lancement standard *via* Astk.**

2b/ *Solveur direct* *MUMPS*; Parallélisme MPI; Gain en CPU et en mémoire. **Lancement standard *via* Astk.**

2c/ *Solveur litératif* *PETSC*; Paralélllisme MPI du solveur de Krylov (généralement pas du préconditionneur); Gain en CPU et en mémoire. **Lancement standard *via* Astk.**

Les parallélismes numériques 2a et, surtout, 2b, sont les plus plébiscités. Ils supportent une **utilisation «industrielle» et «grand public».** Ces parallélismes généralistes et fiables procurent des gains notables en CPU et en RAM. Leur paramétrisation est simple, leur mise en oeuvre facilitée *via* les menus de l'outil Astk. Ils peuvent facilement se cumuler, en amont et/ou en aval des systèmes linéaires, avec le parallélisme des calculs élémentaires (1b).
D'autres cumuls de parallélisme peuvent s'opérer. Leur mise en oeuvre n'est pas automatisée et ils s'adressent à des utilisateurs plutôt avancés: 1c+2b ou 2c, 2a+3a, 1a+2a/2b/2c/3a, 1a+1c+2b... Certains cumuls sont cependant proscrits car contre-productifs (1c+2a) ou non prévus fonctionnellement (2b/2c+3a). D'autres sont intrinsèques à la méthode (par ex. 1b+3a).

D'un point de vue pratique, l'utilisateur n'a plus à se soucier, pour une utilisation standard, de leur mise en œuvre en interaction fine avec les solveurs. En renseignant des menus dédiés d'Astk[7], on fixe le nombre de processeurs requis (en terme MPI et/ou OpenMP) avec éventuellement leur répartition par nœud.
Dans le déroulement du fichier de commande *Aster*, si plusieurs processus MPI sont activés (paramètre mpi_nbcpu pour 1b/2b/2c/3a), on distribue les mailles entre les processeurs (cf figure 2.1). Cette distribution se décline de différentes manières et elle est paramétrable dans l'opérateur AFFE_MODELE:
•CENTRALISE: Les mailles ne sont pas distribuées (comme en séquentiel). Chaque processeur connaît tout le maillage. Le parallélisme 1b n'est donc pas mis en œuvre. Ce mode d'utilisation est utile pour les tests de non-régression et pour certaines études où le parallélisme 1b rapporte peu voire est contre-productif (par ex. si on doit rassembler les données élémentaires pour nourrir un système

6 Ceux ci sont variables suivant les fonctionnalités sollicitées et leur paramétrage, le jeu de données et la plate-forme logicielle utilisée. Il s'agit essentiellement de gain en temps de calcul. Sauf dans le cas de MUMPS, de PETSC, où l'on peut aussi gagner en mémoire.

7 Menus Options+ncpus/mpi_nbcpus/mpi_nbnoeud.

linéaire non distribué et que les communications MPI requises sont trop lentes). 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 **(défaut)**/MAIL_DISPERSE/MAIL_CONTIGUE/SOUS-DOMAINE: Les mailles sont distribuées par groupe d'éléments finis du même type/suivant une distribution cyclique, par paquets de mailles contiguës ou suivant des sous-domaines (préalablement construits *via* l'opérateur DEFI_PARTITION). Les distributions par mailles sont les plus simples et les plus robustes à mettre en œuvre mais elles peuvent conduire à des déséquilibres de charge car elles ne tiennent pas compte des mailles particulières (mailles de peau, mailles spectatrices, zones non linéaires...). La distribution par sous-domaine est plus souple et elle peut s'avérer plus efficace en permettant d'adapter plus finement le flot de données à sa simulation.

Dans le cas d'opérateurs de calcul (MECA_STATIQUE, STAT_NON_LINE...), cette distribution des mailles qui sous-tend (potentiellement si on a choisi un mode distribué) le parallélisme «informatique» des calculs élémentaires va ensuite nourrir le parallélisme «numérique» des solveurs choisis par le mot-clé SOLVEUR.
Suivant le cas de figure, **solveur linéaire distribué** (MUMPS) **ou non** (MULT_FRONT, LDLT, GCPC, PETSC), on fournit les données, par morceaux ou après les avoir rassemblées, au solveur linéaire. Ce dernier souvent les redistribue en interne au grès de ses contingences. Mais dans le cas d'un solveur linéaire acceptant un flot de données d'entrée déjà distribué, les gains en temps et en mémoire sont patents.
Par ailleurs, on ne **rassemble les données** (ou on effectue les communications MPI idoines) que si le déroulement algorithmique l'impose: par exemple pour un calcul de maximum sur toutes les mailles du modèle ou pour un produit matrice-vecteur.
Si le **solveur linéaire parallèle utilise des threads** (OpenMP des scénarios 1c et 2a) ceux-ci s'exécutent indépendamment des éventuels aspects MPI. Dans le cas le plus usuel (2a), les threads se substituent ensuite aux processus MPI. Dans un cas de parallélisme hybride (1c+2b/c), ils démultiplient leur efficacité.

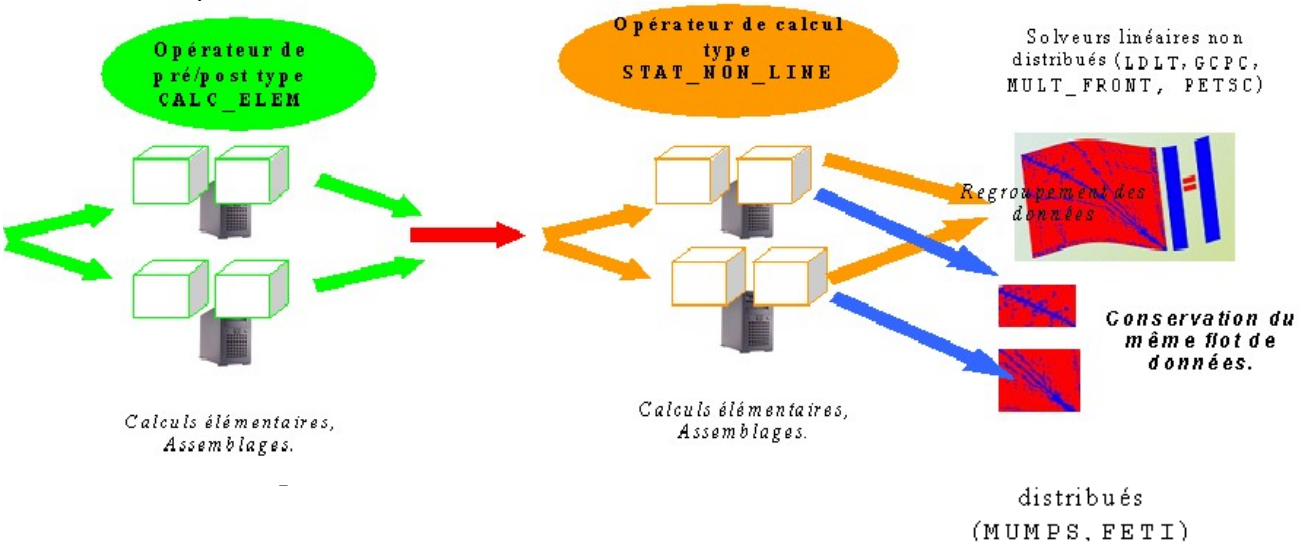

Figure 5.-1 : Lors d'un calcul paralléle, distribution des flots de données/traitements suivant les opérateurs (pré/ post-traitement ou résolution) et le type de solveur utilisé.

Dans le cas d'un **opérateur de pré/post-traitements** (type CALC_CHAMP), là encore, on ne rassemble les données (ou on effectue les communications MPI idoines) que si le déroulement algorithmique l'impose.
De toute façon, **tout calcul parallèle *Aster*** (sauf bien sûr le scénario 1a de calculs indépendants) **doit respecter le paradigme suivant**: les bases globales de chaque processeur sont identiques à la fin de chaque opérateur. Car on ne sait pas si l'opérateur qui suit dans le fichier de commande, a prévu un flot de données incomplet. Il faut donc organiser les communications idoines pour compléter les champs éventuellement incomplets.

Remarque :

Entre chaque commande, l'utilisateur peut même changer la répartition des mailles suivant les processeurs. Il lui suffit d'utiliser la commande `MODI_MODELE` *. Ce mécanisme reste licite lorsqu'on enchaîne différents calculs (mode* `POURSUITE` *). La règle étant bien sûr que cette nouvelle répartition perdure, pour le modèle concerné, jusqu'à l'éventuel prochain* `MODI_MODELE` *et que cette répartition doit rester compatible avec le paramétrage parallèle du calcul (nombre de noeuds/processeurs...).*

Pour plus d'informations on pourra consulter la notice d'utilisation du parallélisme [U2.08.06].

6 Indicateurs de performance d'un calcul

Lors d'une simulation *Code_Aster*, des affichages par défaut tracent dans le fichier message (.mess) certaines caractéristiques dimensionnantes du calcul. On retrouve notamment, pour chaque opérateur *Aster* :

- Les caractéristiques du calcul (nombre de nœuds, d'équations, de Lagrange, taille de la matrice...),
- Les mémoires JEVEUX plancher (pour passer en Out-Of-Core) et optimale (pour passer en In-Core),
- La mémoire requise par certains produits externes (par ex. MUMPS),
- Les temps CPU, système et «utilisateur» (elapsed),
- La ventilation des temps consommés suivant les étapes du calcul (calcul élémentaire, assemblage, résolution du système linéaire, déchargement sur disque).

Cette dernière description des temps consommés peut se décliner suivant différents niveaux de lecture (impression synthétique, détaillée et détaillée par incrément de calcul) *via* le paramètre MESURE_TEMPS/NIVE_DETAIL des commandes DEBUT/POURSUITE. En mode parallèle, on rajoute la valeur moyenne, sur tous les processeurs, des temps consommés ainsi que leur écart-type.

Pour plus d'informations on pourra consulter la notice d'utilisation [U1.03.03]: 'Indicateur de performance d'un calcul (temps/mémoire)'.

7 Informations complémentaires sur les mot-clés

La signification des différents paramètres du mot-clé SOLVEUR font l'objet de la documentation Utilisateur [U4.50.01]. Celle-ci doit être synthétique pour aider l'utilisateur dans un usage standard du code. Pour une utilisation plus avancée quelques informations complémentaires peuvent s'avérer fort utiles. Ce chapitre récapitule ces éléments.

7.1 Détection de singularité et mot-clés NPREC/ STOP_SINGULIER/ RESI_RELA

L'étape essentielle des solveurs directs (SOLVEUR=_F(METHODE='LDLT'/'MULT_FRONT'/'MUMPS')) en terme de consommation. Or elle peut achopper dans deux cas de figures: problème de construction de la factorisée (matrice structurellement ou numériquement singulière) et détection numérique d'une singularité (processus approximé plus sensible). Le comportement du code va dépendre du cas de figure, du paramétrage de NPREC/STOP_SINGULIER/RESI_RELA et du solveur utilisé. La combinatoire des cas de figures est décrite dans le tableau ci-joint.

Type de solveur/Type de problème	**Construction de la factorisée** *En cas de problème que se passe-t-il ?*	**Détection numérique de singularité(s).** *En cas de singularité que se passe-t-il ?*
LDLT/MULT_FRONT	Arrêt en ERREUR_FATALE	**Cas n°1:** On factorise une matrice dynamique dans un opérateur de dynamique (cf. R5.01.01 § 3.8, CALC_MODES ...): Arrêt en ERREUR_FATALE si il s'agit de la matrice de travail de l'algorithme. Émission d'une ALARME si il s'agit d'une étape du test de Sturm. Dans ces deux cas STOP_SINGULIER n'a aucune incidence sur le processus et NPREC doit être positif. **Cas n°2:** Si STOP_SINGULIER='OUI', Arrêt en ERREUR_FATALE dans la phase de post-traitement du solveur linéaire. **Cas n°3:** Si STOP_SINGULIER='NON', Émission d'une ALARME dans la phase de post-traitement du solveur linéaire. Solution potentiellement imprécise pas détectée par le solveur linéaire. Hormis un éventuel processus englobant (Newton...) on a aucun garde-fou numérique pour garantir la qualité de la solution. **Cas n°4:** Si STOP_SINGULIER='DECOUPE', □ Lancement du processus de découpage du pas de temps. On reconstruit un nouveau problème pour un autre incrément de temps/chargement.
MUMPS	Arrêt en ERREUR_FATALE	**Cas n°1/4:** Si NPREC> 0,même comportement que LDLT / MULT_FRONT (on retrouve les cas n °1 à 4) . **Cas n°5:** Si NPREC<0 et RESI_RELA>0,

Type de solveur/Type de problème	Construction de la factorisée *En cas de problème que se passe-t-il ?*	Détection numérique de singularité(s). *En cas de singularité que se passe-t-il ?*
		Détection de singularité désactivée mais on mesure la qualité de la solution dans le solveur linéaire. Si le système est singulier, son conditionnement sera très élevé et la qualité de résolution sera très mauvaise. Si cette qualité de résolution est supérieure à la valeur paramétrée dans RESI_RELA: arrêt en ERREUR_FATALE. **Cas n°6:** Si NPREC<0 et RESI_RELA<0, Détection de singularité et de mesure de la qualité de la solution toutes les deux désactivées. Hormis un éventuel processus englobant (Newton...) on a aucun garde-fou numérique pour garantir la qualité de la solution.

Tableau 7.1-1. Comportement du code, en fonction du paramétrage, lorsque la factorisation numérique détecte des problèmes (mauvaise mise en données, instabilités numériques, fort conditionnement...).

On compare quelques détails numériques des deux types de critères de détection (LDLT/MULT_FRONT *versus* MUMPS) de singularité dans le tableau ci-dessous:

Caractéristiques/Type de solveur	LDLT/MULT_FRONT	MUMPS
Critère	Local à chaque degré de liberté (valeur relative)	Global pour tous les degrés de liberté
Terme testé	Valeur absolue du terme diagonal de chaque ligne	Norme infinie de la ligne/colonne du la ligne correspondant au pivot
Détection du numéro de ligne (ISINGU dans les messages)	Toujours	Oui sauf lors de problèmes avec la construction de la factorisée
Fourniture du nombre de décimales perdues	Oui, sauf lors de problèmes avec la construction de la factorisée	Non
Désactivable	Non	Oui

Tableau 7.1-2. Différences dans les processus de détection de singularité suivant les solveurs.

Cependant, au delà des différences de mises en œuvre et des messages d'erreurs (tel solveur pointe un degré de liberté, tel autre solveur un autre degré de liberté), les deux classes de solveurs directs, LDLT/MULT_FRONT et MUMPS, concluent généralement au même type de diagnostics en cas de problèmes[8].
Elles pointent une mise en données déficiente: blocages redondants ou, au contraire, absents; relations linéaires surabondantes dues au contact-frottement; données numériques très hétérogènes (terme de pénalisation trop grands) ou illicites (module de Young négatif...).

Remarques:

Au pire, il faut ajuster la valeur de NPREC (augmenter ou diminuer de 1) pour conduire au même constat. En général, les singularités sont si flagrantes, que le paramétrage par défaut convient tout à fait.

8 Cf. Cas-tests erreu03a et erreu04a.

Contrairement aux deux autres solveurs, MUMPS ne précise pas le nombre de décimales perdues, par contre l'activation de son critère de qualité (par exemple RESI_RELA=1.10⁻⁶) constitue un ultime garde-fou efficace contre ce type d'anicroche.
Il est normal et assumé que le degré de liberté détecté soit parfois différent lorsqu'on change un paramètre numérique (solveur, renuméroteur, prétraitements...) ou informatique (parallélisme...). Tout dépend de l'ordre dans lequel chaque processeur traite les inconnues dont il a la charge et des techniques de pivotage/équilibrage éventuellement mises en œuvre. En général, même différents, les résultats concourent au même diagnostic: revoir les blocages de sa mise en données.
Pour obtenir le conditionnement matriciel [9] de son opérateur «brut» (c'est-à-dire sans les éventuels prétraitements opérés par le solveur linéaire), on peut utiliser la combinaison MUMPS+PRETRAITEMENTS='NON'+INFO=2+NPREC<0+RESI_RELA>0 . Une valeur très importante (> 10 [12]) trahit alors la présence d'au moins une singularité. Par contre, si le surcoût calcul de la détection de singularité est indolore, celui de l'estimation du conditionnement et de la qualité de la solution l'est moins (jusqu'à 40% en temps elapsed; cf. § 7.2.5).

Pour être plus précis, on a 9 cas de figures distincts listés dans le tableau ci-dessous. Ils sont basés sur la nullité exacte ou approchée[10] des termes « pivots » (cf. [R6.02.03] §2.3) sélectionnés par la phase de factorisation numérique du solveur direct considéré. Le premier cas de figure apparaît lors de la factorisation numérique proprement dite (matrice numériquement ou structurellement singulière). Le second cas provient de la phase de post-traitement activée à l'issue de cette factorisation numérique.

Prioritairement l'utilisateur doit suivre les conseils prodigués par le message d'alarme (listé dans le tableau ci-dessous). Si cela ne suffit vraiment pas, l'utilisateur avancé peut essayer de jouer sur les paramètres numériques du solveur (renuméroteur...), voire sur le critère NPREC lorsqu'il s'agit d'un pivot presque nul.

Type de problème	Informations disponibles	Conseils
Matrice ne s'appuyant pas sur un maillage. Pivot nul	Numéro de ligne (si MF/LDLT).	Mise en données (conditions limites, caractéristiques matériaux...). Essayer MUMPS (si MF/LDLT).
Matrice ne s'appuyant pas sur un maillage. Pivot presque nul.	Numéro de ligne, Nombre de décimales perdues (si MF/LDLT).	Idem.
Le pivot est un degré de liberté physique hors X-FEM. Pivot nul.	Numéro de ligne/noeud/composante (si MF/LDLT).	Mise en données (conditions limites, caractéristiques matériaux...). Essayer MUMPS (si MF/LDLT).
Le pivot est un degré de liberté physique hors X-FEM. Pivot presque nul.	Numéro de ligne/noeud/composante, Nombre de décimales perdues (si MF/LDLT)	Mode de corps rigide mal bloqué (défaut de blocage). Si le calcul comporte du contact il ne faut pas que la structure ne « tienne » que par les relations de contact.
Le pivot est un degré de liberté physique X-FEM. Pivot presque nul.	Numéro de ligne/noeud/composante (si MF/LDLT).	La level-set (fissure) passe très prés du noeud considéré (augmentez NPREC jusqu'à 10).
Le pivot est un Lagrange lié à une relation linéaire entre degrés de liberté. Pivot nul.	Numéro de ligne (si MF/LDLT).	Relations linéaires entre degrés de liberté surabondantes (LIAISON, contact...). Mise en données (conditions

9 *Au sens résolution de système linéaire creux, cf. papiers de M.Arioli/J.Demmel/I.Duff.*
10 Contrôlé par NPREC (cf. [U4.50.01] §3.2).

Type de problème	Informations disponibles	Conseils
		limites, caractéristiques matériaux...). Essayer MUMPS (si MF/LDLT).
Le pivot est un Lagrange lié à une relation linéaire entre degrés de liberté. Pivot presque nul	Numéro de ligne, Nombre de décimales perdues (si MF/LDLT).	Relations linéaires entre degrés de liberté surabondantes (LIAISON, contact...).
Le pivot est un degré de liberté de Lagrange lié à un blocage d'un degré de liberté. Pivot nul	Numéro de ligne (si MF/LDLT). Blocage concerné.	Blocage surabondant. Mise en données (conditions limites, caractéristiques matériaux...). Essayer MUMPS (si MF/LDLT).
Le pivot est un degré de liberté de Lagrange lié à un blocage d'un degré de liberté. Pivot presque nul	Numéro de ligne. Blocage concerné. Nombre de décimales perdues (si MF/LDLT).	Blocage surabondant.

Tableau 7.1-3. Différents cas de détection de singularité et conseils associés.

7.2 Solveur MUMPS (METHODE='MUMPS')

7.2.1 Généralités

Le **solveur MUMPS** actuellement développé par **CNRS/INPT-IRIT/INRIA/CERFACS** est un **solveur direct de type multifrontal, parallélisé** (en MPI) et robuste, car il permet de pivoter les lignes et colonnes de la matrice lors de la factorisation numérique.
MUMPS fournit une estimation de la qualité de la solution $\mathbf{u}$ (cf. mot-clé RESI_RELA) du problème matriciel $\mathbf{K}\,\mathbf{u}=\mathbf{f}$ *via* les notions d'*erreur directe relative* ('relative forward error') et d'erreur *inverse* ('backward error'). Cette 'backward error', $\eta(\mathbf{K},\mathbf{f})$, mesure le comportement de l'algorithme de résolution (quand tout va bien, ce réel est proche de la précision machine, soit 10^{-15} en double précision). MUMPS calcule aussi une estimation du conditionnement de la matrice, $\kappa(\mathbf{K})$, qui traduit le bon comportement du problème à résoudre (réel compris entre 10^4 pour un problème bien conditionné jusqu'à 10^{20} pour un très mal conditionné). Le produit des deux est un majorant de l'erreur relative sur la solution ('relative forward error'):

$$\frac{\|\delta u\|}{\|u\|} < C^{st}\cdot\kappa(\mathbf{K})\cdot\eta(\mathbf{K},\mathbf{f})$$

En précisant une valeur strictement positive au mot-clé RESI_RELA (par ex. 10^{-6}), l'utilisateur indique qu'il souhaite tester la validité de la solution de chaque système linéaire résolu par MUMPS à l'aune de cette valeur. Si le produit $\kappa(\mathbf{K})\cdot\eta(\mathbf{K},\mathbf{f})$ est supérieur à RESI_RELA le code s'arrête en ERREUR_FATALE, en précisant la nature du problème et les valeurs incriminées. Avec l'affichage INFO=2, on détaille chacun des termes du produit: $\eta(\mathbf{K},\mathbf{f})$ et $\kappa(\mathbf{K})$.

Pour poursuivre le calcul, on peut alors:

- **Augmenter la tolérance de RESI_RELA**. 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 (cf. remarque suivante sur le conditionnement et §3.4).
- Si **c'est la 'backward error'** qui est trop importante: il est conseillé de modifier l'algorithme de résolution. C'est-à-dire, dans notre cas, de jouer sur les paramètres de lancement de MUMPS (TYPE_RESOL, PRETRAITEMENTS...).
- Si **c'est le conditionnement de l'opérateur** qui est en cause, il est conseillé d'équilibrer les termes de la matrice, en dehors de MUMPS ou *via* MUMPS (PRETRAITEMENTS='OUI'), ou de changer la formulation du problème.

Remarque :

•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.
***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.*

7.2.2 Périmètre d'utilisation

C'est un solveur linéaire universel. Il est déployé pour toutes les fonctionnalités de *Code_Aster.*
D'autre part, l'usage de solveur souffre de petites limitations peu fréquentes et que l'on peut contourner aisément le cas échéant.

- En mode POURSUITE on ne sauvegarde sur fichier que les objets FORTRAN77 d'*Aster* et donc pas les occurrences de produits externes (MUMPS, PETSc). Donc attention à l'usage des commandes éclatées dans ce cadre (NUME_DDL/FACTORISER/RESOUDRE). Avec MUMPS, il n'est pas possible de FACTORISER ou de RESOUDRE un système linéaire construit lors d'un run Aster précédent (avec NUME_DDL).
- De même on limite le nombre d'occurrence simultanées de MUMPS (et PETSc) à NMXINS=5. Lors de la construction de son problème *via* des commandes éclatées, l'utilisateur doit veiller à ne pas dépasser ce chiffre.

7.2.3 Paramètre RENUM

Ce mot clé permet de contrôler l'outil utilisé pour renuméroter le système linéaire[11]. L'utilisateur *Aster* peut choisir différents outils répartis en deux familles: les outils « frustres » dédiés à un usage et fournis avec MUMPS ('AMD', 'AMF', 'QAMD', 'PORD'), et, les bibliothèques plus « riches » et plus « sophistiquées » qu'il faut installer séparément ('METIS', 'SCOTCH').

Le choix du renuméroteur a une grande importance sur les consommations mémoire et temps du solveur linéaire. Si on cherche a optimiser/régler les paramètres numériques liés au solveur linéaire, ce paramètre doit être un des premiers à essayer.
En particulier, lors de calculs modaux parallèles, on a parfois observé des speed-ups décevants du fait d'un choix inapproprié de renuméroteur. En effet, le produit MUMPS décompose ses calculs en trois étapes (cf. [R6,02,03] §1.6): phase d'analyse, de factorisation numérique et de descente-remontée. Seules les deux dernières étapes sont parallélisées, le première est purement séquentielle. Ce n'est pas en général préjudiciable car le temps passé dans la première étape est négligeable devant celui de la seconde.
Cependant, on a observé dans certains cas (calcul modal cf. cas-test perf013), une grande sensibilité de ce temps de calcul en fonction du renuméroteur (de 1 à 30 par exemple). Du coup cette étape devient prédominante par rapport à la seconde et elle dimensionne le temps total consommé dans MUMPS. Et comme cette étape est séquentielle, les gains en temps qu'apportent habituellement le parallélisme deviennent alors négligeables. Il ne reste plus que des gains en mémoire (car MUMPS continue néanmoins à distribuer ses données entre les processeurs).
Dans ce cas là on a constaté que le choix d'un renuméroteur « sophistiqué » était contre-performant. **Il vaut mieux imposer à MUMPS un simple 'AMF' ou 'QAMD', plutôt que 'METIS' (souvent pris automatiquement en mode 'AUTO').**

7.2.4 Paramètre ELIM_LAGR2

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 Lagranges (AFFE_CHAR_MECA/THER…) a été modifiées en introduisant des doubles Lagranges. 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 ce mot-clé à 'OUI', **on ne tient plus compte que d'un Lagrange, l'autre étant spectateur[12]**. D'où une matrice de travail $\mathbf{K}_1$ simplement dualisée

11 Afin, notamment, de limiter le phénomène de remplissage de la factorisée, cf. [R6.02.03] §1.7.

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

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

Pour les **problèmes comportant de nombreux Lagranges (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_LAGR2='NON' peut être alors très intéressant (gain de 40% en CPU sur le cas-test mac3c01).
On **débranche** aussi temporairement ce paramètre lorsqu'on souhaite calculer **le déterminant de la matrice**, car sinon sa valeur est faussée par ces modifications des termes de blocage. L'utilisateur est averti de cette modification automatique de paramétrage par un message dédié (visible en INFO=2 uniquement).

7.2.5 Paramètre RESI_RELA

Valeur par défaut=-1.d0 en non linéaire et en modal, 1.d-6 en linéaire.
Ce paramètre est désactivé par une valeur négative.

En précisant une valeur strictement positive à ce mot-clé (par ex. 10^{-6}), l'utilisateur indique qu'il souhaite tester la validité de la solution de chaque système linéaire résolu par MUMPS à l'aune de cette valeur.
Cette démarche prudente est conseillée lorsque la solution n'est pas elle même corrigée par un autre processus algorithmique (algorithme de Newton, détection de singularité…) bref dans les opérateurs linéaires THER_LINEAIRE et MECA_STATIQUE. En non linéaire, le critère de détection de singularité et la correction de Newton sont des garde-fous suffisants. On peut donc débrancher ce processus de contrôle (c'est ce qui est fait par défaut *via* la valeur -1). En modal, cette détection de singularité est un outil algorithmique pour capturer les modes propres. Cette détection fait l'objet d'un paramétrage dédié propre à chaque méthode.
Si l'erreur relative sur la solution estimée par MUMPS est supérieure à resi le code s'arrête en ERREUR_FATALE, en précisant la nature du problème et les valeurs incriminées.
L'activation de ce mot-clé initie aussi un processus de raffinement itératif dont l'objectif est d'améliorer la solution obtenue. Ce post-traitement bénéficie d'un paramétrage particulier (mot-clé POSTTRAITEMENTS). C'est la solution résultant de ce processus d'amélioration itérative qui est testée par RESI_RELA.

Remarque :
• Ce processus de contrôle implique l'estimation du conditionnement matriciel et quelques descentes-remontées du post-traitement de raffinement itératif. Il peut donc être assez coûteux, notamment en OOC, du fait des I/O RAM/disque lors des descentes-remontées (jusqu'à 40%). Lorsque suffisamment de garde-fous sont mises en œuvre on peut le débrancher en initialisant resi à une valeur négative.

7.2.6 Paramètres pour optimiser la gestion mémoire (MUMPS et/ou JEVEUX)

En général une grande partie des temps calcul et des pics mémoires RAM d'une simulation *Code_Aster* sont imputables aux solveurs linéaires. Le solveur linéaire MUMPS n'échappe pas à la règle mais la richesse de son paramétrage interne et son couplage fin avec *Code_Aster* ménagent

12 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 $\mathbf{K}_2$ à $\mathbf{K}_1$, plutôt qu'au scénario optimal $\mathbf{K}_0$.

une certaine souplesse à l'utilisateur. Notamment en ce qui concerne la gestion de la consommation en mémoire RAM.

En fait, **lors d'un pic mémoire** survenant pour une résolution de système linéaire *via* MUMPS, la mémoire RAM peut se décomposer en 5 parties:

- Les objets JEVEUX hormis la matrice,
- Les objets JEVEUX associés à la matrice (généralement les SD MATR_ASSE et NUME_DDL),
- Les objets MUMPS permettant de stocker la matrice,
- Les objets MUMPS permettant de stocker la factorisée,
- Les objets MUMPS auxiliaires (pointeurs, vecteurs, buffers de communication...).

Grossièrement, la partie 4 est la plus encombrante. En particulier, elle est beaucoup plus grosse que les parties 2 et 3 (facteur d'au moins 30 dû au phénomène de remplissage cf. [R6.02.03]). Ces dernières sont équivalentes en taille mais l'une s'exerce dans l'espace dévolu à JEVEUX, tandis que l'autre, vit dans l'espace complémentaire alloué par le gestionnaire de tâche au job. Quant aux deux autres parties, 1 et 5, elles jouent souvent un rôle marginal[13].

Pour diminuer ces consommations mémoire, l'**utilisateur *Aster* dispose de plusieurs bras de levier** (la plupart du temps cumulables):

- **Le calcul parallèle centralisé** sur n coeurs (partie 4 divisée par n) ou distribué (parties 3 et 4 divisées par n).
- **Le calcul parallèle distribué** + **MATR_DISTRIBUEE** (périmètre d'utilisation limité): parties 3 et 4 divisées par n , partie 2 divisée par un peu moins de n .
- L'activation de l'**OOC de MUMPS** (mot-clé GESTION_MEMOIRE='OUT-OF-CORE'): la partie 4 diminue des 2/3.
  Sachant qu'avant chaque appel au cycle 'analyse+factorisation numérique' de MUMPS, on décharge sur disque systématiquement les plus gros objets JEVEUX de la partie 2.

Conseils:
Au vu des éléments précédent, une tactique évidente consiste à passer le calcul en mode parallèle distribué (valeur par défaut) plutôt qu'en séquentiel. Le mode parallèle centralisé n'apportant rien de ce point de vue[14], il n'est pas à envisager. Si toutefois, on reste tributaire d'un mode particulier, voici les conseils associés à chacun :

- **En mode séquentiel:** les principaux gains viendront de l'activation de GESTION_MEMOIRE='OUT_OF_CORE' (sur des problèmes de taille raisonnable).
- **En mode parallèle distribué:** passé une dizaine de processeurs, l'OUT_OF_CORE ne procure plus beaucoup de gain. MATR_DISTRIBUEE peut alors aider dans certaines situations.

Si ces stratégies n'apportent pas suffisamment de gains, on peut aussi essayer en non linéaire (au prix d'éventuels pertes de précision et/ou de temps) de relaxer la résolution du système linéaire (FILTRAGE_MATRICE, MIXER_PRECISION) voire celles du processus englobant (matrice tangente élastique, réduction de l'espace de projection en modal...). Si la matrice est bien conditionnée et si on n'a pas besoin de détecter les singularités du problème (donc pas calcul modal, flambement...), on peut aussi tenter un solveur itératif (GCPC/PETSC+LDLT_SP).

Solution	Gain en mémoire RAM	Surcoût en temps	Perte de précision
Parallélisme	+++	Au contraire, gain en temps	Aucune
GESTION_MEMOIRE= 'OUT_OF_CORE'	++	Faible sauf si nombreuses descente-remontées	Aucune
MATR_DISTRIBUEE	+	Aucun	Aucune

13 Mais pas toujours. Ainsi la partie 1 peut finir par être coûteuse lorsqu'on calcule beaucoup de pas de temps, lorsqu'on charge des champs projetés ou lorsqu'une loi de comportement manipule beaucoup de variables internes. De même, la partie 5 peut devenir non négligeable lors d'un calcul parallèle sur un grand nombre de processeur.

14 Il est principalement utilisé pour tester et valider de nouveaux développements ainsi que des études parallèles.

(périmètre limité)			
Relaxation des résolutions FILTRAGE_MATRICE, MIXER_PRECISION (périmètre limité)	++	Variable	Variable. Possibilité de non convergence.
Changer de solveur: GCPC/PETSC+LDLT_SP (périmètre limité)	+++	Variable	Variable. Possibilité de non convergence.

Tableau 7.2-1. Synoptique des différentes solutions pour optimiser la mémoire lors d'un calcul avec MUMPS.

Mot-clé GESTION_MEMOIRE='OUT_OF_CORE'

Pour activer ou désactiver les facultés OOC de MUMPS qui va alors décharger entièrement sur disque la partie réelle des blocs de factorisée gérés par chaque processeur. Cette fonctionnalité est bien sûr cumulable avec le parallélisme, d'où une plus grande variété de fonctionnement pour s'adapter aux contingences d'exécution. L'OOC, tout comme le parallélisme, contribue à réduire la mémoire RAM requise par processeur. Mais bien sûr (un peu) au détriment du temps CPU: prix à payer pour les I/O pour l'un, pour les communications MPI pour l'autre.

Attention: Lors d'un calcul parallèle, si le nombre de processeurs est important, la taille des objets MUMPS déchargeables sur disque devient faible. Le passage en OOC peut alors s'avérer contre-productif (gain faible en RAM et surcoût en temps) par rapport au mode IC paramétré par défaut.
Ce phénomène se produit d'autant plus précocement que la taille du problème est faible et il est d'autant plus sensible qu'on effectue beaucoup de descentes-remontées[15] dans le solveur.
Grosso-modo[16], en dessous de 50.10^3 degrés de liberté par processeurs et si on dispose de suffisamment de RAM (3 ou 4 Go) par processeur, on peut sans doute basculer en IC.

Remarques :

• Pour l'instant, lors d'une exécution MUMPS en OOC, seuls les vecteurs de réels contenant la factorisée sont (entièrement) déchargés sur disque. Les vecteurs d'entiers accompagnant cette structure de données (de taille tout aussi importante) ne bénéficient pas encore de ce mécanisme. D'autre part, ce déchargement ne s'opère qu'après la phase d'analyse de MUMPS. Bref, sur de très gros cas (plusieurs millions de degrés de liberté), même avec cet OOC, des contingences mémoires peuvent empêcher le calcul. On sort en principe avec une ERREUR_FATALE documentée.

*• Dans MUMPS, de manière à optimiser l'occupation mémoire, l'essentiel des entiers est codé en INTEGER*4. Seuls les entiers correspondant à une adresse mémoire sont transcrits en INTEGER*8. Cela permet d'adresser des problèmes de plus grande taille, sur des architectures 64 bits. cette cohabitation entiers courts/longs pour optimiser la place mémoire a été étendue à certains gros objets JEVEUX.*

• Lorsque Code_Aster a fini d'assembler la matrice de travail, avant de passer « le relais » à MUMPS, il décharge sur disque les plus gros objets JEVEUX liés à la résolution du système linéaire (SMDI/HC, DEEQ, NUEAQ, VALM...). Et ce afin de laisser le plus de place RAM possible à MUMPS. En mode GESTION_MEMOIRE='AUTO', si cette place mémoire n'est pas suffisante pour que MUMPS fonctionne en IC, on complète cette libération partielle d'objets JEVEUX par une libération générale de tous les objets libérables (c'est-à-dire non ouvert en lecture/écriture). Cette opération peut procurer beaucoup de gains lorsqu'on « traîne » en mémoire beaucoup d'objets JEVEUX périphériques (projection de champs, long transitoire...). Par contre, ce déchargement massif peut faire perdre du temps. En particulier en mode parallèle du fait d'engorgements des accès coeurs/RAM.

Mot-clé MATR_DISTRIBUEE

Ce paramètre est utilisable dans les opérateurs MECA_STATIQUE, STAT_NON_LINE, DYNA_NON_LINE, THER_LINEAIRE et THER_NON_LINE avec AFFE_CHAR_MECA ou

15 Par exemple, dans un STAT_NON_LINE avec beaucoup d'itérations de Newton et/ou des itérations de raffinement itératif.

16 Cela dépend beaucoup de l'étude.

AFFE_CHAR_CINE. Il n'est actif qu'en parallèle distribué (AFFE_MODELE, PARTITION, PARALLELISME DISTRIBUE). Cette fonctionnalité est bien sûr cumulable avec GESTION_MEMOIRE='OUT_OF_CORE', d'où une plus grande variété de fonctionnement pour s'adapter aux contingences d'exécution.

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 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 7.2.2 illustrent ce gain en parallèle sur deux études: une Pompe RIS et la cuve Epicure.

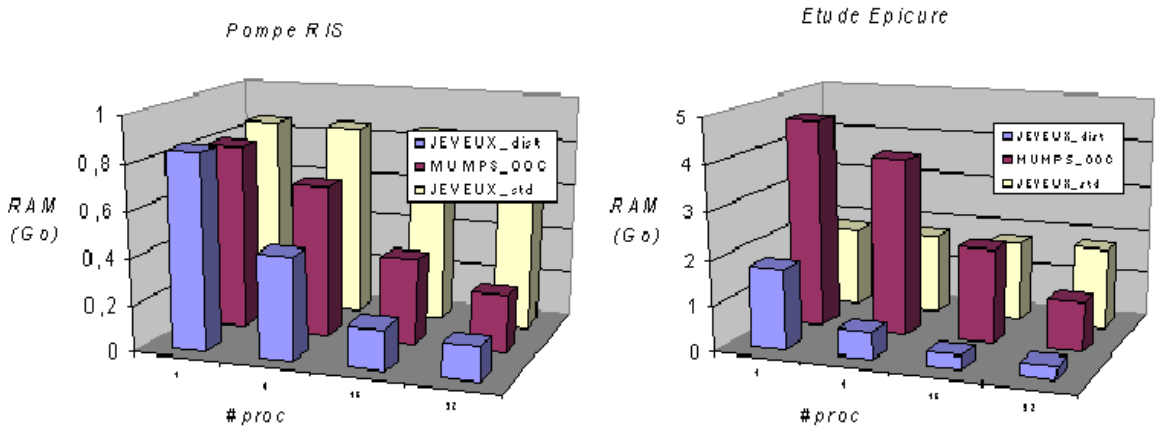

Figure 7.2-2: Évolution des consommations RAM (en Go) en fonction du nombre de processeurs, de Code_Aster (JEVEUX standard MATR_DISTRIBUE='NON' et distribué, resp. 'OUI') et de MUMPS OOC. Résultats effectués sur une Pompe RIS et sur la cuve de l'étude Epicure.

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.