Notice d'utilisation du parallélisme

Résumé :

Toute simulation *Code_Aster* peut bénéficier des gains de performance que procure le parallélisme. Les gains peuvent être de deux ordres: sur le temps de calcul et sur l'espace RAM/disque requis (par cœur alloué).

Code_Aster propose différentes stratégies parallèles pour s'adapter à l'étude et à la plate-forme de calcul. Certaines sont plutôt axées sur des aspects informatiques (distribution de calculs complets ou de calculs modaux indépendants, construction de systèmes linéaires, opérations basiques d'algèbre linéaire), d'autres sont plus algorithmiques (solveurs linéaires HPC MUMPS et PETSc).

Ce document décrit brièvement l'organisation du parallélisme dans le code. Puis il rappelle quelques fondamentaux afin d'aider l'utilisateur à tirer parti de ces stratégies parallèles. Puis on détaille leurs mises en oeuvre, leurs périmètres d'utilisation et leurs gains potentiels. Les chaînages/cumuls de différentes stratégies (souvent naturels et paramétrés par défaut) sont aussi abordés.

L'utilisateur pressé peut d'emblée se reporter au chapitre 2 (« Le parallélisme en quelques clics ! »). Il résume le mode opératoire pour mettre en oeuvre la stratégie parallèle préconisée par défaut.

Remarque:

Pour utiliser Code_Aster en parallèle, (au moins) trois cas de figures peuvent se présenter:

- *On a accès à la machine centralisée Aster et on souhaite utiliser l'interface d'accès Astk,*
- *On effectue des calculs sur un cluster ou sur une machine multi-coeurs avec Astk,*
- *Idem que le cas précédent mais sans Astk.*

Table des Matières

1 Le parallélisme en quelques clics !....3
1.1 Pourquoi ?....3
1.2 Comment ?....3
1.3 En pratique via Astk....4
2 Généralités....7
2.1 Parallélismes informatiques....7
2.2 Parallélismes numériques....7
2.3 Parallélisation des systèmes linéaires....8
2.4 Distribution de calculs modaux....10
3 Quelques conseils préalables....11
3.1 Préambule....11
3.2 Quelques chiffres empiriques....12
3.3 Calculs indépendants....12
3.4 Gain en mémoire RAM....12
3.5 Gain en temps....12
4 Parallélismes informatiques....14
4.1 Rampes de calculs indépendants....14
4.1.1 Descriptif....14
4.1.2 Mise en oeuvre....14
4.2 Calculs élémentaires et assemblages....15
4.2.1 Descriptif....15
4.2.2 Mise en oeuvre....15
4.2.3 Structures de données distribuées....16
4.3 Distribution des calculs d'algèbre linéaire basiques....17
4.3.1 Descriptif....17
4.3.2 Mise en oeuvre....17
4.4 Calculs modaux d' INFO_MODE/CALC_MODES....18
4.4.1 Descriptif....18
4.4.2 Mise en oeuvre....18
5 Parallélismes numériques....20
5.1 Solveur direct MULT_FRONT....20
5.1.1 Descriptif....20
5.1.2 Mise en oeuvre....20
5.2 Package MUMPS....21
5.2.1 Descriptif....21
5.2.2 Mise en œuvre....21
5.3 Solveur itératif PETSC....23
5.3.1 Descriptif....23

5.3.2 Mise en œuvre....23

1 Le parallélisme en quelques clics !

1.1 Pourquoi ?

Souvent une simulation *Code_Aster* peut **bénéficier de gains importants de performance** en distribuant ses calculs sur plusieurs cœurs d'un PC ou sur un ou plusieurs nœuds d'une machine centralisée.
On peut **gagner en temps** (avec le parallélisme MPI et avec le parallélisme OpenMP) comme en **mémoire** (seulement *via* MPI). Ces gains sont variables suivant les fonctionnalités sollicitées, leurs paramétrages, le jeu de données et la plate-forme logicielle utilisée : cf. figure 1.1.

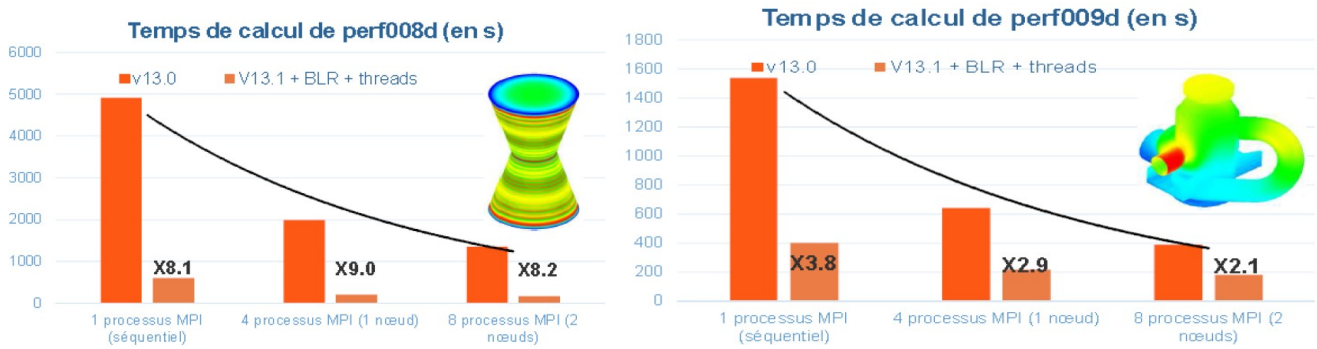

Figure 1.1._ Exemple de gains en temps procurés par le parallélisme MPI de Code_Aster v13.0, et avec celui hybride MPI+OpenMP (+ compressions low-rank cf. [U4.50.01]) de Code_Aster v13.1. Comparaisons effectuées sur les cas-tests de performance perf008/9d et sur la machine centralisée Aster5.

1.2 Comment ?

Dans *Code_Aster,* par défaut, le calcul est séquentiel. Mais on peut activer **différentes stratégies de parallélisation**. Celles-ci dépendent de l'étape de calcul considérée et du paramétrage choisi. Elles sont souvent chaînables ou cumulatives. On peut ainsi initier des schémas parallèles comportant jusqu'à 3 niveaux de parallélisme imbriqués.
On a trois grandes classes de problèmes parallélisables, la seconde étant la plus courante:

- soit la simulation peut s'organiser en plusieurs **sous-calculs indépendants** ,
- soit ce n'est pas la cas mais :
  - celle-ci reste **dominée par des calculs linéaires ou non linéaires** (opérateurs `STAT/DYNA/THER_NON_LINE` , `MECA_STATIQUE` ...),
  - celle-ci reste **dominée par des calculs modaux** divisibles en sous- bandes fréquentielles (`INFO_MODE/CALC_MODES+'BANDE'`).

Pour avoir une **estimation du temps passé par un opérateur** et donc des étapes prédominantes d'un calcul, on peut activer le mot-clé `MESURE_TEMPS` des commandes `DEBUT/POURSUITE`[U1.03.03] sur une étude type (éventuellement raccourcie ou expurgée).
Dans tous les cas, on conseillera de **diviser les plus gros calculs en différentes étapes** afin de séparer celles purement **calculatoires**[1], de celles concernant des **affichages**, des **post-traitements** et des **manipulations de champs**[2].

Dans le premier cas de figure, on se référera à l'usage dédié de l'outil Astk (cf. §4.1 et [U2.08.07]).

1 Eventuellement de différents types (catégorie n°2 ou n°3 citée précédemment) et qui gagneront à être effectuées en parallèle.
2 Qui seront souvent plus rapides en séquentiel du fait des risques d'engorgements lors des accès mémoire.

Dans le second cas de figure, il faut commencer par repérer les résolutions de systèmes linéaires dans le fichier de commande (mot-clé SOLVEUR) et modifier leurs paramétrages afin d'utiliser un solveur linéaire HPC [U4.50.01]. Pour ce faire, on spécifie la valeur 'MUMPS' ou 'PETSC' au mot-clé METHODE.
On distribue ensuite, sur les processus MPI, les étapes de construction et celles de résolution de systèmes linéaires (cf. §4.2/5.2/5.3). Cela permet de baisser le niveau de mémoire requis et d'accélérer la simulation.
On peut rajouter un deuxième niveau de parallélisme en utilisant, pour chaque processus MPI, plusieurs threads OpenMP (cf. §4.3). Ce second niveau n'accélére, par contre, qu'une partie de la résolution des systèmes linéaires et il ne permet pas de baisser le pic en mémoire RAM.

Dans le troisième cas de figure, on distribue les calculs modaux sur différents blocs de processeurs (cf. §4.4), puis, en utilisant le solveur linéaire MUMPS, on peut rajouter un deuxième niveau MPI de parallélisation au sein de chacun de ses sous-blocs (cf. cas de figure précédent). On peut éventuellement rajouter un troisième niveau en activant des threads OpenMP au sein de chaque processus MPI de résolution (cf. §4.3). Mais ce scénario est rarement productif. Il vaut mieux réserver ces cœurs à une plus large distribution en sous-bandes.
Notons que l'efficacité de cette stratégie requiert des bandes fréquentielles assez bien équilibrées. Pour calibrer ces bandes, il est conseillé d'utiliser préalablement l'opérateur INFO_MODE[U4.52.01]. Lui aussi bénéficie d'un parallélisme MPI à deux niveaux très performant.

1.3 En pratique via Astk

Pour la mise en œuvre effectives des cas de figure n°2 et n°3, il faut pré-sélectionner une version parallèle de *Code_Aster* (notée *** _mpi), puis préciser le nombre de coeurs retenus (menu Options d'A*stk*) *via* les champs suivants:

- (facultatif) **ncpus=k**, nombre de threads alloués en OpenMP; généralement utilisé en complément de MPI; valeur paramétrée par défaut si on ne rempli par le champ et qu'il reste vide.
- **mpi_nbcpu=m**, nombre de processus MPI alloués.
- **mpi_nbnoeud=p**, nombre de nœuds sur lesquels vont être distribués ces m x k calculs parallèles.

On conseille en générale de **ne pas allouer tous les cœurs d'un nœud en MPI seul** . Cela peut avoir pour effet de **ralentir la simulation** car, même si une partie des calculs s'en trouve accélérée du fait de sa distribution sur plus de cœurs, comme ceux-ci partagent certaines ressources mémoire, les accès aux données sont, eux, ralentis.
Pour utiliser plus efficacement et à 100% toutes les ressources allouées on conseille plutôt de **panacher et d'équilibrer les parallélismes MPI et OpenMP** (parallélisme hybride à 2 niveaux).

Si le calcul est plutôt coûteux dans la phase de factorisation numérique de MUMPS (cas le plus fréquent, cf. [U1.03.03]) et si les accès disque ne pénalisent pas trop l'usage de sa gestion mémoire OUT-OF-CORE (usage sur machine centralisée Aster), on peut même privilégier le second niveau. C'est souvent la stratégie la plus performante : par exemple pour un modèle comportant au moins 1M ddls, on peut allouer 6 processus MPI, chacun d'eux déployant 12 threads OpenMP : parallélisme hybride dit « 6MPIx12OpenMP ». *Via* l'outil Astk, cela s'effectuera, si les nœuds réservés au calcul parallèle sont constitués de 24 cœurs[3], en positionnant : ncpus à 12, mpi_nbcpu à 6 et mpi_nbnoeud à 3.

Pour rester efficace , il faut bien sûr veiller à **ne pas dépasser les capacités physiques des plate-formes** :

- pas plus de threads OpenMP (ncpus) que de cœurs partageant une mémoire physique (12 voire 24 sur Aster5) ;

3 C'est la cas de la machine centralisée actuelle, Aster5.

- pas plus de processus MPI (`mpi_nbcpu`), multipliés éventuellement par le nombre de threads précédent, que de cœurs physiques disponibles: par nœud (24 sur Aster5) et au total (taille de la machine) ;
- respecter les contraintes du gestionnaire batch et, en particulier, les ressources maximums allouables par un seul utilisateur.

Remarques:

- *Pour le seul parallélisme MPI des étapes de manipulation de systèmes linéaires (second cas de figure) il n'est pas utile d'allouer trop de processus. En général, la granularité de 1 processus MPI pour 30 ou 50. 10^3 ddls est largement suffisante.*
  Si on souhaite continuer à accélérer le calcul, on peut initier un deuxième niveau de parallélisme et réduire cette granularité à quelques milliers de ddls par threads. Par exemple, un modèle comportant 0.6M ddls pourra très probablement bénéficier d'un parallélisme efficace en allouant 12 MPI x 5 OpenMP=60 cœurs.
- *Pour `INFO_MODE` ou `CALC_MODES` (troisième cas de figure), on commence par distribuer les sous-bandes, puis on tient compte du parallélisme éventuel du solveur linéaire (si `MUMPS`). Par exemple, pour un calcul modal comportant 8 sous-bandes, on peut poser `mpi_nbcpu=32` . Chacune des sous-bandes va alors utiliser `MUMPS` sur 4 processus MPI. Et sur chacune de ces résolutions indépendantes on peut requérir le schéma parallèle précédent. Soit potentiellement trois niveaux de parallélisme imbriqués.*

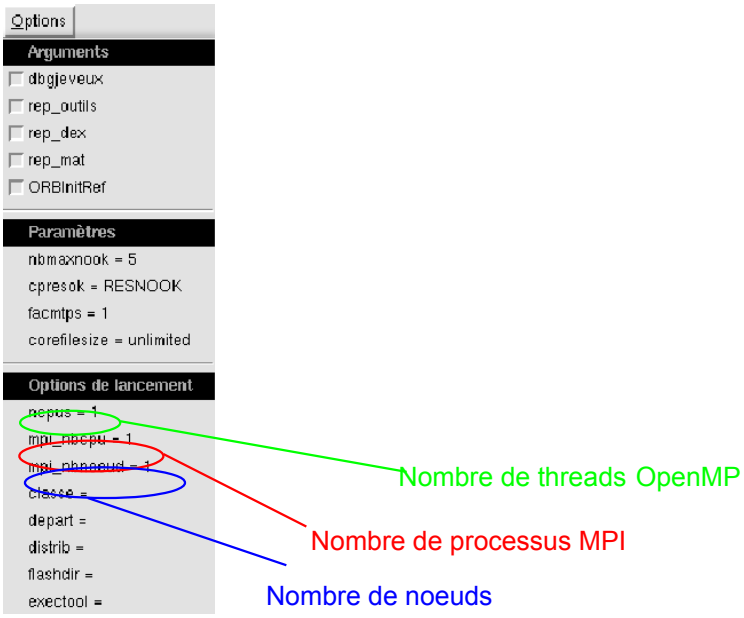

Figure 1.2._ Paramètres d'Astk dédiés au parallélisme.

Les chapitres de ce document détaillent tous ces éléments. Une fois ceux-ci fixés, on peut lancer son calcul comme on le ferait en séquentiel (sur la machine centralisée, en mode batch uniquement). Sauf, qu'avec le parallélisme, on peut bien sûr réduire les spécifications en temps et en mémoire du calcul renseignées dans Astk (cf. [U1.03.03]).
Par exemple, dans le second cas de figure, le fait de distribuer le calcul sur 12 processus MPI permet généralement de diminuer d'au moins :

- un facteur X2 son pic mémoire RAM (par processus MPI, cf. champ ' `Total memory` ' d'Astk),
- un facteur X3 son temps d'exécution (temps dit 'elapsed' ou temps d'attente « effectif » de retour du calcul, cf. champ ' `Time` ' d'Astk).

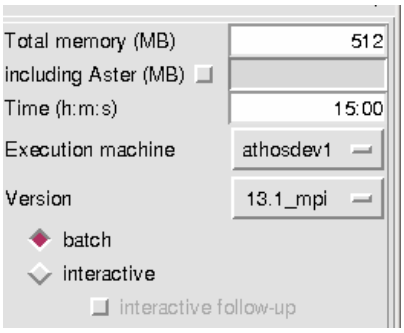

Figure 1.3._ Paramètres d'Astk consacrés aux ressources en temps et en mémoire RAM allouées (pour chaque processus MPI).

2 Généralités

Toute simulation *Code_Aster* peut bénéficier des gains de performance que procure le parallélisme. Du moment qu'il effectue des calculs élémentaires/assemblages, des résolutions de systèmes linéaires, de gros calculs modaux ou des simulations indépendantes/similaires. Les gains peuvent être de deux ordres : sur le temps de calcul et sur l'espace RAM/disque requis (par processus MPI).
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. Une fois paramétrées, la plupart s'enchaînent et se couplent de manière automatique. Les paragraphes suivant en font le synoptique.

2.1 Parallélismes informatiques

- **1a/ Lancement de rampes de calculs indépendants/similaires** (calculs paramétriques, tests...).
  Outil: scriptage shell.
  Gain: en temps elapsed.
  Lancement: standard *via* Astk[U2.08.07].
  Chaînage: aucun.
  Cumul: possibles avec les autres schémas parallèles mais uniquement en usage avancé (surcharge de sources).

- **1b/ Distribution des calculs élémentaires et des assemblages** matriciels et vectoriels dans les pré/post-traitements et dans les constructions de systèmes linéaires. Parallélisme en mémoire distribuée.
  Outil: langage MPI.
  Gain: en temps elapsed, voire en pic mémoire RAM avec `MUMPS+MATR_DISTRIBUEE`.
  Lancement: standard *via* Astk (`mpi_nbcpu/mpi_nbnoeud`).
  Chaînage: utile avec 2b ou 2c ; possible mais peu utile avec 1c ou 1d seuls ; possible mais inutile avec 2a.
  Cumul: aucun.

- **1c/ Distribution des calculs d'algèbre linéaire basiques** (sous-étapes de MUMPS, bibliothèque BLAS)**.** Parallélisme en mémoire partagée activé uniquement avec le package MUMPS (cf. 2b)**.**
  Outil: langage OpenMP.
  Gain: en temps elapsed (mais augmentation du temps CPU).
  Lancement: standard *via* Astk (`ncpus`).
  Chaînage: possible mais peu utile avec 1b seul.
  Cumul: contre-productif avec 2a, utile avec 2b, possible mais peu utile avec 2c ou 1d.

- **1d/ Distribution des calculs modaux** (resp. des calibrations modales) dans l'opérateur **CALC_MODES** (resp. **INFO_MODE**). Parallélisme en mémoire distribuée.
  Outil: langage MPI.
  Gain: en temps elapsed (mais augmentation du pic mémoire RAM à contrebalancer par le cumul avec 2b).
  Lancement: standard *via* Astk (`mpi_nbcpu/mpi_nbnoeud`).
  Chaînage: possible mais peu utile avec 1b seul.
  Cumul: utile avec 2b (voire 2b/1c) ; possible mais peu utile avec 2a; impossible avec 2c.

2.2 Parallélismes numériques

- **2a/ Solveur direct MULT_FRONT**; Parallélisme en mémoire partagée.
  Outil: langage OpenMP.

Gain: en temps elapsed (mais augmentation du temps CPU).
Lancement: standard *via* Astk (`ncpus`).
Chaînage: possible mais peu utile avec 1b, 2b ou 2c.
Cumul: contre-productif avec 1c, possible mais peu utile avec 1d.

- **2b/ Package `MUMPS`** (soit en tant que solveur direct, *via* `METHODE='MUMPS'` , soit en tant que préconditionneur de `PETSC/GCPC` *via* `PRE_COND='LDLT_SP'`). Parallélisme en mémoire distribuée.
  Outil: langage MPI.
  Gain: en temps elapsed et en pic mémoire RAM.
  Lancement: standard *via* Astk (`mpi_nbcpu/mpi_nbnoeud`).
  Chaînage: utile avec 1b ou 2c, possible mais peu inutile avec 2a.
  Cumul : utile avec 1c ou 1d.

- **2c/ Solveur itératif `PETSC`** (avec éventuellement `MUMPS` comme préconditionneur cf. `PRECOND='LDLT_SP'`); Parallélisme en mémoire distribuée.
  Outil: langage MPI.
  Gain: en temps elapsed et en pic mémoire RAM.
  Lancement: standard *via* Astk (`mpi_nbcpu/mpi_nbnoeud`).
  Chaînage: utile avec 1b ou 2b, possible mais inutile avec 2a.
  Cumul : possible mais peu utile avec 1c, hors-périmètre avec 1d.

Les schémas parallèles 1b+2b/1c ou 1b+2b+2c 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 pic RAM par coeur. Leur paramétrisation est simple, leur mise en œuvre facilitée *via* Astk (cf. §1).
Pour une utilisation standard, l'utilisateur n'a plus à se soucier de la mise en œuvre fine du parallélisme. En renseignant les menus dédiés d'Astk[4], on fixe le nombre de coeurs requis (pour le MPI et/ou l'OpenMP) ainsi que le nombre de nœuds sur lesquels ils se distribuent.

2.3 Parallélisation des systèmes linéaires

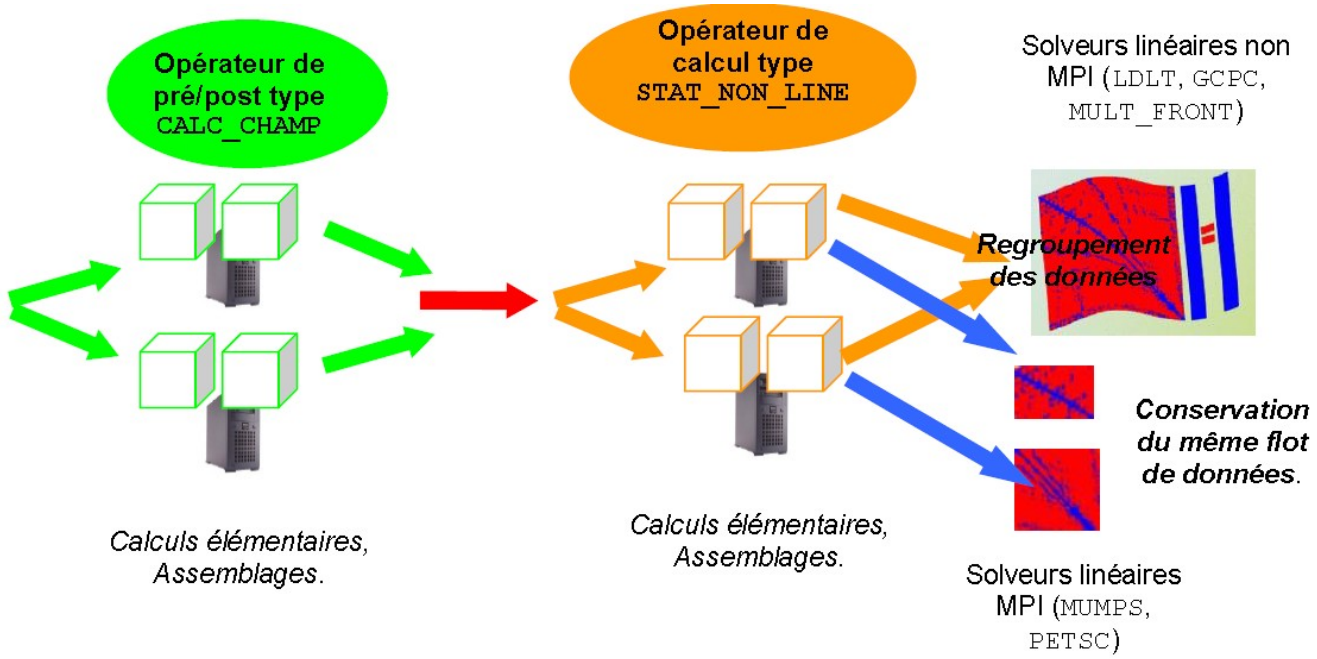

Figure 2.3.1._ Organisation du schéma parallèle MPI de construction et de résolution des systèmes linéaires.

4 Menus `Options+ncpus/mpi_nbcpus/mpi_nbnoeud`.

Dans le déroulement du fichier de commande *Code_Aster*, si plusieurs processus MPI sont activés (paramètre `mpi_nbcpu`), on commence par distribuer les mailles et/ou les sous-bandes fréquentielles entre les processeurs (stratégies parallèles pour 1b, 1d, 2b et 2c, cf figures 2.3.1/2.4.1). Cette distribution se décline de différentes manières et elle est paramétrable dans les opérateurs `AFFE/MODI_MODELE` (pour agir sur les schéma parallèles 1b, 2b et 2c), le mot-clé `SOLVEUR` (pour 2b et 2c) et `INFO_MODE/CALC_MODES` (pour le schéma parallèle 1d).

Le parallélisme le plus courant (stratégies 1b+2b, 1b+2c ou 1b+2b+2c) est fondé sur la **distribution des tâches** et des **structures de données** impliquées dans les manipulations de **systèmes linéaires** (cf. figure 2.3.1) . Car ce sont ces étapes de construction et de résolution de systèmes linéaires qui sont souvent les plus sollicitantes en temps de calcul et en ressources mémoire. Elles sont présentes dans la plupart des opérateurs car elles sont enfouies au plus profond d'autres algorithmes «plus métiers»: solveur non linéaire, calcul modal et vibratoire, schéma en temps...
Une fois la première étape de distribution des éléments finis du modèle effectuée entre tous les processus MPI (stratégie 1b), chacun ne va plus gérer que les traitements et les données associés aux éléments dont il a la charge. La construction des systèmes linéaires dans *Code_Aster* (calculs élémentaires, assemblages) s'en trouve alors accélérée. On parle souvent de « **parallélisme en espace** ». C'est un schéma parallèle plutôt d'ordre « informatique ».

Une fois ces portions de système linéaire construites (schéma parallèle 1b), deux cas de figures se présentent:

- soit le **traitement suivant est naturellement séquentiel** et donc tous les processus MPI doivent avoir accès à l'information globale. Pour ce faire on rassemble ces bouts de systèmes linéaires et donc l'étape suivante ne sera ni accélérée, ni ne verra baisser ses consommations mémoire. Il s'agit le plus souvent d'une fin d'opérateur, d'un post-traitement ou d'un solveur linéaire non parallélisé en MPI (stratégie 1b+2a).
- soit le **traitement suivant accepte le parallélisme MPI**, il s'agit alors principalement des solveurs linéaires HPC `MUMPS` (1b+2b), `PETSC` (1b+2c) ou les deux à la fois (1b+2b+2c). Le flot parallèle de données construit en amont leur est alors transmis (après quelques adaptations). Ces packages d'algèbre linéaire réorganisent ensuite, en interne, leurs propres schémas paralléles (avec une vision plus algébrique). On parle alors de schéma parallèle d'ordre plutôt « numérique ». Cette combinaison « parallélisme informatique », au niveau de l'assemblage du système linéaire, et, « parallélisme numérique », au niveau de sa résolution, les 2 *via* MPI, est la combinaison la plus courante.

Remarques:

- *Notons qu'à l'issue du cycle «construction de système linéaire – résolution de celui-ci », quelque soit le scénario mis en oeuvre (solveur linéaire séquentiel ou parallèle MPI), le vecteur solution est ensuite transmis, en entier, à tous les processus MPI. Le cycle peut ainsi continuer quelque soit la configuration suivante.*

De plus, on peut **superposer ou substituer à ce parallélisme MPI** (qui fonctionne sur toutes les plate-formes), un autre niveau de parallélisme géré cette fois par le **langage OpenMP**. Celui-ci est cependant limité aux fractions de machine partageant physiquement la même mémoire (PC multi-coeurs ou nœuds de serveur de calcul).
Il ne permet pas de baisser les consommations mémoire mais par contre il accélère certains types de calcul et ce, avec une granularité plus faible que celle du MPI : il procure une meilleure accélération même si le flot de données/traitements n'est pas très important. C'est un schéma parallèle d'ordre « informatique » qui intervient principalement dans les opérations basiques d'algorithmes d'algèbre linéaire (*via* par exemple la librairie BLAS).
Ce parallélisme peut être :

- soit complémentaire du parallélisme MPI en accélérant les calculs au sein de chaque processus MPI (dans la partie résolution de système linéaire avec `MUMPS`, stratégie dite « 2b/1c »).
- soit se substituer au parallélisme MPI en accélérant la résolution de système linéaire avec `MULT_FRONT` (stratégie 2a).

2.4 Distribution de calculs modaux

Lorsque la simulation ne peut pas se décomposer en calculs *Aster* indépendants, mais qu'elle reste **dominée** néanmoins **par des calculs modaux généralisés** (opérateurs INFO_MODE et CALC_MODES), on peut organiser un schéma parallèle spécifique.

Il est fondé sur la **distribution de calculs modaux** indépendants : chacun étant en charge d'une sous-bande fréquentielle . Ce schéma parallèle d'ordre purement « informatique » ne procure que des gains en temps.

Il peut toutefois cohabiter avec les schémas parallèles précédents :

- **chaînage** avec le parallélisme MPI de construction des systèmes linéaires (dans par exemple CALC_MATR_ELEM, stratégie 1b+1d),
- **cumul** avec le parallélisme MPI (voire OpenMP) dans les résolutions de systèmes linéaires (si utilisation de MUMPS, stratégie à 2 niveaux de parallélisme, 1d/2b voire trois, 1d/2b/1c).

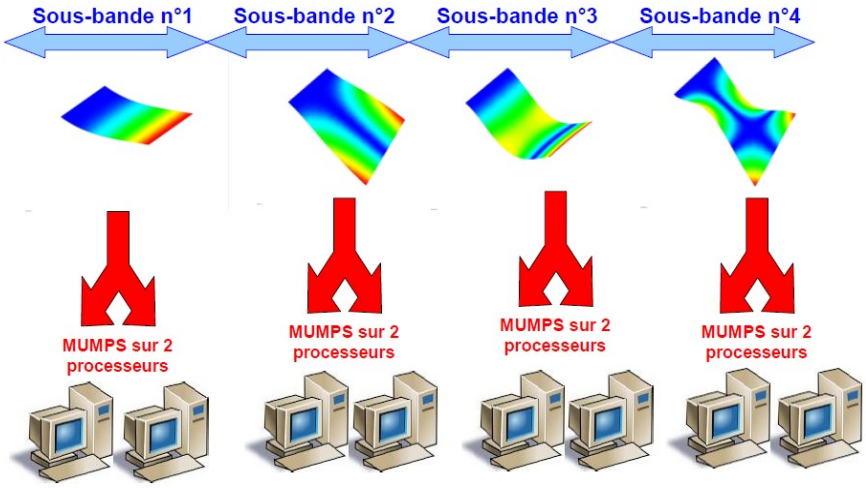

Figure 2.4.1._ Organisation du schéma parallèle MPI de distribution des calculs modaux et de résolutions des systèmes linéaires associés.

De toute façon, **tout calcul parallèle *Aster*** (sauf bien sûr le scénario 1a de calculs indépendants) **doit respecter les paradigmes suivant** : en fin d'opérateur de calcul[5], les bases globales de chaque processeur sont identiques[6] et le communicateur MPI courant est le communicateur standard (MPI_COMM_WORLD). Tous les autres éventuels sous-communicateurs MPI doivent être détruits.

Car on ne sait pas si l'opérateur qui suit dans le fichier de commandes 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 nœuds/processeurs...).

5 Ce n'est pas le cas d'opérateurs d'impression (par exemple, IMPR_RESU) ou de clôture de calcul (FIN/POURSUITE).

6 Et très proches de la base générée en mode séquentiel.

3 Quelques conseils préalables

On formule ici **quelques conseils pour aider l'utilisateur à tirer parti des stratégies de calcul parallèle du code**. Mais il faut bien être conscient, qu'avant tout chose, il faut d'abord optimiser et valider son calcul séquentiel en tenant compte des conseils qui fourmillent dans les documentations des commandes. Pour ce faire, on utilise, si possible, un maillage plus grossier et/ou on n'active que quelques pas de temps.

Le paramétrage par défaut et les affichages/alarmes du code proposent un fonctionnement équilibré et instrumenté. On liste ci-dessous et, de manière non exhaustive, plusieurs questions qu'il est intéressant de se poser lorsqu'on cherche à paralléliser son calcul. Bien sûr, certaines questions (et réponses) sont cumulatives et peuvent donc s'appliquer simultanément.

3.1 Préambule

Il est intéressant de **valider**, au préalable, **son calcul parallèle** en comparant quelques itérations en mode séquentiel et en mode parallèle. Cette démarche permet aussi de **calibrer les gains maximums atteignables** (speed-up théoriques) et donc d'éviter de trop «gaspiller de processeurs». Ainsi, si on note f la portion parallèle du code (déterminée par exemple *via* un run séquentiel préalable), alors le speed-up théorique S_p maximal accessible sur p processeurs se calcule suivant la formule d'Amdhal (cf. [R6.01.03] §2.4) :

$$S_p = \frac{1}{1 - f + \frac{f}{p}}$$

Par exemple, si on utilise le parallélisme MUMPS distribué par défaut (scénario 1b+2b) et que les étapes de construction/résolution de système linéaire représentent 90% du temps séquentiel (f=0.90), le speed-up théorique est borné à la valeur $S_\infty = \frac{1}{1-0.9+0.9/\infty} = 10$! Et ce, quelque soit le nombre de processus MPI alloués.

Il est intéressant **d'évaluer les principaux postes de consommation (temps/RAM) :** en mécanique quasi-statique, ce sont généralement les étapes de **calculs élémentaires/assemblages**, de **résolution de système linéaire** et les algorithmes de **contact-frottement**. Mais leurs proportions dépendent beaucoup du cas de figure (caractéristiques du contact, taille du problème, complexité des lois matériaux...). Si les calculs élémentaires sont importants, il faut les paralléliser *via* la scénario 1b (scénario par défaut). Si, au contraire, les systèmes linéaires focalisent l'essentiel des coûts, les scénarios 2b ou 2c peuvent suffire. Par contre, si c'est le contact-frottement qui dimensionne le calcul, il faut chercher à optimiser son paramétrage et/ou paralléliser son solveur linéaire interne (cf. méthode GCP+MUMPS).
En dynamique, si on effectue un calcul par projection sur base modale, cela peut être cette dernière étape qui s'avère la plus coûteuse. Pour gagner du temps, on peut alors utiliser l'opérateur CALC_MODES avec l'option 'BANDE' découpée en plusieurs sous-bandes, en séquentiel et, surtout, en parallèle (scénario 1d). Cet opérateur exhibe une distribution de tâches quasi-indépendantes qui conduit à de bons speedups.

Pour optimiser son calcul parallèle, il faut surveiller les éventuels **déséquilibres de charge** du flot de données (CPU et mémoire) et limiter les surcoûts dus **aux déchargements mémoire** (JEVEUX et MUMPS OOC) et aux **archivages de champs**. Sur le sujet, on pourra consulter la documentation [U1.03.03] «Indicateur de performance d'un calcul (temps/mémoire)». Elle indique la marche à suivre pour établir les bons diagnostics et elle propose des solutions.

Pour CALC_MODES avec l'option 'BANDE' découpée en plusieurs sous-bandes, **le respect du bon équilibrage de la charge est crucial pour l'efficacité du calcul parallèle** : toutes les sous-bandes doivent comporter un nombre similaire de modes. On conseille donc de procéder en trois étapes:

- Calibrage préalable de la zone spectrale par des appels à INFO_MODE(si possible en parallèle),

- Examen des résultats,
- Lancement en mode POURSUITE du calcul CALC_MODES, avec l'option 'BANDE' découpée en plusieurs sous-bandes parallélisées.

Pour plus de détails on pourra consulter la documentation utilisateur de l'opérateur[U4.52.02].

3.2 Quelques chiffres empiriques

On conseille d'allouer au moins 30 à 50.10^3 ddls par processus MPI (scénarios 1b, 2b ou 2c). Cette granularité peut descendre à quelques milliers de ddls par threads OpenMP tout en restant efficace (scénarios 1c ou 2a).
Un calcul thermo-mécanique standard bénéficie généralement, sur 32 processeurs, d'un gain de l'ordre de la dizaine en temps elapsed et d'un facteur 4 en pic mémoire RAM (par coeur).

Pour CALC_MODES, on conseille de décomposer son calcul en sous-bandes de quelques dizaines de modes (par exemple 20) et, ensuite, de prévoir 2, 4 voire 8 processus MUMPS par sous-bande. Il faut bien sûr composer avec le nombre de processeurs disponibles et le pic mémoire requis par le problème[7].
On peut obtenir des gains d'un facteur 30, en temps elapsed, sur une centaine de processeurs. Les gains en mémoire sont plus modestes (quelques dizaines de pourcents).
L'étape de calibration modale *via* INFO_MODE ne coûte elle pratiquement rien en parallèle: quelques minutes, tout au plus, pour des problèmes de l'ordre du million d'inconnus, parallélisés sur une centaine de processeurs. Les gains en temps sont d'un facteur X70 sur une centaine de processeurs et jusqu'à x2 en pic mémoire RAM.

3.3 Calculs indépendants

Lorsque la simulation que l'on souhaite effectuer se décompose naturellement (étude paramétrique, calcul de sensibilité...) ou, artificiellement (chaînage thermo-mécanique particulier...), en calculs similaires mais indépendants, on peut gagner beaucoup en temps calcul grâce au parallélisme numérique 1a.

3.4 Gain en mémoire RAM

Lorsque le facteur mémoire dimensionnant concerne la résolution de systèmes linéaires (ce qui est souvent le cas), le cumul des parallélismes informatiques 1b (calculs élémentaires/assemblages) et numériques 2b (solveur linéaire distribué MUMPS) est tout indiqué[8] (voire 1b+2b/1c).

Une fois que l'on a distribué son calcul *Code_Aster*+MUMPS/PETSC sur suffisamment de processeurs, les consommations RAM de JEVEUX peuvent devenir prédominantes (par rapport à celles de MUMPS que l'on a étalées sur les processeurs). Pour rétablir la bonne hiérarchie (le solveur externe doit supporter le pic de consommation RAM !) il faut activer, en plus, l'option SOLVEUR/MATR_DISTRIBUEE [U4.50.01].

Pour résoudre des problèmes frontières de très grandes tailles (> 5M ddls), on peut aussi essayer les solveurs itératifs de PETSC (stratégie parallèle 2c ou 2b+2c).

3.5 Gain en temps

7 Distribuer sur plus de processus, le solveur linéaire MUMPS permet de réduire le pic mémoire. Autres bras de levier: fonctionnement en Out-Of-Core et changement de renumeroteur.
8 Pour baisser les consommations mémoire de MUMPS on peut aussi jouer sur d'autres paramètres : OOC, usage en tant que préconditionneur ou relaxation des résolutions[U4.50.01].

Si l'essentiel des coûts concerne uniquement les résolutions de systèmes linéaires de petite taille (N<0.5M ddls) on peut se contenter d'utiliser les solveurs linéaires MUMPS en mode centralisé (stratégie 2b) ou MULT_FRONT (2a). Dès que la construction des systèmes devient non négligeable (>5%) ou que ceux-ci s'avèrent assez gros, il est primordial d'étendre le périmètre parallèle en activant la distribution des calculs élémentaires/assemblages (1b) et en passant à MUMPS distribué (valeur par défaut).

Sur des problèmes frontières de grandes tailles (N>3M ddls), une fois les bons paramètres numériques sélectionnés[9] (préconditionneur, relaxation... cf. [U4.50.01]), les solveurs itératifs parallèle (2c) peuvent procurer des gains en temps très appréciables par rapport aux solveurs directs génériques (2a/2b). Surtout si les résolutions sont relaxées[10] car, par la suite, elles sont corrigées par un processus englobant (algorithme de Newton de THER/STAT_NON_LINE...).

Pour les gros problèmes modaux (en taille de problème et/ou en nombre de modes), il faut bien sûr penser à utiliser CALC_MODES avec l'option 'BANDE' découpée en plusieurs sous-bandes[11].

9 Il n'y a par contre pas de règle universelle, tous les paramètres doivent être ajustés au cas par cas.

10 C'est-à-dire que l'on va être moins exigeant quant à la qualité de la solution. Cela peut passer par un critère d'arrêt médiocre, le calcul d'un préconditionneur frustre et sa mutualisation durant plusieurs résolutions de système linéaire... Les fonctionnalités des solveurs non linéaires et linéaires de *Code_Aster* permettent de mettre en œuvre facilement ce type de scénarios.

11 Equilibré *via* des pré-calibrations modales effectuées avec INFO_MODE. Si possible en parallèle.

4 Parallélismes informatiques

4.1 Rampes de calculs indépendants

4.1.1 Descriptif

Utilisation: grand public *via Astk*.
Périmètre d'utilisation: calculs indépendants (paramétrique, étude de sensibilité...).
Nombre de coeurs conseillés: limite de la machine/gestionnaire *batch*.
Gain: en temps CPU.
Speedup: proportionnel au nombre de cas indépendants.
Type de parallélisme: informatique *via* des scripts shell.
Scénario: 1a du §3. Cumul avec toutes les autres stratégies de parallélisme licite mais s'adressant à des utilisateurs avancés (hors périmètre d'*Astk*).

4.1.2 Mise en oeuvre

L'outil ***Astk* permet d'effectuer toute une série de calculs similaires mais indépendants** (en séquentiel et surtout en parallèle MPI). On peut utiliser une version officielle du code ou une surcharge privée préalablement construite. Les fichiers de commande explicitant **les calculs sont construits dynamiquement à partir d'un fichier de commande «modèle» et d'un mécanisme de type «dictionnaire» :** jeux de paramètres différents pour chaque étude (mot-clé `VALE` du fichier `.distr`), blocs de commandes *Aster*/Python variables (`PRE/POST_CALCUL`)...
Le lancement de ces rampes parallèles s'effectue avec les paramètres de soumission usuels d'*Astk*. On peut même re-paramétrer la configuration matérielle du calcul (liste des nœuds, nombre de cœurs, mémoire RAM totale par nœud...) *via* un classique fichier `.hostfile`.

Pour plus d'informations sur la mise en oeuvre de ce parallélisme, on pourra consulter les documentations [U1.04.00]/[U2.08.07].

Remarques :

- *Avant de lancer une telle rampe de calculs, il est préférable d'optimiser au préalable sur une étude type, les différents paramètres dimensionnants: gestion mémoire `JEVEUX`, aspects solveurs non linéaire/modaux/linéaire, mot-clé `ARCHIVAGE`, algorithme de contact-frottement... (cf. documentations [U1.03.03], [U4.50.01]...).*
- *On peut facilement écrouler une machine en lançant trop de calculs vis-à-vis des ressources disponibles. Il est conseillé de procéder par étapes et de se renseigner quant aux possibilités d'utilisation de moyens de calculs partagés (classe batch, gros jobs prioritaires...).*

4.2 Calculs élémentaires et assemblages

4.2.1 Descriptif

Utilisation: grand public *via Astk*.
Périmètre d'utilisation: calculs comportant des calculs élémentaires/assemblages coûteux (mécanique non linéaire). Activé par défaut dès que le nombre de processus MPI>1.
Nombre de coeurs conseillés: seul, entre 4 et 8. Chaîné avec le parallélisme distribué de MUMPS ou de PETSC (valeur par défaut), typiquement 16, 32 voire 64.
Gain: en temps voire en mémoire avec solveur linéaire MUMPS/PETSC (si MATR_DISTRIBUEE).
Speedup: Gains variables suivant les cas (efficacité parallèle[12]>50%). Il faut une assez grosse granularité pour que ce parallélisme reste efficace : 30 ou 50.10^3 ddls par processus MPI.
Type de parallélisme: informatique *via* la langage MPI (mpi_nbcpu/mpi_nbnoeud).
Scénario: 1b du §2. Nativement conçu pour se chaîner aux parallélismes numériques 2b ou 2c. Chaînage possible avec 1c, possible mais peu utile avec 1d ou 2a. Pas de cumul possible.

4.2.2 Mise en oeuvre

La mise en œuvre de ce schéma parallèle s'effectue de manière transparente pour l'utilisateur. *Via* Astk, elle s'initialise par défaut dès qu'on a sélectionné une version parallèle de *Code_Aster* (notée ***_mpi) ainsi qu'un nombre de processus MPI au moins égale à 2.
Ainsi sur le serveur centralisé *Aster*, il faut paramétrer les champs suivants dans le menu Options**:**

- **mpi_nbcpu=m,** nombre de processus MPI alloués.
- **mpi_nbnoeud=p,** nombre de noeuds sur lesquels vont être distribués ces processus MPI.

Par exemple, sur la machine centralisée Aster5, les nœuds sont composés de 24 coeurs. Pour allouer 32 processus MPI à raison de 8 processus par nœud, il faut donc positionner mpi_nbcpu à 32 et mpi_nbnoeud à 4.
On conseille, en général, de **ne pas allouer tous les cœurs d'un nœud en MPI seul** . Cela peut avoir pour effet de **ralentir la simulation** car, même si une partie des calculs s'en trouve accélérée du fait de sa distribution sur plus de cœurs, comme ceux-ci partagent certaines ressources mémoire, l es accès aux données sont, eux, ralentis.
Pour utiliser plus efficacement et à 100 % toutes les ressources allouées on conseille plutôt de **panacher parallélisme MPI et OpenMP** (cf. scénarios 1b+2b/1c ou 1b+2b/1c+2c).

Une fois ce nombre de processus MPI fixé, on peut lancer son calcul (en batch sur la machine centralisé) avec le même paramétrage qu'en séquentiel. Si ce schéma parallèle est chaîné avec le parallélisme numérique de MUMPS ou celui de de PETSC (ou les 2, cf. scénarios 2b et 2c), on peut réduire son pic mémoire RAM en activant l'option MATR_DISTRIBUEE.

Dès que plusieurs processus MPI sont activés, l'affectation du modèle dans le fichier de commandes *Aster* (opérateur AFFE_MODELE) **distribue ses mailles entre les processeurs**. *Code_Aster* étant un code éléments finis, c'est une distribution naturelle des données (et des tâches associées). Par la suite, les étapes *Aster* de calculs élémentaires et d'assemblages (matriciels et vectoriels) vont se baser sur cette distribution pour «tarir» les flots de données/traitements locaux à chaque processeur. Chaque processeur ne va effectuer que les calculs associés au groupe de maille dont il a la charge.

Cette **répartition maille/processeur** se décline de différentes manières et elle est paramétrable dans les opérateurs AFFE_MODELE[U4.41.01]/MODI_MODELE[U4.41.02] *via* les valeurs du mot-clé DISTRIBUTION/METHODE=:

12 On gagne au moins un facteur 2 (sur les temps consommés par les étapes parallélisées) en quadruplant le nombre de processeurs.

- 'CENTRALISE': **Les mailles ne sont pas distribuées** (comme en séquentiel). Chaque processeur connaît l'intégralité des mailles du modèle. 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 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' /'MAIL_DISPERSE' /'MAIL_CONTIGU' /'SOUS_DOMAINE' **(défaut)**/'SOUS_DOM.OLD': les mailles sont distribuées en se basant sur différents critères : par type, par distribution cyclique, par paquets de même taille ou suivant des stratégies sous-domaines.
  Dans les deux derniers scénarios, la distribution s'effectue *via* les partitionneurs METIS (défaut) ou SCOTCH (cf. mot-clé PARTITIONNEUR). Ceux-ci doivent donc être installés et linkés à la versions *Code_Aster* utilisée. C'est évidemment fait par défaut sur la machine centralisée.

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 doit rester compatible avec le paramétrage parallèle du calcul (nombre de nœuds/processeurs...).*

4.2.3 Structures de données distribuées

La distribution des données qu'implique ce type de parallélisme numérique ne diminue pas forcément les consommations mémoire JEVEUX. Par soucis de lisibilité/maintenabilité, les objets *Code_Aster* usuels sont initialisés avec la même taille qu'en séquentiel. Chaque processus MPI se «contente» juste de les remplir partiellement avec les données produites par les mailles dont a la charge le processeur. A charge pour le solveur linéaire parallèle utilisé dans la suite de l'opérateur (MUMPS, PETSC ou les 2) d'assembler ces données incomplètes et distribuées. On ne retaille donc généralement pas ces structures de données, elles comportent beaucoup de valeurs nulles.
Cette stratégie n'est tenable que tant que les objets JEVEUX principaux impliqués dans les calculs élémentaires/assemblages (CHAM_ELEM, RESU_ELEM, MATR_ASSE et CHAM_NO) ne dimensionnent pas les contraintes mémoire du calcul (cf. §5 de [U1.03.03]).
Normalement leur occupation mémoire est négligeable comparée à celle du solveur linéaire. Mais lorsque ce dernier (par ex. MUMPS) est lui aussi Out-Of-Core [13] et parallélisé en MPI (avec la répartition des données entre processeurs que cela implique), cette hiérarchie n'est plus forcément respectée. D'où l'introduction d' **une option** (mot-clé MATR_DISTRIBUEE cf. [U4.50.01]) **permettant de véritablement retailler, au plus juste, le bloc de matrice *Aster* propre à chaque processus MPI.**

Remarque :

- *En mode distribué, chaque processus MPI ne manipule que des matrices incomplètes (retaillées ou non). Par contre, afin d'éviter de nombreuses communications MPI (lors de l'évaluation des critères d'arrêt, calculs de résidus...), ce scénario n'a pas été retenu pour les vecteurs seconds membres. Leurs constructions sont bien parallélisées, mais, à l'issue de l'assemblage, les contributions partielles de chaque processus sont rassemblées. Ainsi, tout processus MPI connaît entièrement les vecteurs (CHAM_NO) impliqués dans le calcul.*

13 L'Out-Of-Core (OOC) est un mode de gestion de la mémoire qui consiste à décharger sur disque certains objets alloués par le code pour libérer de la RAM. La stratégie OOC permet de traiter des problèmes plus gros mais ces accès disque ralentissent le calcul. *A contrario*, le mode In-Core (IC) consiste à garder les objets en RAM. Cela limite la taille des problèmes accessibles mais privilégie la vitesse.

4.3 Distribution des calculs d'algèbre linéaire basiques

4.3.1 Descriptif

Utilisation: grand public *via Astk*.
Périmètre d'utilisation: calculs comportant des résolutions de systèmes linéaires *via* MUMPS (usage solveur direct ou préconditionneur de PETSC/GCPC).
Nombre de coeurs conseillés: sur Aster5, entre 2 et 12. Pas plus que de cœurs physiques partageant la même mémoire physique.
Gain: en temps elapsed (mais augmentation du temps CPU).
Speedup: Gains variables suivant les cas (efficacité parallèle[14]>50%). Une granularité faible suffit pour que ce parallélisme reste efficace : 10.10^3 ddls par threads OpenMP.
Type de parallélisme: informatique *via* le langage OpenMP (ncpus).
Scénario: 1c du §2. Une utilisation classique consiste à tirer parti d'un parallélisme hybride MPI+OpenMP pour accentuer les performances de MUMPS et de tirer partie à 100% des ressources machine (2b/1c ou (2b/1c)+2c).
Cumul contre-productif avec 2a, utile avec 2b, possible mais peu utile avec 2c (seul) ou 1d.

4.3.2 Mise en oeuvre

La mise en œuvre de ce schéma parallèle s'effectue de manière transparente pour l'utilisateur. *Via* Astk, elle s'initialise par défaut dès qu'on a sélectionné une version parallèle de *Code_Aster* (notée ***_mpi) ainsi qu'un nombre de threads OpenMP au moins égale à 2.
Ainsi sur le serveur centralisé *Aster*, il faut paramétrer les champs suivants dans le menu Options**:**

- **ncpus=k,** nombre de threads OpenMP alloués (par processus MPI si mpi_nbcpu>1).

Ce schéma parallèle est généralement utilisé en conjonction du parallélisme MPI de MUMPS. Car on conseille, en général, de **ne pas allouer tous les cœurs d'un nœud en MPI seul**. Cela peut avoir pour effet de **ralentir la simulation** car, même si une partie des calculs s'en trouve accélérée du fait de sa distribution sur plus de cœurs, comme ceux-ci partagent certaines ressources mémoire, les accès aux données sont, eux, ralentis. Pour utiliser plus efficacement et à 100% toutes les ressources allouées on conseille plutôt de **panacher parallélisme MPI et OpenMP** (cf. scénarios 2b/1c ou (2b/1c) +2c).
Dans ce type de parallélisme hybride (MPI/OpenMP), l'outil Astk[U1.04.00] complète automatiquement le nombre de threads en fonction des ressources machines, dès que le champ ncpus est laissé vide.
Par exemple, sur la machine centralisée *Aster5*, les nœuds sont composés de 24 coeurs. Si on souhaite organiser un parallélisme hybride 12MPIx4OpenMP, il suffit de positionner mpi_nbcpu à 12, mpi_nbnoeud à 2 et ncpus=<vide> (ou 4 explicitement).

Remarque:

- *La mise en oeuvre de ce parallélisme dépend du contexte informatique (matériel, logiciel) et des librairies d'algèbre linéaire utilisées. Sur la machine centralisée Aster, on utilise les BLAS threadées MKL.*

14 On gagne au moins un facteur 2 (sur les temps consommés par les étapes parallélisées) en quadruplant le nombre de processeurs.

4.4 Calculs modaux d' INFO_MODE/CALC_MODES

4.4.1 Descriptif

Utilisation: grand public *via Astk.*
Périmètre d'utilisation: calculs comportant de coûteuses recherches de modes propres.
Nombre de coeurs conseillés: plusieurs dizaines (par exemple, nombre de sous-bandes fréquentielles x 2, 4 ou 8).
Gain : en temps *elapsed v*oire en mémoire RAM (grâce au deuxième niveau de parallélisme).
Speedup: Gains variables suivant les cas: efficacité de l'ordre de 70% sur le premier niveau de parallélisme (sur les sous-bandes fréquentielles) complété par le parallélisme éventuel du second niveau (si SOLVEUR=MUMPS, efficacité complémentaire de l'ordre de 20%).
Type de parallélisme: informatique *via* le langage MPI (mpi_nbcpu/mpi_nbnoeud).
Scénario: 1d du §2. Nativement conçu pour se coupler au parallélisme 2b (voire 2b/1c).
Chaînage possible mais peu utile avec 1b. Cumul possible mais peu utile avec 2a et impossible avec 2c (hors périmètre).

4.4.2 Mise en oeuvre

L'usage de CALC_MODES avec l'option 'BANDE' découpée en plusieurs sous-bandes est à privilégier lorsqu'on traite des problèmes modaux **de tailles moyennes ou grandes** (>0.5M ddls) et/ou que l'on cherche une **bonne partie de leurs spectres** (> 50 modes).
On découpe alors le calcul en plusieurs sous-bandes fréquentielles. Sur chacune de ces sous-bandes, un solveur modal effectue la recherche de modes associée. Pour ce faire, ce solveur modal utilise intensivement un solveur linéaire.
Ces deux briques de calcul (solveur modal et solveur linéaire) sont les **étapes dimensionnantes** du calcul en terme de consommation mémoire et temps. C'est sur elles qu'il faut mettre l'accent si on veut réduire significativement les coûts calcul de cet opérateur.

Or, l'organisation du calcul modal sur des sous-bandes distinctes offre ici un cadre idéal de parallélisme: **distribution de gros calculs presque indépendants**. Son parallélisme permet de gagner beaucoup en temps mais au prix d'un surcoût en mémoire[15].
Si on dispose d'un nombre de processeurs suffisant (> au nombre de sous-bandes non vides), on peut alors enclencher un **deuxième niveau de parallélisme *via* le solveur linéaire** (si on a choisit METHODE='MUMPS'). Celui-ci permettra de continuer à gagner en temps mais surtout, il permettra de compenser le surcoût mémoire du premier niveau voire de diminuer notablement le pic mémoire séquentiel.

Pour un **usage optimal** de CALC_MODES avec l'option 'BANDE' découpée en plusieurs sous-bandes parallélisées, il est donc conseillé de :

- **Construire des sous-bandes de calcul relativement équilibrées**. Pour ce faire, on peut donc, au préalable, calibrer le spectre étudié *via* un appel à INFO_MODE [U4.52.01] (si possible en parallèle). Puis lancer le calcul CALC_MODES avec l'option 'BANDE' découpée en plusieurs sous-bandes parallélisées en fonction du nombre de sous-bandes choisies et du nombre de processeurs disponibles.

- De **prendre des sous-bandes** plus fines qu'en séquentiel, **entre 10 et 20 modes** au lieu de 40 à 80 modes en séquentiel. La qualité des modes et la robustesse du calcul s'en trouvera accrue. Le pic mémoire en sera diminué. Il reste cependant à avoir suffisamment de processeurs disponibles (et avec assez de mémoire).

- **Sélectionner un nombre de processeurs** qui est un multiple du nombre de sous-bandes (non vides). Ainsi, on réduit les déséquilibrages de charges qui nuisent aux performances.

15 Du fait des buffers MPI requis par les communications de vecteurs propres en fin de MODE_ITER_SIMULT.

Pour plus de détails on pourra consulter la documentation utilisateur de l'opérateur[U4.52.02].

5 Parallélismes numériques

5.1 Solveur direct MULT_FRONT

5.1.1 Descriptif

Utilisation: grand public *via Astk*.
Périmètre d'utilisation: calculs comportant des résolutions de systèmes linéaires coûteuses (en général STAT/DYNA_NON_LINE, MECA_STATIQUE...).
Nombre de coeurs conseillés: 2 ou 4.
Gain : en temps elapsed.
Speedup: Gains variables suivant les cas (efficacité parallèle≈50%). Il faut une bonne granularité pour que ce parallélisme reste efficace : au moins 50.10^3 ddls par coeur.
Type de parallélisme: numérique *via* le langage OpenMP (ncpus).
Scénario: 2a du §2. Chaînage possible mais peu utile avec 1b, 2b ou 2c. Cumul contre-productif avec 1c, possible mais peu utile avec 1d.

5.1.2 Mise en oeuvre

Cette méthode multifrontale développée en interne (cf. [R6.02.02] ou [U4.50.01] §3.5) est utilisée *via* la mot-clé SOLVEUR/METHODE='MULT_FRONT'. C'est le solveur linéaire (historique et auto-portant) préconisé par défaut en séquentiel sur les modèles de taille petite ou moyenne (<0.5M ddls).
La mise en oeuvre de ce parallélisme s'effectue de manière transparente pour l'utilisateur. Elle s'initialise par défaut dès qu'on lance un calcul *via Astk* (menu Options) utilisant plusieurs threads OpenMP.
Ainsi sur le serveur centralisé *Aster*, il faut paramétrer le champs suivants :

- ncpus=n, nombre de threads OpenMP alloués.

Une fois ce nombre de threads fixé on peut lancer son calcul (en batch sur la machine centralisé) avec le même paramétrage qu'en séquentiel. On peut bien sûr baisser les spécifications en temps du calcul.

5.2 Package MUMPS

5.2.1 Descriptif

Utilisation: grand public *via Astk*.
Périmètre d'utilisation: calculs comportant des résolutions de systèmes linéaires coûteuses (en général STAT/DYNA_NON_LINE, MECA_STATIQUE...).
Nombre de coeurs conseillés: chaîné avec le parallélisme distribué des calculs élémentaires/assemblages typiquement 16, 32 voire 64.
Gain: en temps CPU et en mémoire RAM.
Speedup: Gains variables suivant les cas (efficacité parallèle≈30%). Il faut une granularité moyenne pour que ce parallélisme reste efficace : entre 30 et 50.10^3 ddls par processus MPI.
Type de parallélisme: numérique *via* le langage MPI.
Scénario: 2b du §3. Nativement conçu pour se chaîner aux parallélismes 1b ou 2c. Chaînage possible mais peu utile avec 2a. Couplage très utile avec 1c ou 1d (voire les 2).

5.2.2 Mise en œuvre

Cette méthode multifrontale s'appuie sur le produit externe MUMPS (cf. [R6.02.03] ou [U4.50.01] §3.7) est utilisée soit en tant que solveur direct (mot-clé SOLVEUR/METHODE='MUMPS'), soit en tant que préconditionneur des solveurs itératifs PETSC ou GCPC (mot-clé SOLVEUR/PRE_COND='LDLT_SP').
C'est le package HPC conseillé pour exploiter pleinement les gains CPU/RAM que peut procurer le parallélisme. Ce type de parallélisme est performant (surtout lorsqu'il est chaîné avec 1b et couplé avec 1c) tout en restant générique, robuste et grand public.

La mise en œuvre de ce schéma parallèle s'effectue de manière transparente pour l'utilisateur. *Via* Astk, elle s'initialise par défaut dès qu'on a sélectionné une version parallèle de *Code_Aster* (notée ***_mpi) ainsi qu'un nombre de processus MPI au moins égale à 2.
Ainsi sur le serveur centralisé *Aster*, il faut paramétrer les champs suivants dans le menu Options**:**

- **mpi_nbcpu=m,** nombre de processus MPI alloués.
- **mpi_nbnoeud=p,** nombre de noeuds sur lesquels vont être distribués ces processus MPI.

Par exemple, sur la machine centralisée Aster5, les nœuds sont composés de 24 coeurs. Pour allouer 32 processus MPI à raison de 8 processus par nœud, il faut donc positionner mpi_nbcpu à 32 et mpi_nbnoeud à 4.
On conseille, en général, de **ne pas allouer tous les cœurs d'un nœud en MPI seul** . Cela peut avoir pour effet de **ralentir la simulation** car, même si une partie des calculs s'en trouve accélérée du fait de sa distribution sur plus de cœurs, comme ceux-ci partagent certaines ressources mémoire, l es accès aux données sont, eux, ralentis.
Pour utiliser plus efficacement et à 100 % toutes les ressources allouées on conseille plutôt de **panacher parallélisme MPI et OpenMP** (cf. scénarios 1b+2b/1c ou 1b+2b/1c+2c).

Idéalement, ce solveur linéaire HPC doit être utilisé en mode parallèle distribué (DISTRIBUTION/METHODE='GROUP_ELEM'/'MAIL_DISPERSE'/'MAIL_CONTIGU'/'SOUS_DOMAINE'/'SOUS_DOM.OLD'). C'est-à-dire qu'il faut avoir initié en amont de ce solveur linéaire, au sein des procédures de calculs élémentaires/assemblages, des flots de données/traitements distribués (scénario parallèle 1b). MUMPS accepte en entrée ces données incomplètes et il les rassemble en interne. On ne perd pas ainsi de temps (comme c'est le cas pour les autres solveurs linéaires) à compléter les données issues de chaque processeur. Ce mode de fonctionnement est activé par défaut dans les commandes AFFE/MODI_MODELE (cf. §4.2).
En mode centralisé (CENTRALISE), la phase amont de construction des systèmes linéaires n'est pas parallélisée (chaque processeur procède comme en séquentiel). MUMPS ne tient alors compte que des données issues du processeur maître.

Dans le premier cas, le code est parallèle de la construction du système linéaire jusqu'à sa résolution (chaînage des parallélismes 1b+2b), dans le second cas, on n'exploite le parallélisme MPI que sur la partie résolution (parallélisme 2b).

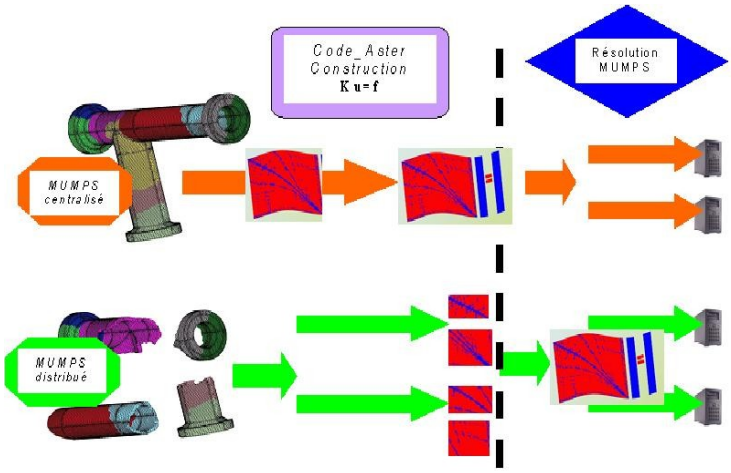

Figure 5.2.1._ Flots de données/traitements parallèles du couplage Code_Aster+MUMPS suivant le mode d'utilisation: centralisé ou distribué.

Remarque :

Lorsque la part du calcul consacrée à la construction du système linéaire est faible (<5%), les deux modes d'utilisation (centralisé ou distribué) affichent des gains en temps similaires. Par contre, seule l'approche distribuée procure, en plus, des gains sur les consommations RAM.

5.3 Solveur itératif PETSC

5.3.1 Descriptif

Utilisation: grand public *via Astk*.
Périmètre d'utilisation: calculs comportant des résolutions de systèmes linéaires coûteuses (en général STAT/DYNA_NON_LINE, MECA_STATIQUE...). Plutôt des problèmes non linéaires de grandes tailles.
Nombre de coeurs conseillés: chaîné avec le parallélisme distribué des calculs élémentaires/assemblages (1b), voire celui du préconditionneur MUMPS (2b), typiquement 16, 32 voire 64.
Gain: en temps CPU et en mémoire RAM (suivant les préconditionneurs).
Speedup: gains variables suivant les cas (efficacité parallèle>50%). Il faut une granularité moyenne pour que ce parallélisme reste efficace : 50.10^3 ddls par processus MPI.
Type de parallélisme: numérique *via* le langage MPI.
Scénario: 2c du §3. Nativement conçu pour se chaîner aux parallélismes 1b ou 2b ; chaînage possible mais peu utile avec 2a. Cumul possible mais peu utile avec 1c, hors-périmètre avec 1d.

5.3.2 Mise en œuvre

Cette bibliothèque de solveurs itératifs (cf. [R6.01.02] ou [U4.50.01] §3.9) est utilisée *via* la mot-clé SOLVEUR/METHODE='PETSC'. Ce type de solveur linéaire est conseillé pour traiter, soit des problèmes frontières de très grande taille (>5M ddls), soit en non linéaire, pour tirer pleinement partie de la mutualisation du préconditionneur entre différents pas de Newton.

La mise en œuvre de ce parallélisme s'effectue comme pour le package MUMPS (cf. §5.2).

Remarque:

- *Contrairement aux solveurs parallèles directs (MUMPS, MULT_FRONT), les itératifs ne sont pas universels (ils ne peuvent pas être utiliser en modal) et toujours robustes. Ils peuvent être très compétitifs (en temps et surtout en mémoire), mais il faut trouver le point de fonctionnement (algorithme, préconditionneur...) adapté au problème. Toutefois, sur ce dernier point, l'usage généralisé (et paramétré par défaut) de MUMPS simple précision comme préconditionneur (PRE_COND='LDLT_SP') a considérablement amélioré les choses.*