Structures de données distribuées et parallélisme

Résumé :

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

Table des matières

1 Qu'est ce qu'une structure de données répartie ?....3
2 Comment se fait la distribution ?....3
3 Qu'elles sont les structures de données concernées ?....4
4 Comment manipule-t-on de telles structures de données ?....4
5 Cas particulier (MATR_DISTRIBUEE)....4
5.1 Règle à respecter lors de la programmation d'un flot de données/traitements parallèles....5

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

1 Qu'est ce qu'une structure de données répartie ?

Actuellement, le code propose deux stratégies pour fonctionner en parallèle via MPI:

- Pré et/ou post-traitements utilisant des calculs élémentaires (par ex. avec l'opérateur *Aster* CALC_CHAMP),
- Construction, assemblage et résolution des systèmes linéaires avec les produits externes MUMPS ou PETSc (par ex. STAT_NON_LINE). On résout ici en parallèle le problème global usuel: on est en mono-domaine.

Ces deux stratégies de parallélisme mettent en œuvre une distribution des tâches et des données associées. C'est-à-dire que les processeurs vont allouer les structures de données usuelles, les initialiser mais ils ne vont les remplir que partiellement. Pour faciliter la maintenance et la lisibilité du code, on ne retaille généralement pas ces structures de données au plus juste. Elles comportent donc beaucoup de valeurs nulles. Pour détecter le caractère complet ou incomplet de telles SD, un de leur attribut prend la valeur **MPI_COMPLET** ou **MPI_INCOMPLET**.

Remarques :

Pour plus d'informations sur ces outils (utilisables aussi en séquentiel), on peut consulter les manuels de Référence R6.01.02/03 et R6.02.03, ainsi que le manuel d'Utilisation U4.50.01.
Pour gagner en performance, le code propose deux autres stratégies de parallélisme complémentaires: en mémoire partagée (OpenMP *) avec le solveur linéaire interne* MULT_FRONT *(R6.02.02) et, avec l'outil* ASTK *qui permet de lancer des études indépendantes.*

2 Comment se fait la distribution ?

Code_Aster étant un code éléments finis, la distribution de données mise en œuvre pour piloter le parallélisme est une distribution par maille (*'EOD'* pour *Element Oriented Distribution*). Chaque processeur est ainsi responsable d'un paquet de mailles (physiques ou tardives) et ne va donc effectuer que les calculs élémentaires, les assemblages et les remplissages de structures de données correspondant.

Cette répartition des mailles entre les processeurs se fait au niveau de l'affectation du modèle (commande AFFE_MODELE). Suivant le mode de distribution choisi par l'utilisateur (MAIL_DISPERSE, MAIL_CONTIGU..) et le nombre de processeurs alloués pour le job, le code opère la répartition MAILLE/PROC et la stocke dans la structure de données PARTITION. Cette répartition peut d'ailleurs être modifiée en cours de calcul *via* la commande *MODI_MODELE*.

Remarque :

Mailles tar dives : Mailles qui n'existent pas dans le maillage initial et qui sont rajoutées au cours du calcul pour faciliter la mise en données. On les rencontre le plus souvent avec les conditions de Dirichlet imposées avec AFFE_CHAR_MECA ou lorsqu'on utilise la méthode de contact continue
Pour faciliter la gestion des mailles tardives celles-ci sont automatiquement assignées au processeur maître. Cette surcharge de travail induit généralement un faible déséquilibrage de charge. Dans le cas contraire, l'utilisateur peut rétablir l'équilibre en déchargeant ce processeur (via le mot-clé CHARGE_PROC0_MA/SD) d'un certain pourcentage des mailles physiques qui lui sont initialement attribuées (ou d'un certain nombre de sous-domaines si la distribution est orientée sous-domaines).

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

3 Qu'elles sont les structures de données concernées ?

Pour l'instant on ne "distribue" que les SD résultant d'un calcul élémentaire (*RESU_ELEM* et *CHAM_ELEM*) ou d'un assemblage matriciel (*MATR_ASSE*). Les vecteurs (*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. La caractérisation *MPI_COMPLET* ou *_INCOMPLET* intervient dans:

- CELK(7) pour les *CHAM_ELEM* (D4.06.05 – structure de Données champ),
- NOLI(3) pour les *RESU_ELEM* (D4.06.05 – structure de Données champ)
- *REFA(11) pour les MATR_ASSE* (D4.06.05 – structure de Données sd_matr_asse),

4 Comment manipule-t-on de telles structures de données ?

Puisque ces structures de données conservent la même organisation et les mêmes dimensions que leur version séquentielle, on peut les manipuler avec les routines utilitaires standards. Cependant, à chaque traitement global (c'est-à-dire impliquant potentiellement toutes les mailles du modèle), il faut détecter leur caractère complet ou incomplet (*via* les flags du paragraphe précédent) et adopter une stratégie *ad hoc*.

Par exemple, avant un produit matrice-vecteur on peut compléter la *MATR_ASSE* si nécessaire (*via sdmpic*) ou décider de s'arrêter en *ERREUR_FATALE* si ce scénario correspond à un cheminement logiciel imprévu. Dans d'autres cas de figures, on ne cherche pas à compléter la SD mais on établit les communications MPI correspondant au traitement global concerné (par ex. recherche de maximum comme dans *assmam*).

Remarque :

Pour faciliter la maintenance, l'installation et la validation du code, on limite les appels MPI à quelques routines utilitaires: mummpi, fetmpi et mpicm1 . Si on veut développer de nouvelles communications, il vaut mieux ainsi enrichir une de ces routines
Version séquentielle : On devrait en toute rigueur les qualifier de centralisée. Car même un calcul parallèle peut conduire à des MATR_ASSE ou des RESU_ELEM non distribuées. Il suffit pour cela que l'utilisateur choisisse, pour tout le calcul ou juste pour une portion, le parallélisme de type CENTRALISE dans les opérateurs AFFE/MODI_MODELE . Les calculs élémentaires et les assemblages ne sont alors pas parallélisés, seul le solveur linéaire l'est (MUMPS ou PETSc).

5 Cas particulier (MATR_DISTRIBUEE)

1) Lorsqu'on effectue un calcul distribué avec le solveur MUMPS , si on active l'option MATR_DISTRIBUEE , les bouts de MATR_ASSE propres à chaque processeur sont retaillés de manière à ne pas stocker de zéros inutiles (REFA(11) est alors taggé en MATR_DISTR) . Mais du coup la manipulation de ces matrices assemblées distribuées "amaigries" est plus complexe. Lors de certains traitements globaux (produit matrice-vecteur), on ne les complète pas et on s'arrête en ERREUR_FATALE . Il faudra enrichir le périmètre d'utilisation de cette fonctionnalité au coup par coup.

Remarques :

Pour distinguer les données associées à chaque sous-domaine, chacun comporte une SD_SOLVEUR , un NUME_DDL , une MATR_ASSE et des CHAM_NO s adaptés à ses dimensions. D'autre part, chaque processeur est responsable d'une SD_SOLVEUR , d'un NUME_DDL , d'une MATR_ASSE et de CHAM_NOs maîtres qui vont être quasi-vides et dont la seule fonction est de pointer vers les SD esclaves des sous-domaines dont le processeur à la gestion. C'est une autre forme de SD distribuées qui implique donc cette récursivité à deux niveaux.

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

5.1 Règle à respecter lors de la programmation d'un flot de données/traitements parallèles

En fin d'opérateur *Aster ,* toutes les bases globales des processeurs doivent être identiques car on ne sait pas si l'opérateur qui suit dans le fichier de commande a prévu un flot de données d'entrée incomplet. Il faut donc organiser, par exemple, les complétudes de champs *ad hoc* dans les routines d'archivage (en essayant de sa baser sur des `MPI_ALLREDUCE` plus simples et plus efficaces à mettre en œuvre). Pour offrir un parallélisme plus efficace il faudra un jour faire sauter ce verrou. Mais d'ici là !

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