Mise en œuvre d'une Bibliothèque d'Outils pour la Résolution par Méthode Directe de Grands Systèmes Linéaires Creux Symétriques Définis Positifs sur Machine Parallèle

Validation sur un Code de Mécanique des Structures

Manuel Utilisateur de la chaîne EMILIO

David Goudin

Décembre 2000

LaBRI, Université Bordeaux I – ENSEIRB
CNRS UMR 5800
351, cours de la Libération, F-33405 Talence Cedex, France

Table des matières

1. Description du contenu du document

Ce document est le rapport utilisateur de la chaîne logicielle EMILIO issue du travail de thèse réalisé par David Goudin au sein de l'équipe **ALiENor** du LaBRI de manière contractuelle pour le CEA/CESTA sur le sujet intitulé

Mise en œuvre d'une Bibliothèque d'Outils pour la Résolution par Méthode Directe de Grands Systèmes Linéaires Creux Symétriques Définis Positifs sur Machine Parallèle Validation sur un Code de Mécanique des Structures.

Il reprend les documents intitulés :

- *Pré-étude de la parallélisation du code OSSAU de mécanique des structures* [1] qui constituait le rapport de DEA de David Goudin (juillet 1997),
- le premier rapport semestriel d'avancement de la première partie [2] (mars 1998),
- le rapport final de cette première partie [3] (février 1999),
- le rapport semestriel d'avancement de la seconde partie [4] (décembre 1999),
- la thèse de David Goudin [5].

Toutes les notions fondamentales et les conclusions contenues dans ces documents ne sont pas rappelées ici et supposées acquises. On renvoie plus particulièrement au chapitre 3 de la référence [5].

Pour chacun des modules de la bibliothèque, et selon l'ordre logique de l'enchaînement de la chaîne, on donne successivement :

- un bref rappel du rôle du module,
- une description de l'interface des fonctionnalités du module,
- les options de compilation/exécution disponibles pour le module.

La dernière section donne une description des différentes structures de données utilisées par ces modules.

Les membres de l'équipe ALiENor participant à ce projet sont en plus de David Goudin, Pascal Hénon, François Pellegrini, Pierre Ramet et Jean Roman (Responsable), et ce en collaboration avec Jean-Jacques Pesqué du CEA/CESTA.

2. Le module SCOTCH

2.1. Rôle du module

Le module SCOTCH reprend l'ensemble du code de la bibliothèque LIBSCOTCH, qui fait partie de la distribution logicielle SCOTCH de placement statique et de renumérotation par blocs de matrices creuses développé au LaBRI.

Les routines du module ne sont appelées que par les routines correspondantes du module ORDER, qui sert d'interface entre les structures de données spécifiques au projet EMILIO et celles utilisées au sein de la LIBSCOTCH.

Les algorithmes de renumérotation utilisés représentent l'état de l'art actuel en la matière. Pour plus de renseignements, veuillez vous reporter aux différentes publications parues sur le projet SCOTCH en ce qui concerne les aspects théoriques, et sur le manuel d'utilisation de SCOTCH pour les aspects d'implémentation.

2.2. Interface

Veuillez vous reporter sur le manuel d'utilisation de SCOTCH.

2.3. Paramétrage

Le paramétrage des routines du module SCOTCH est entièrement reporté au niveau du module ORDER.

2.4. Compilation

La compilation s'effectue de façon classique, au moyen des fichiers `makefile` et `make_gen`. Le module ne possède aucun paramètre spécifique de compilation à part le drapeau de débogage du module, SCOTCH_DEBUG.

3. Le module ORDER

3.1. Rôle du module

Le module ORDER réalise la renumérotation d'un graphe décrit sous forme de listes d'adjacences ou de façon algorithmique, afin de minimiser le nombre de termes non nuls dans la matrice et la masse de calculs de factorisation à réaliser.

L'utilisation de ce module est extrêmement simple, puisqu'elle ne nécessite l'appel que d'une seule routine parmi celles accessibles à l'utilisateur.

Les routines accessibles à l'utilisateur prennent toutes une description de graphe (sous la forme de tableaux d'adjacences ou sous forme algorithmique), et retournent une description de renumérotation par blocs de type Order.

Les algorithmes de renumérotation utilisés pour les grilles bidimensionnelles et tridimensionnelles sont basés sur des dissections emboîtées telles que chaque sous-volume courant est coupé en deux moitiés égales (à un hyper-plan près) selon un hyper-plan perpendiculaire à la plus grande dimension du sous-volume.

Les algorithmes de renumérotation utilisés pour les graphes généraux sont ceux implémentés au sein de la libScotch (voir description du module SCOTCH). Ils sont basés sur l'utilisation combinée de plusieurs algorithmes, dont la stratégie d'utilisation est décrite par une chaîne de caractères d'un format prédéfini, appelée chaîne de stratégie. Pour plus de précisions sur la structure des chaînes de stratégie de renumérotation, veuillez vous reporter au manuel utilisateur de la distribution logicielle Scotch.

3.2. Interface

3.2.1. orderGraph

- ```
  int orderGraph (Order * const        ordeptr,
                  const Graph * const  grafptr );
  ```

- La routine orderGraph calcule une renumérotation du graphe qui lui est passé en paramètre, au moyen de la stratégie de renumérotation par défaut.

  Les numéros affectés aux sommets par le processus de renumérotation peuvent commencer à partir de 0 ou 1, selon que le graphe a été déclaré en utilisant une numérotation commençant à 0 (style C) ou à 1 (style Fortran).

- La routine orderGraph retourne 0 si la structure Order a été calculée avec succès, et 1 sinon, en cas de pénurie de mémoire.

3.2.2. orderGraphList
  ```

-
```
int orderGraph (Order * const        ordeptr,
                const Graph * const  grafptr,
                const INT            listnbr,
                const INT * const    listtab );
```

- La routine `orderGraphList` calcule une renumérotation d'un sous-ensemble des sommets du graphe qui lui est passé en paramètre, au moyen de la stratégie de renumérotation par défaut.

  La liste des sommets à considérer est constituée du nombre de sommets, `listnbr`, et d'un tableau à `listnbr` entrées, numérotées de 0 à (`listnbr` -1), et dont chaque élément est un indice de sommet dans la structure `Graph`, exprimé par rapport à l'index de début de la numérotation du graphe. Les indices de numéros peuvent donc commencer à partir de 0 ou 1, selon que le graphe a été déclaré en utilisant une numérotation commençant à 0 (style C) ou à 1 (style Fortran).

  Les numéros affectés aux sommets listés commencent également à partir de 0 ou 1, en fonction du style de numérotation adopté pour le `Graph`.

- La routine `orderGraphList` retourne 0 si la structure `Order` a été calculée avec succès, et 1 sinon, en cas de pénurie de mémoire.

3.2.3. `orderGraphStrat`

-
```
int orderGraphStrat (Order * const        ordeptr,
                     const Graph * const  grafptr,
                     const char * const   stratptr );
```

- La routine `orderGraphStrat` calcule une renumérotation du graphe qui lui est passé en paramètre, au moyen de la chaîne de stratégie fournie.

  Les numéros affectés aux sommets par le processus de renumérotation peuvent commencer à partir de 0 ou 1, selon que le graphe a été déclaré en utilisant une numérotation commençant à 0 (style C) ou à 1 (style Fortran).

- La routine `orderGraphStrat` retourne 0 si la structure `Order` a été calculée avec succès, et 1 sinon, en cas de pénurie de mémoire.

3.2.4. `orderGraphListStrat`

-
```
int orderGraphListStrat (Order * const        ordeptr,
                         const Graph * const  grafptr,
                         const INT            listnbr,
                         const INT * const    listtab,
                         const char * const   stratptr );
```

- La routine `orderGraphListStrat` calcule une renumérotation d'un sous-ensemble des sommets du graphe qui lui est passé en paramètre, au moyen de la stratégie fournie.

  La liste des sommets à considérer est constituée du nombre de sommets, `listnbr`, et d'un tableau à `listnbr` entrées, numérotées de 0 à (`listnbr` -1), et dont chaque élément est un indice de sommet dans la structure `Graph`, exprimé par rapport à l'index de début de la numérotation du graphe. Les indices de numéros peuvent donc commencer à partir de 0 ou 1, selon que le graphe a été déclaré en utilisant une numérotation commençant à 0 (style C) ou à 1 (style Fortran).

  Les numéros affectés aux sommets listés commencent également à partir de 0 ou 1, en fonction du style de numérotation adopté pour le `Graph`.

- La routine `orderGraphListStrat` retourne 0 si la structure `Order` a été calculée avec succès, et 1 sinon, en cas de pénurie de mémoire.

3.2.5. `orderGrid2`

-
```
int orderGrid2 (Order * const  ordeptr,
                const INT      xnbr,
                const INT      ynbr,
                const INT      baseval,
                const INT      xlim,
                const INT      ylim );
```

- La routine `orderGrid2` calcule une renumérotation d'une grille 2D de taille `xnbr` $\times$ `ynbr` au moyen de dissections emboîtées, de telle sorte que chaque sous-volume courant soit coupé en deux moitiés égales (à un hyper-plan près) selon un hyper-plan perpendiculaire à la plus grande dimension courante du sous-volume. Les dissections emboîtées terminent lorsque l'une au moins des dimensions courantes de chaque sous-volume est inférieure aux limites de récursion fixées, à savoir `xmin` et `ymin`, respectivement. Cette renumérotation convient à la fois à des grilles à 4 et à 8 voisins.

  Les numéros affectés aux sommets renumérotés commencent à partir de la valeur `baseval` passée en paramètre.

- La routine `orderGrid2` retourne 0 si la structure `Order` a été calculée avec succès, et 1 sinon, en cas de pénurie de mémoire.

3.2.6. `orderGrid3`

- ```
  int orderGrid3 (Order * const  ordeptr,
                  const INT      xnbr,
                  const INT      ynbr,
                  const INT      znbr,
                  const INT      baseval,
                  const INT      xlim,
                  const INT      ylim,
                  const INT      zlim );
  ```

- La routine `orderGrid3` calcule une renumérotation d'une grille 3D de taille `xnbr` × `ynbr` × `znbr` au moyen de dissections emboîtées, de telle sorte que chaque sous-volume courant soit coupé en deux moitiés égales (à un hyper-plan près) selon un hyper-plan perpendiculaire à la plus grande dimension courante du sous-volume. Les dissections emboîtées terminent lorsque l'une au moins des dimensions courantes de chaque sous-volume est inférieure aux limites de récursion fixées, à savoir `xmin`, `ymin` et `zmin`, respectivement. Cette renumérotation convient à la fois à des grilles à 6 et à 26 voisins.

  Les numéros affectés aux sommets renumérotés commencent à partir de la valeur `baseval` passée en paramètre.

- La routine `orderGrid3` retourne 0 si la structure `Order` a été calculée avec succès, et 1 sinon, en cas de pénurie de mémoire.

3.3. Paramétrage

Les routines du module `ORDER` ne nécessitent aucun paramétrage particulier. La chaîne de stratégie de renumérotation par défaut est définie au sein de la routine `orderGraphList` du fichier `order_graph.c`, et peut être modifiée pour répondre à des besoins spécifiques. Cependant, la chaîne actuelle donne satisfaction dans la plus grande majorité des cas, et sa modification ne doit donc être envisagée qu'avec la plus grande circonspection. Il est bien plus intéressant, pour tester une nouvelle chaîne, d'utiliser les routines de la forme `orderGraph*Strat`, qui permettent directement de passer la chaîne de stratégie en paramètre.

3.4. Compilation

La compilation s'effectue de façon classique, au moyen des fichiers `makefile` et `make_gen`. Le module ne possède aucun paramètre spécifique de compilation à part le drapeau de débogage du module, `ORDER_DEBUG`.
  ```

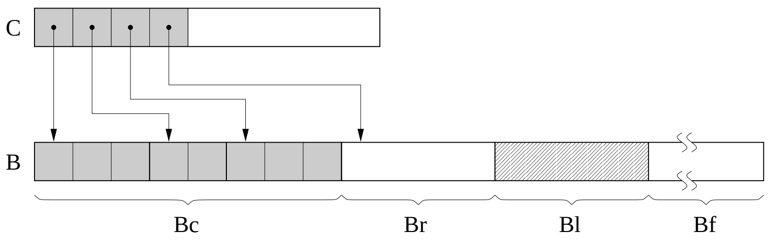

Figure 1. Structures de données par blocs d'une matrice en cours de factorisation symbolique. Le tableau de blocs colonnes C indice le tableau de blocs B. B est un tableau linéaire de taille croissante, qui est rempli bloc colonne par bloc colonne. Bc est le sous-tableau des blocs colonnes déjà calculés, Br est la zone réservée au bloc colonne en cours de calcul, Bl est la zone utilisée par la liste chaînée des blocs fusionnés à partir de Bc, et Bf est l'espace libre qui grossit sur demande à mesure que les nouveaux blocs sont calculés.

4. Le module `FAX`

4.1. Rôle du module

Le module `FAX` réalise la factorisation symbolique par blocs d'une matrice renumérotée par le module `ORDER`.

L'utilisation de ce module est extrêmement simple, puisqu'elle ne nécessite l'appel que d'une seule routine parmi celles accessibles à l'utilisateur.

Les routines accessibles à l'utilisateur prennent toutes une description de graphe (sous la forme de tableaux d'adjacences ou sous forme algorithmique) et une description de renumérotation par blocs, et retournent une `SymbolMatrix` correspondant à la factorisation symbolique de ce graphe.

L'algorithme de factorisation utilise des structures de données par blocs, qui permettent d'obtenir des complexités en temps et en espace quasi-linéaires par rapport au nombre d'arêtes du graphe à factoriser ; de plus, les structures de données par blocs sont bien plus compactes que le stockage compressé par colonnes habituellement utilisé dans les algorithmes de factorisation par colonnes.

L'algorithme implémenté tire le plus possible parti des caches (voir figure 1). Les blocs de la matrice factorisée sont calculés et ajoutés, bloc colonne par bloc colonne, à un tableau de blocs de taille croissante, de telle sorte que les blocs de contributions des blocs colonnes déjà calculés puissent être facilement fusionnés pour construire les blocs du bloc colonne courant.

4.2. Interface

4.2.1. `symbolFax`

```
int symbolFax (SymbolMatrix * const  symbptr,
               const INT             vertnbr,
               const INT             edgenbr,
               const INT             degrmax,
               const INT             baseval,
               void * const          ngbdptr,
               INT                   ngbfrst (void * const, const INT),
               INT                   ngbnext (void * const),
               const Order * const   ordeptr );
```

- La routine `symbolFax` calcule la factorisation symbolique du graphe qui lui est passé en paramètre sous forme algorithmique, et dont les sommets ont été renumérotés selon la rerumérotation `ordeptr`. Elle permet de calculer la structure factorisée par blocs d'une matrice de grande taille sans avoir besoin de construire les graphes associés lorsque ceux-ci possèdent une grande régularité, comme par exemple pour les grilles 2D ou 3D.

  Cette routine générale permet de factoriser par blocs n'importe quelle structure de graphe, régulière ou non. Un certain nombre de routines de plus haut niveau ont été définies pour les structures `Graph` explicites et les grilles 2D et 3D, et sont décrites ci-après.

  Le graphe est décrit par les paramètres suivants :

  – `vertnbr` : nombre de sommets du graphe ;

  – `edgenbr` : nombre d'arcs, c'est-à-dire deux fois le nombre d'arêtes ;

  – `degrmax` : degré maximum du graphe, c'est-à-dire nombre maximum d'arêtes adjacentes à tout sommet. Cette valeur sert à déterminer la taille des structures de données internes servant à compacter les indices des voisins de chaque sommet, et peut donc être un majorant du degré maximum réel, mais pour des raisons d'efficacité et d'occupation mémoire il est conseillé d'utiliser une valeur la plus proche possible du degré maximum réel.

  – `baseval` : index de début de la numérotation du graphe. Cette valeur peut être égale à 0 ou 1, selon qu'on appelle cette routine à partir de programmes utilisant une numérotation commençant à 0 (style C) ou à 1 (style Fortran).

  – `ngbdptr` : pointeur sur une zone de données servant à conserver l'état courant de l'itérateur sur les voisins d'un sommet.

  – `ngbfrst` : pointeur sur une fonction renvoyant le numéro du premier voisin du sommet dont le numéro lui est passé en deuxième paramètre. Le premier paramètre passé à cette routine lors de chaque appel est le pointeur `ngbdptr`, permettant ainsi d'initialiser cette structure pour faciliter la détermination des voisins suivants. La valeur retournée par la fonction `ngbfrst` est le numéro du premier voisin du sommet, la numérotation commençant à partir de `baseval`. Si le sommet est isolé, la fonction doit retourner la valeur (`baseval` − 1).

– `ngbnext` : pointeur sur une fonction renvoyant le numéro du voisin suivant. Le seul paramètre passé à cette routine est le pointeur `ngbdptr`, qui permet d'accéder et de mettre à jour les informations nécessaires à l'obtension du voisin suivant. Les numéros des voisins d'un sommet peuvent être donnés dans n'importe quel ordre. Lorsque tous les voisins d'un sommet ont déjà été donnés, la fonction doit retourner la valeur (`baseval` $- 1$).

La structure `symbptr` à remplir doit avoir été préalablement initialisée au moyen de la routine `symbolInit`. La structure `ordeptr` doit contenir une renumérotation valide, calculée au moyen d'une des routines du module `ORDER` (voir section 3).

- La routine `symbolFax` retourne 0 si la `SymbolMatrix` a été calculée avec succès, et 1 sinon, en cas de pénurie de mémoire.

4.2.2. `symbolFaxGraph`

-
```
int symbolFaxGraph (SymbolMatrix * const  symbptr,
                    const Graph * const   grafptr,
                    const Order * const   ordeptr );
```

- La routine `symbolFaxGraph` calcule la factorisation symbolique du graphe `grafptr` qui lui est passé en paramètre, et dont les sommets ont été renumérotés selon la permutation directe `ordeptr`.

  La structure `symbptr` à remplir doit avoir été préalablement initialisée au moyen de la routine `symbolInit`.

- La routine `symbolFaxGraph` retourne 0 si la `Symbol Matrix` a été calculée avec succès, et 1 sinon, en cas de pénurie de mémoire.

4.2.3. `symbolFaxGrid2D`

-
```
int symbolFaxGrid2D (SymbolMatrix * const  symbptr,
                     const INT             xnbr,
                     const INT             ynbr,
                     const INT             baseval,
                     const Order * const   ordeptr );
```

- La routine `symbolFaxGrid2D` calcule la factorisation symbolique de la grille 2D à quatre voisins (voir figure 2.a) dont la taille et la renumérotation sont passés en paramètre.

  Les paramètres entiers `xnbr` et `ynbr` représentent les dimensions de la grille, et `baseval` est l'indice initial servant à la numérotation des sommets de la grille, qui peut commencer à 0 (style C) ou à 1 (style Fortran). La structure `symbptr` à remplir doit avoir été préalablement initialisée au moyen de la routine `symbolInit`.

- La routine `symbolFaxGrid2D` retourne 0 si la `Symbol Matrix` a été calculée avec succès, et 1 sinon, en cas de pénurie de mémoire.

a. Quatre voisins. **b.** Huit voisins.

Figure 2. Exemples de grilles 2D à 4 et 8 voisins.

4.2.4. `symbolFaxGrid2E`

- ```
  int symbolFaxGrid2E (SymbolMatrix * const  symbptr,
                       const INT             xnbr,
                       const INT             ynbr,
                       const INT             baseval,
                       const Order * const   ordeptr );
  ```
- La routine `symbolFaxGrid2E` calcule la factorisation symbolique de la grille 2D à huit voisins (voir figure 2.b) dont la taille et la renumérotation sont passées en paramètre.

  Les paramètres entiers `xnbr` et `ynbr` représentent les dimensions de la grille, et `baseval` est l'indice initial servant à la numérotation des sommets de la grille, qui peut commencer à 0 (style C) ou à 1 (style Fortran). La structure `symbptr` à remplir doit avoir été préalablement initialisée au moyen de la routine `symbolInit`.
- La routine `symbolFaxGrid2E` retourne 0 si la `Symbol Matrix` a été calculée avec succès, et 1 sinon, en cas de pénurie de mémoire.

4.2.5. `symbolFaxGrid3D`

- ```
  int symbolFaxGrid3D (SymbolMatrix * const  symbptr,
                       const INT             xnbr,
                       const INT             ynbr,
                       const INT             znbr,
                       const INT             baseval,
                       const Order * const   ordeptr );
  ```
- La routine `symbolFaxGrid3D` calcule la factorisation symbolique de la grille 3D à six voisins dont la taille et la renumérotation sont passées en paramètre.

  Les paramètres entiers `xnbr`, `ynbr` et `znbr` représentent les dimensions de la grille, et `baseval` est l'indice initial servant à la numérotation des sommets de la grille, qui peut commencer à 0 (style C) ou à 1 (style Fortran). La structure `symbptr` à remplir doit avoir été préalablement initialisée au moyen de la routine `symbolInit`.

- La routine `symbolFaxGrid3D` retourne 0 si la `Symbol Matrix` a été calculée avec succès, et 1 sinon, en cas de pénurie de mémoire.

4.2.6. symbolFaxGrid3E

- ```
  int symbolFaxGrid3E (SymbolMatrix * const  symbptr,
                       const INT             xnbr,
                       const INT             ynbr,
                       const INT             znbr,
                       const INT             baseval,
                       const Order * const   ordeptr );
  ```

- La routine `symbolFaxGrid3E` calcule la factorisation symbolique de la grille 3D à vingt-six voisins dont la taille et la renumérotation sont passés en paramètre.

  Les paramètres entiers `xnbr`, `ynbr` et `znbr` représentent les dimensions de la grille, et `baseval` est l'indice initial servant à la numérotation des sommets de la grille, qui peut commencer à 0 (style C) ou à 1 (style Fortran). La structure `symbptr` à remplir doit avoir été préalablement initialisée au moyen de la routine `symbolInit`.

- La routine `symbolFaxGrid3D` retourne 0 si la `Symbol Matrix` a été calculée avec succès, et 1 sinon, en cas de pénurie de mémoire.

4.3. Paramétrage

Les routines du module `FAX` ne nécessitent aucun paramétrage particulier, à part la donnée des routines de premier voisin et de voisin suivant dans la routine `symbolFax`.

4.4. Compilation

La compilation s'effectue de façon classique, au moyen des fichiers `makefile` et `make_gen`. Le module ne possède aucun paramètre spécifique de compilation à part le drapeau de débogage du module, `FAX_DEBUG`.

Le corps de l'algorithme de factorisation par blocs, `symbol Fax`, se trouve dans le fichier `fax.c`. La routine `symbolFax` est appelée de façon normale par les routines `faxGrid*` contenues dans le fichier `fax_grid.c`. En revanche, pour éviter l'appel répété aux itérateurs de sommets et d'arêtes dans le cas des graphes classiques, la routine `symbolFaxGraph` inclut directement le fichier `fax.c`, en redéfinissant au préalable les identificateurs d'appel aux itérateurs pour accéder directement aux tableaux de sommets et d'arêtes. La duplication de code objet qui en résulte ne s'effectue donc pas au prix d'une duplication du code source.
  ```

5. Le module SACHA

5.1. Rôle du module

Ce module calcule une “bonne” distribution des éléments du maillage à partir de la distribution des blocs de la matrice calculée par le module BLEND (cf. section 6). Cette distribution des éléments équilibre le calcul des matrices et/ou des seconds membres élémentaires ainsi que les mises à jour dans la matrice globale distribuée.

5.2. Interface

le module SACHA dispose d'une seule fonction externe;

- ```
  int sacha(Order *          ordemesh,
            SolverMatrix *   solve_matrix,
            Assembly1D *     assembly_matrix,
            Mesh *           mesh,
            Dof *            matrdeof,
            DistribElts *    distrib_elts);
  ```

En entrée on a donc :

- une structure Order qui contient la nouvelle numérotation des nœuds du maillage, ainsi que la permutation inverse , ce qui permet de retrouver la numérotation initiale issue du mailleur;
- une Solvermatrix contenant toutes les informations locales au processeur courant (voir module BLEND);
- une structure Assembly1D renseignée par le module BLEND et qui permet de connaître la relation nœud - bloc colonne - processeur propriétaire, ainsi que d'autres informations nécessaires pour le module SACHA et le module PABLAS;
- une structure Mesh qui contient la description du maillage lu lors du pré-traitement avec les numéros des éléments et les numéros de nœuds (avec notre numérotation) composant cet élément. Ces numéros sont triés dans l'ordre croissant afin d'améliorer les temps d'accés aux structures 1D;
- une structure Dof qui donne le nombre de ddl par nœud;
- une structure DistribElts qui, une fois complétée par le module SACHA contiendra toutes les informations sur la distribution des éléments.

En sortie de SACHA on aura :

- la structure DistribElts locale au processeur courant; cette structure contient les éléments attribués à ce processeur (éléments totalement et non totalement locaux) et des informations indispensables pour gérer les communications lors de l'assemblage parallèle.
  ```

5.3. Paramétrage

Les routines du module `SACHA` ne nécessitent aucun paramétrage particulier.

5.4. Compilation

La compilation s'effectue de façon classique, au moyen des fichiers `makefile` et `make_gen`. Le module ne possède aucun paramètre spécifique de compilation à part des drapeaux de débogage et de prise de temps situés à l'intérieur du module.

6. Le module BLEND

6.1. Rôle du module

Il s'agit du partitionneur/distributeur de la chaîne EMILIO. Ce module participe au prétraitement de la factorisation numérique effectuée dans le module `SOPALIN`. Son rôle est de :

- repartitionner la matrice symbolique issue du module `FAX` de manière à exploiter le parallélisme induit par les blocs denses dans le solveur parallèle;
- distribuer les blocs colonnes sur chaque processeur;
- établir un ordonnancement efficace des calculs et des communications pour le solveur parallèle.

Il y a deux manières de l'utiliser:

- soit en faisant un appel à `BLEND` sur chaque processeur participant à la résolution en parallèle: dans ce cas, en sortie de l'appel à `BLEND`, il suffira d'utiliser le solveur parallèle avec la `SolverMatrix` locale;
- soit en générant en un seul appel à `BLEND` toutes les `SolverMatrix` correspondant à tous les processeurs que l'on souhaite utiliser dans la résolution parallèle; dans ce cas `BLEND` générera autant de fichiers correspondant aux `SolverMatrix` qui devront être chargées sur les processeurs concernés.

6.2. Interface

le module `BLEND` dispose d'une seule fonction externe;

-
```
void solverBlend(SolverMatrix *  solvmtx,
                 Assembly1D *    assemb,
                 int             procnbr,
                 int             procnum,
                 BlendParam *    option,
                 const Dof *     const dofptr);
```

En entrée on a donc :

- une `Solvermatrix` dont la `SymbolMatrix` est renseignée (à ce stade la `SymbolMatrix` décrit la matrice entière);
- une structure `Assembly1D` qui, une fois calculée dans `BLEND` sera utilisée par le module `SACHA` pour la distribution des éléments;
- le nombre de processeurs qu'utilisera le solveur parallèle `SOPALIN`;

- le numéro du processeur pour lequel `BLEND` doit générer sa `SolverMatrix` locale;
- une structure `BlendParam` correctement initialisée (voir plus loin dans cette section) qui contient le paramétrage de `BLEND`;
- une structure `Dof` qui donne le nombre de ddl par nœuds.

En sortie de `BLEND` on aura donc:

- la `SolverMatrix` contenant toutes les informations locales au processeur `procnum` et nécessaires à l'exécution du solveur parallèle. Ces informations sont: la description des bloc-colonnes locaux, l'ordonnancement du calcul des bloc-colonnes ainsi que l'ordonnancement des envois de "cibles fan-in" afin de maximiser le recouvrement calcul/communication;
- la structure `Assembly1D` locale au processeur `procnum`.

6.3. Paramétrage

Le paramétrage de `BLEND` se fait à l'aide d'une structure `BlendParam`: comme les autres structures de la chaîne `EMILIO`, elle dispose d'une procédure d'initialisation, et d'une procédure de désallocation:

```
int      blendParamInit(BlendParam *);
void     blendParamExit(BlendParam *);
```

Il est intéressant de noter que la procédure d'initialisation fixe la valeur par défaut de toutes ces options. Voici un exemple d'utilisation ainsi qu'un commentaire pour les options modifiables par l'utilisateur :

```
BlendParam          blendpar;
blendParamInit(&blendpar);
blendpar.verbose = 1;
blendpar.tracegen = 1;
blendpar.leader = 0;
blendpar.trace_filename = "oimbe.trf";
blendpar.sequentiel = 0;
blendpar.solvmtx_filename = "solvmtx.";
blendpar.timer = 1;
blendpar.assembly = 0;
blendpar.blcolmin  = 60;

solverBlend(&solvmatr, &assemb, procnbr, 0, &blendpar, &dofstr);
```

Commentaire sur les options paramétrables à l'aide de la structure `BlendParam` :

- **verbose** : autorise le fonctionnement en mode bavard de `BLEND`; ceci produit l'affichage des différentes étapes de `BLEND` ainsi que quelques autres informations comme la prédiction du temps nécessaire à la factorisation par le solveur parallèle;

- **tracegen** : permet la génération d'un fichier de trace PICL utilisable dans un environnement ParaGraph; les traces générées sont une prédiction de l'exécution de la factorisation numérique parallèle;
- **leader** : lorsque l'option **tracegen** est sélectionnée, ce numéro correspond au numéro du processeur qui générera le fichier de trace PICL;
- **trace_filename** : nom du fichier de trace PICL;
- **sequentiel** : permet la génération de toutes les `SolverMatrix`, dans des fichiers séparés, pour tous les processeurs;
- **solvmtx_filename** : préfixe utilisé pour le nom des fichiers contenant les `SolverMatrix`, le suffixe de ce nom étant le numéro du processeur concerné;
- **timer** : permet l'affichage du temps mis pour l'exécution de `BLEND`;
- **assembly** : active le remplissage de la structure `Assembly1D` pour la distribution des éléments;
- **blcolmin** : donne la largeur minimum des bloc-colonnes pour une utilisation efficace des primitives `BLAS`.

6.4. Compilation

La compilation s'effectue de façon classique, au moyen des fichiers `makefile` et `make_gen`. Le module ne possède aucun paramètre spécifique de compilation à part le drapeau de débogage du module, `BLEND_DEBUG`.

7. Le module PABLAS

7.1. Rôle du module

Ce module réalise l'assemblage parallèle de la matrice de rigidité globale et/ou du second membre. Il utilise les informations qui lui sont données par toutes les étapes séquentielles qui l'ont précédées : la distribution des éléments (elle même induite par la distribution des blocs de la matrice), la renumérotation des nœuds effectuée dans le module ORDER et le maillage dupliqué sur l'ensemble des processeurs.

7.2. Interface

le module PABLAS dispose d'une fonction externe pour l'assemblage de la matrice, cette fonction a été adaptée pour les besoins de son intégration dans le code OSSAU;

```
void emiliopablas(Order *        ordemesh,
                  SolverMatrix * solve_matrix,
                  Assembly1D *   assembly_infos,
                  Mesh *         mesh,
                  DistribElts *  elts_distrib,
                  Dof *          matrdeof,
                  INT *          ossau_imat,
                  INT *          ossau_imt,
                  INT *          ossau_nbpel,
                  INT *          ossau_ityp,
                  INT *          ossau_nint,
                  INT *          ossau_nflg,
                  FLOAT *        ossau_cord,
                  INT *          ossau_nop,
                  FLOAT *        ossau_ort,
                  FLOAT *        ossau_r2,
                  FLOAT *        ossau_deltat,
                  FLOAT *        ossau_helem,
                  FLOAT *        ossau_felem,
                  FLOAT *        ossau_sigma0,
                  FLOAT *        ossau_bidon);
```

En entrée on a donc :

- tous les paramètres de type INT * ou FLOAT * qui correspondent à des informations fournies ou renvoyées au code OSSAU lors de l'appel dans la routine FORTRAN. Deux paramètres sont particulièrement importants : ossau_helem et ossau_felem qui contiennent respectivement la matrice élémentaire et le second membre élémentaire calculés par les routines FORTRAN du code OSSAU;

- la `SolverMatrix` contenant toutes les informations données par les étapes séquentielles précédentes. Le seul champ manquant est le tableau des coefficients de la matrice globale distribuée sur le processeur courant, cette allocation se fait juste avant l'assemblage et ces tableaux seront renseignés par l'assemblage parallèle lui même; bloc colonne - processeur propriétaire, ainsi que d'autres informations nécessaires pour le module `SACHA` et le module `PABLAS`;

- une structure `Assembly1D` qui permet de connaitre la distribution des nœuds du maillage sur les différents processeurs et donc faire les envois de messages pour les éléments non totalement locaux;

- une structure `Mesh` qui contient la description du maillage lu lors du pré-traitement;

- une structure `DistribElts` renseignée par le module `SACHA` et qui contient toutes les informations nécessaires à l'assemblage pour "piloter" les calculs des matrices élémentaires des éléments attribués et les communications induites par certains de ces éléments;

- une structure `Dof` qui donne le nombre de ddl par nœud.

En sortie de `PABLAS` on aura donc :

- une `SolverMatrix` locale au processeur courant, et qui contient les coefficients de la matrice globale de rigidité locale au processeur.

7.3. Paramétrage

Les routines du module `PABLAS` ne nécessitent aucun paramétrage particulier.

7.4. Compilation

La compilation s'effectue de façon classique, au moyen des fichiers `makefile` et `make_gen`. Le module ne possède aucun paramètre spécifique de compilation à part des drapeaux de débogage et de prise de temps situés à l'intérieur du module. Il faut néammoins faire attention à certains drapeaux qui ont été affectés en fonction des cas tests du code OSSAU.

8. Le module SOPALIN

8.1. Rôle du module

Il s'agit du solveur parallèle de la chaîne EMILIO qui utilise les stratégies d'ordonnancement du partitionneur/distributeur BLEND.

Le solveur réalise actuellement une factorisation LDL^t sur une distribution 1D; il a été conçu pour pouvoir évoluer facilement vers une distribution 2D. Les traitements appliqués sur la matrice creuse travaillent au niveau d'un bloc ou d'un bloc colonne pour tirer parti au mieux des possibilités des routines BLAS. On considère enfin une version "Fan-In" pour laquelle on calcule localement l'accumulation de l'ensemble des contributions à un même bloc-colonne non local. Pour optimiser les effets de recouvrement et gérer l'irrégularité du creux, on utilise les routines de communication MPI non blocantes, en particulier l'émission asynchrone.

8.2. Interface

le module SOPALIN dispose de deux fonctions externes; l'appel à chacunes de ces fonctions est très simple :

```
void sopalin(SolverMatrix *m);
void up_down(SolverMatrix *m);
```

Il est nécessaire de passer en paramètre une `Solver Matrix` dont tous les champs sont renseignés à l'exception des coefficients des `Fan-In Targets` qui seront alloués dynamiquement.
De plus, il faut avoir effectué la factorisation (appel à `sopalin()`) avant de pouvoir exécuter une étape de "descente/remontée" (appel à `up_down()`). Il est par contre possible d'enchaîner plusieurs étapes de "descente/remontée" sans avoir à re-factoriser la matrice.

8.3. Paramétrage

Il n'y a pas de paramètre particulier à positionner pour le module SOPALIN. Certaines fonctionnalités sont activables/désactivables à l'aide d'options de compilation (voir le paragraphe suivant).

8.4. Compilation

Dans cette section nous allons présenter les principales options de compilation du module SOPALIN. On précise **[true]** ou **[false]** suivant qu'une option est activée ou non par défaut dans la version livrée.

Les options de "debug"

- `#define SOLVER_DEBUG` [**true**] : cette option est positionnée lorsque l'on compile en mode "debug" au niveau du projet, elle permet d'activer certaines instructions de vérification (également le champ `color` dans la structure `Symbol Matrix` utilisée pour les traces PICL);
- `#define DUMP` [**false**] : cette option permet de d'afficher une partie de la matrice entre chaque étape de la factorisation (voir la fonction `dump()`);
- `#define COMPUTE` [**true**] : les opérations de calcul BLAS ne sont pas effectuées si on désactive cette option;
- `#define DEBUG_SOPALIN` [**false**] : cette option permet d'afficher les étapes importantes de la factorisation (et la "descente/remontée") ainsi que les valeurs des variables caractéristiques :

```
#ifdef DEBUG_SOPALIN /* Select type of DEBUG */
#define DEBUG_SOPALIN_PHASE1
#define DEBUG_SOPALIN_PHASE2
#define DEBUG_SOPALIN_COMPUTE_CONTRIB
#define DEBUG_SOPALIN_COMPUTE_CONTRIB_COMPACT
#define DEBUG_SOPALIN_ADD_CONTRIB_LOCAL
#define DEBUG_SOPALIN_COMPUTE_CONTRIB_LOCAL
#define DEBUG_SOPALIN_ADD_CONTRIB_TARGET
#define DEBUG_SOPALIN_COMPUTE_CONTRIB_TARGET
#define DEBUG_SOPALIN_ADD_TARGET
#define DEBUG_SOPALIN_ENVOYER
#define DEBUG_SOPALIN_RECEVOIR
#define DEBUG_SOPALIN_DOWN
#define DEBUG_SOPALIN_UP
#define DEBUG_SOPALIN_INIT
#define DEBUG_SOPALIN_MAIN
#endif
```

Les options d'optimisation

- `#define COMPACT_CTRB` [**true**] : cette option permet d'activer le calcul des opérations BLAS sur un ensemble de blocs (correctement alignés) lorsque cela est possible;
- `#define SAVE_TRSM` [**true**] : cette option permet l'utilisation de calculs partiels;
- `#define ALLOC_FTGT` [**true**] : l'allocation des `Fan-In Targets` est réalisée dynamiquement lorsque cette option est activée. Dans le cas contraire l'ensemble des `Fan-In Targets` est alloué lors de l'initialisation du module `SOPALIN`.

Les options liées à la "descente/remontée"

- `#define SOLVER_UPDOWN` [**true**] : cette option doit être positionnée si on souhaite effectuer une étape de "descente/remontée";
- `#define ASYNC_SMX` [**true**] : permet d'utiliser ou non les routines de communication asynchrones lors de la "descente/remontée" (pour la factorisation les routines de communications sont toujours asynchrones);
- `#define FANIN_SMX` [**true**] : lorsque cette option est active, on utilise un schéma de calcul avec aggrégation comme pour la factorisation;
- `#define DUMP_SMX` [**false**] : cette option permet d'afficher une partie du vecteur second-membre/solution entre chaque étape de la "descente/remontée" (voir la fonction `dump_smx()`).

Les options liées à des fonctionnalités supplémentaires

- `#define TRACE_PICL` [**false**] : active ou non la génération des traces PICL; chaque processeur génère un fichier dont le nom est contenu dans la variable `tracefilename`. Ces fichiers doivent ensuite être concaténés, puis il faut trier le fichier résultant à l'aide de la commande `tracesort` de Paragraph;
- `#define COMPUTE_ALLOC` [**false**] : cette option permet d'obtenir au cours du temps le pourcentage que représente le surcoût en mémoire des `Fan-In Targets` allouées (la prise de temps s'effectue à chaque fois que l'on traite un nouveau bloc-colonne);
- `#define ALLOC_COEFF` [**false**] : cette option doit être activée pour initialiser la matrice lorsque l'on souhaite court-circuiter la phase d'assemblage. Par exemple, pour une matrice pleine, le résultat est la construction du système de taille N suivant :

$$\begin{pmatrix} \alpha & & & \\ 1 & \alpha & & \\ \multicolumn{4}{c}{\dots\dots\dots\dots} \\ 1 & 1 & \dots & \alpha \end{pmatrix} \cdot X = \begin{bmatrix} N-1+\alpha \\ N-1+\alpha \\ \dots \\ N-1+\alpha \end{bmatrix}$$

dont on sait que la solution est $X = 1$ (si $\alpha \geq N$).

Les options liées au calibrage des routines BLAS

- `#define LOG_PERF` [**false**] : cette option génère une prise de temps pour chaque appel à une routine BLAS;
- `#define TIME_BLAS` [**false**] : cette option affiche à la fin de la factorisation le pourcentage relatif du temps cumulé passé dans chacune des routines BLAS.

Les dépendances automatiques entre options

```
/* Flag dependencies */
/* LOG_PERF  needs FLAG_ASSERT  */
#if (defined LOG_PERF) && (!(defined FLAG_ASSERT))
#define FLAG_ASSERT
#endif
/* SAVE_TRSM needs COMPACT_CTRB */
#if (defined SAVE_TRSM) && (!(defined COMPACT_CTRB))
#define COMPACT_CTRB
#endif
/* TIME_BLAS needs LOG_PERF     */
#if (defined TIME_BLAS) && (!(defined LOG_PERF))
#define LOG_PERF
#endif
/* FANIN_SMX needs ALLOC_FTGT   */
#if (defined FANIN_SMX) && (!(defined ALLOC_FTGT))
#define ALLOC_FTGT
#endif
```

Quelques macros utiles

Suivant l'architecture, on définit 3 macros pour manipuler le compteur de temps (dans le cas où on génère des traces PICL, il est nécessaire d'initialiser le compteur de temps à 1 seconde à cause d'un "bug" dans la routine de tri des traces Paragraph).

```
#if (defined X_ARCHpower2_ibm_aix4) || (defined X_ARCHpowerPC_ibm_aix4)
static int t1[2];
static int t2[2];
#define SOPALIN_CLOCK_INIT {rtc(t1);}
#define SOPALIN_CLOCK_STOP {rtc(t2);}
#define SOPALIN_CLOCK_GET  ((double)(t2[0]-t1[0]))+((double)(t2[1]-t1[1]))/1000000000.
#else /*(defined X_ARCHpower2_ibm_aix4) || (defined X_ARCHpowerPC_ibm_aix4)*/
static Clock sop_clk;
#define SOPALIN_CLOCK_INIT {clockInit(&sop_clk);clockStart(&sop_clk);}
```

```
#define SOPALIN_CLOCK_STOP {clockStop(&sop_clk);}
#define SOPALIN_CLOCK_GET  clockVal(&sop_clk)
#endif /*(defined X_ARCHpower2_ibm_aix4) || (defined X_ARCHpowerPC_ibm_aix4)*/
```

Tous les accès aux structures de données sont encapsulés à l'aide des macros suivantes, le but étant de pouvoir intégrer rapidement tous les changements sur les structures de données et d'obtenir une meilleur lisibilité du code.

```
#define SYMB_CBLKTAB      datacode->symbmtx.cblktab
#define SYMB_BLOKTAB      datacode->symbmtx.bloktab
#define SYMB_CBLKNBR      datacode->symbmtx.cblknbr
#define SYMB_BLOKNBR      datacode->symbmtx.bloknbr
#define SYMB_NODENBR      datacode->symbmtx.nodenbr
#define SYMB_BLOKNUM(x)   datacode->symbmtx.cblktab[x].bloknum
#define SYMB_FCOLNUM(x)   datacode->symbmtx.cblktab[x].fcolnum
#define SYMB_LCOLNUM(x)   datacode->symbmtx.cblktab[x].lcolnum
#define SYMB_FROWNUM(x)   datacode->symbmtx.bloktab[x].frownum
#define SYMB_LROWNUM(x)   datacode->symbmtx.bloktab[x].lrownum
#define SYMB_CBLKNUM(x)   datacode->symbmtx.bloktab[x].cblknum /*<0 if remote*/
#define SOLV_SYMBMTX      datacode->symbmtx
#define SOLV_CBLKTAB      datacode->cblktab
#define SOLV_BLOKTAB      datacode->bloktab
#define SOLV_COEFTAB      datacode->coeftab
#define SOLV_FTGTTAB      datacode->ftgttab
#define SOLV_PROCNUM      datacode->procnum
#define SOLV_PROCNBR      datacode->procnbr
#define SOLV_COEFNBR      datacode->coefnbr
#define SOLV_FTGTNBR      datacode->ftgtnbr
#define SOLV_BPFTMAX      datacode->bpftmax /* max blocks for greatest ftgt  */
#define SOLV_CPFTMAX      datacode->cpftmax /* max elements for greatest ftgt*/
#define SOLV_COEFMAX      datacode->coefmax /* max elements for greatest cblk*/
#define SOLV_CTRBNBR(x)   datacode->cblktab[x].ctrbnbr
#define SOLV_CTRBCNT(x)   datacode->cblktab[x].ctrbcnt
#define SOLV_PRIONUM(x)   datacode->cblktab[x].prionum
#define SOLV_STRIDE(x)    datacode->cblktab[x].stride
#define SOLV_COLOR(x)     datacode->cblktab[x].color
#define SOLV_COEFIND(x)   datacode->bloktab[x].coefind
#define SOLV_FTGTIND(x)   datacode->bloktab[x].ftgtind /*<0 if local*/
#define FANIN_INFOTAB(x)  datacode->ftgttab[x].infotab
#define FANIN_INDTAB(x)   datacode->ftgttab[x].indtab
#define FANIN_ROWTAB(x)   datacode->ftgttab[x].rowtab
#define FANIN_COEFTAB(x)  datacode->ftgttab[x].coeftab
#define FANIN_CTRBNBR(x)  datacode->ftgttab[x].infotab[FTGT_CTRBNBR]
#define FANIN_CTRBCNT(x)  datacode->ftgttab[x].infotab[FTGT_CTRBCNT]
#define FANIN_PROCDST(x)  datacode->ftgttab[x].infotab[FTGT_PROCDST]
#define FANIN_CBLKDST(x)  datacode->ftgttab[x].infotab[FTGT_CBLKDST]
```

```
#define FANIN_PRIONUM(x)  datacode->ftgttab[x].infotab[FTGT_PRIONUM]
#define FANIN_FCOLNUM(x)  datacode->ftgttab[x].infotab[FTGT_FCOLNUM]
#define FANIN_LCOLNUM(x)  datacode->ftgttab[x].infotab[FTGT_LCOLNUM]
#define FANIN_BLOKNBR(x)  datacode->ftgttab[x].infotab[FTGT_BLOKNBR]
#ifdef SOLVER_UPDOWN
#define UPDOWN_SM2XTAB    datacode->updovct.sm2xtab
#define UPDOWN_SM2XMAX    datacode->updovct.sm2xmax
#define UPDOWN_SM2XIND(x) datacode->updovct.cblktab[x].sm2xind
#define UPDOWN_CTRBCNT(x) datacode->updovct.cblktab[x].ctrbcnt
#define UPDOWN_BROWNBR(x) datacode->updovct.cblktab[x].brownbr
#define UPDOWN_BROWCBLKTAB(x) datacode->updovct.cblktab[x].browcblktab
#define UPDOWN_BROWPROCTAB(x) datacode->updovct.cblktab[x].browproctab
#endif
```

9. Les structures de données

Dans cette partie nous allons présenter les différentes structures de données manipulées par les modules du projet. Les définitions de ces structures sont maintenant figées pour la distribution 1D, et évoluent assez facilement pour la distribution 2D.

9.1. La `Symbol Matrix`

Il s'agit de la structure de données partagée par l'ensemble des modules du projet. Elle contient l'ensemble des informations nécessaires à la description géométrique de la matrice creuse à factoriser. Sa définition est suffisamment générique pour représenter soit la matrice globale, soit la partie locale à un processeur.

Elle est constituée de deux tableaux; le premier contient la description des blocs colonnes et pointe sur les éléments du second tableau contenant la description des blocs diagonaux ou extra-diagonaux (voir figure 3).

La `Symbol Matrix` est construite une première fois par le module `FAX` et représente la matrice globale après la renumérotation. Elle est ensuite reconstruite par le module `BLEND` et représente la matrice locale à un processeur après l'étape de repartitionnement et distribution. Cette structure est ensuite manipulée par les modules `PABLAS` et `SOPALIN`.

```
/*+ The column block structure. +*/
typedef struct SymbolCblk_ {
  INT  fcolnum;  /*+ First column index              +*/
  INT  lcolnum;  /*+ Last column index (inclusive)    +*/
  INT  bloknum;  /*+ First block in column (diagonal) +*/
} SymbolCblk;

/*+ The block structure. +*/
typedef struct SymbolBlok_ {
  INT  frownum;  /*+ First row index            +*/
  INT  lrownum;  /*+ Last row index (inclusive) +*/
  INT  cblknum;  /*+ Facing column block        +*/
} SymbolBlok;

/*+ The symbolic block matrix. +*/
typedef struct SymbolMatrix_ {
  INT         baseval;  /*+ Base value for numberings         +*/
  INT         cblknbr;  /*+ Number of column blocks           +*/
  INT         bloknbr;  /*+ Number of blocks                  +*/
  SymbolCblk *cblktab;  /*+ Array of column blocks [+1,based] +*/
  SymbolBlok *bloktab;  /*+ Array of blocks [based]           +*/
  INT         nodenbr;  /*+ Number of nodes in matrix         +*/
} SymbolMatrix;
```

9.2. La Solver Matrix

Il s'agit d'un enrichissement de la structure de donnée précédente; elle contient des informations supplémentaires nécessaires à la factorisation. En particulier, elle caractérise les blocs et les blocs colonnes, associe à chaque bloc une `Fan-In Target` si nécessaire, et contient le tableau des coefficients de la matrice. Sa définition est suffisamment générique pour représenter soit la matrice globale soit la partie locale à un processeur.

Elle est constituée de deux tableaux qui enrichissent ceux de la `Symbol Matrix`; en particulier, on dispose d'un compteur maintenant le nombre de contributions à appliquer avant de pouvoir commencer à traiter un bloc colonne. De plus, chaque bloc est caractérisé par une entrée dans le tableau des `Fan-In Targets` et une entrée dans le tableau des coefficients (voir figure 3). Les coefficients sont stockés continuement par colonne compléte dans un bloc colonne donné, ceci afin d'optimiser les traitements des blocs à l'aide des routines BLAS 3.

La `Solver Matrix` est construite par le module `BLEND`; le module `PABLAS` alloue et remplit ensuite le tableau des coefficients résultant de l'assemblage. Cette structure est ensuite manipulée par le module `SOPALIN`.

```
/*+ Solver column block structure. +*/
typedef struct SolverCblk_  {
  INT  ctrbnbr;  /*+ Number of contributions           +*/
  INT  ctrbcnt;  /*+ Number of remaining contributions  +*/
  INT  prionum;  /*+ Column block priority              +*/
  INT  stride;   /*+ Column block stride                +*/
#ifdef SOLVER_DEBUG
  INT  color;    /*+ Color of column block (PICL trace) +*/
#endif
} SolverCblk;

/*+ Solver block structure. +*/
typedef struct SolverBlok_ {
  INT  coefind;  /*+ Index in coeftab +*/
  INT  ftgtind;  /*+ Index in ftgttab +*/
} SolverBlok;

/*+ Solver matrix structure. +*/
typedef struct SolverMatrix_ {
  SymbolMatrix symbmtx;  /*+ Symbolic matrix                              +*/
#ifdef SOLVER_UPDOWN
  UpDownVector updovct;  /*+ UpDown vector                                +*/
#endif
  SolverCblk  *cblktab;  /*+ Array of solver column blocks                +*/
  SolverBlok  *bloktab;  /*+ Array of solver blocks                       +*/
  INT          coefnbr;  /*+ Number of coefficients                       +*/
  INT          ftgtnbr;  /*+ Number of fanintargets                       +*/
```

```
  FLOAT       *coeftab;  /*+ Coefficients access vector             +*/
  FanInTarget *ftgttab;  /*+ Fanintarget access vector              +*/
  INT          coefmax;  /*+ Working block max size                 +*/
  INT          bpftmax;  /*+ Maximum of blocks per fanintarget      +*/
  INT          cpftmax;  /*+ Maximum of coefficients per fanintarget +*/
  int          procnum;  /*+ current processor number               +*/
  int          procnbr;  /*+ number of processors                   +*/
} SolverMatrix;
```

9.3. La Fan-In Target

Une Fan-In Target permet de cumuler localement les contributions destinées à un même bloc colonne distant. Elle est caractérisée par le bloc recouvrant minimal du bloc colonne cible; il s'agit donc d'une liste de blocs. On dispose d'un compteur maintenant le nombre de contributions à appliquer avant de pouvoir envoyer la Fan-In Target (voir figure 3).

Les Fan-In Targets sont initialisées par le module BLEND. Ces structures sont ensuite manipulées par le module SOPALIN qui allouera dynamiquement le vecteur de coefficients associé.

```
/*+ Fanintarget info type +*/
typedef enum {
  FTGT_CTRBNBR = 0, /*+ Number of contributions           +*/
  FTGT_CTRBCNT,     /*+ Number of contributions remaining +*/
  FTGT_PROCDST,     /*+ Destination for fanintarget       +*/
  FTGT_CBLKDST,     /*+ Column block destination          +*/
  FTGT_PRIONUM,     /*+ Fanintarget priority              +*/
  FTGT_FCOLNUM,     /*+ Fanintarget first column          +*/
  FTGT_LCOLNUM,     /*+ Fanintarget last column           +*/
  FTGT_BLOKNBR,     /*+ Number of blocks in fanintarget   +*/
  MAXINFO
} FanInInfo;

/*+ Fanintarget structure +*/
typedef struct FanInTarget_ {
  INT    infotab[MAXINFO];  /*+ Fanintarget descriptor            +*/
  INT   *indtab;            /*+ Block index in fanintarget vector  +*/
  INT   *rowtab;            /*+ Block first row                    +*/
  FLOAT *coeftab;           /*+ Fanintarget vector access          +*/
} FanInTarget;
```

9.4. Le `Vector UpDown`

Il s'agit du vecteur contenant le second-membre. Son extension pour une résolution multi-second-membres est très facile. Le processeur détenant le bloc diagonal d'un bloc colonne détient également la partie correspondante du vecteur second-membre.

Le `Vector Updown` est alloué et rempli par le module `PABLAS` lors de l'assemblage du second membre. Cette structure est ensuite manipulée par le module `SOPALIN` lors de l'étape de "descente/remontée".

```
/*+ UpDown block structure. +*/
typedef struct UpDownCblk_  {
  INT  sm2xind;             /*+ Index in unknown vector           +*/
  int *browcblktab;         /*+ Brow for column block             +*/
  int *browproctab;         /*+ Brow for column block             +*/
  int  brownbr;             /*+ Brow size                         +*/
  int *bcoltab;             /*+ Bcol for column block             +*/
  int  bcolnbr;             /*+ Bcol size                         +*/
  INT  ctrbcnt;             /*+ Number of contributions           +*/
} UpDownCblk;

/*+ UpDown vector structure. +*/
typedef struct UpDownVector_ {
  UpDownCblk *cblktab;      /*+ Array of solver column blocks       +*/
  FLOAT      *sm2xtab;      /*+ Unknown vector                      +*/
  INT         sm2xmax;      /*+ Maximum of coeffs per unknown vector +*/
} UpDownVector;
```

9.5. La `DistribElts`

Il s'agit d'une structure de données manipulée par le module `SACHA`; elle contient les éléments attribués à un processeur et des informations sur la gestion de la communication lors de l'assemblage parallèle.

```
typedef struct DistribElts_
{
  INT elocnbr;  /*+ Number of local elements +*/
  INT nonlnbr;  /*+ Maximum number of non local nodes for a local element +*/
  INT nelonbr;  /*+ Number of local elements who are not local at all +*/
  INT smsgnbr;  /*+ Number of messages the local processor will send +*/
  INT rmsgnbr;  /*+ Number of messages the local processor will receive +*/
  INT *eloctab; /*+ Local elements vector +*/
  INT *elgotab; /*+ Local elements geometry vector +*/
  INT *elddtab; /*+ Number of DDL for every non local element +*/
```

```
  INT *elintab; /** Indexes for elnltab, local element number **/
  INT *elnltab; /** Informations about non local elements and nodes **/
  INT *elostab; /** Indexes for Fortran routines **/
} DistribElts;
```

9.6. La structure Assembly1D

Il s'agit d'une structure de données manipulée par le module PABLAS; elle permet de connaitre la distribution des nœuds du maillage sur les différents processeurs.

```
typedef struct Assembly1D_ {
  int * blprtab; /** Array of block to processor mapping **/
  INT * nocbtab; /** Array of node to column block (1D) mapping **/
  INT * rnumtab; /** Absolute number to relative number tab **/
} Assembly1D;
```

9.7. Les relations entre les structures de données.

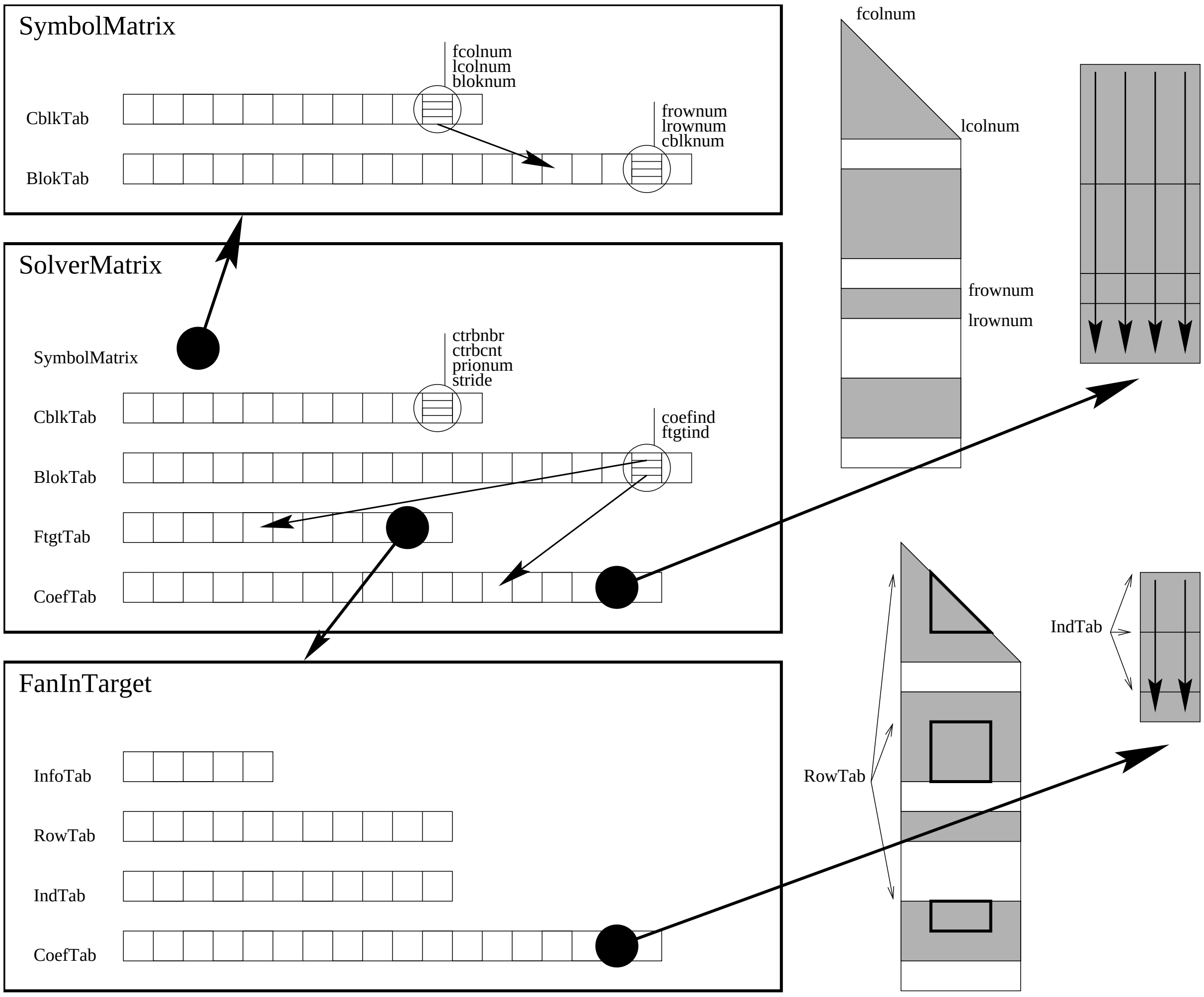

Figure 3. Relations entre les structures de données.

Bibliographie

[1] D. Goudin. Pré-étude de la parallélisation du code OSSAU de mécanique des structures – Mémoire de DEA. Technical report, CEA/CESTA, July 1997.

[2] D. Goudin. Mise en œuvre d'une bibliothèque d'outils pour la résolution par méthode directe de grands systèmes linéaires creux symétriques définis positifs sur machine parallèle – Rapport semestriel première partie. Technical report, CEA/CESTA, March 1998.

[3] D. Goudin. Mise en œuvre d'une bibliothèque d'outils pour la résolution par méthode directe de grands systèmes linéaires creux symétriques définis positifs sur machine parallèle – Rapport Final Première Partie. Technical report, CEA/CESTA, February 1999.

[4] D. Goudin. Mise en œuvre d'une bibliothèque d'outils pour la résolution par méthode directe de grands systèmes linéaires creux symétriques définis positifs sur machine parallèle – Rapport semestriel seconde partie. Technical report, CEA/CESTA, December 1999.

[5] D. Goudin. *Mise en œuvre d'une Bibliothèque d'Outils pour la Résolution Parallèle Hautes Performances par Méthode Directe de Grands Systèmes Linéaires Creux et application à un code de Mécanique des Structures.* Thèse de doctorat, Université Bordeaux I, soutenue le 30 novembre 2000.