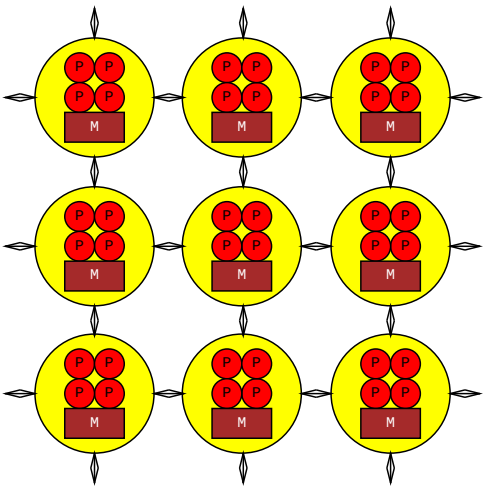
Formation d'Ingénieurs de l'Institut Galilée

Université Paris 13 MACS 2^{eme} année

Parallélisation des applications numériques
(notes de cours)



Ph. d'Anfray

Mars 2002 *(maj. 2004)* *Philippe.d-Anfray@renater.fr*

Remerciements :
…aux collègues qui ont fourni de nombreuses idées pour ce cours et notamment Francois-Xavier Roux pour l'art du découpage, quelques points de Gauss…et de gradient ; Charles Roge pour de nombreux points de gradient !

Table des matières

Table des figures

Chapitre 1

Généralités

Ces notes de cours traitent de la parallélisation des applications numériques. Il est clair que dans ce domaine, il n'y a pas de méthode universelle et chaque cas nécessitera un traitement spécifique. Le but de ces notes est de faire un tour d'horizon des techniques mises en œuvre pour permettre d'analyser au mieux un nouveau problème de parallélisation.

Bilan très rapide des cours précédents : "sur quoi peut-on travailler ?"

1. Sur la modélisation du problème c'est à dire les aspects physiques et sur sa mise en équations, aspects mathématiques.
2. Sur les méthodes numériques utilisées : schémas, solveurs etc...
3. Sur le code source lui-même : vectorisation, parallélisation par directives...

L'ordre dans lequel ces trois façons d'aborder la parallélisation sont présentées n'est pas du au hasard. Seul les points `1)` et `2)` permettent d'espérer obtenir des résultats appréciables et qui "passent à l'échelle", c'est à dire qui pouront s'adapter à une machine massivement parallèle comportant un très grand nombre de processeurs. Nous réserverons le point `3)` à l'optimisation locale du code sur le processeur (vectoriel ou super scalaire) et sur le nœud de calcul (souvent multiprocesseur à mémoire partagée).

Un approche générale et relativement simple à mettre en œuvre, surtout si l'on envisage encore une fois le "passage à l"échelle", nous est donnée par le modèle `SPMD`, pour "*Single Program Multiple Data*", où le même exécutable traîte sur chaque nœud de la machine parallèle une partie des données du problème.

Bien sûr ces différents exécutables doivent se synchroniser et communiquer entre-eux, ce qui est assuré en pratique par l'utilisation d'une bibliothèque qui permet la mise en œuvre du modèle de programmation par échanges de messages, le plus souvent `MPI`[1] ou encore `PVM`[2].

[1] "Message Passing Interface"
[2] "Parallel Virtual Machine"

Il faut alors répartir les données du problème et la première idée est de découper le domaine de calcul. Nous introduisons de la sorte un coût de gestion du parallélisme proportionnel à la taille des “interfaces”, c’est à dire des nouvelles frontières ainsi introduites entre les sous domaines.

1.1 L’art du découpage

Il y a bien un art du découpage. Le découpage idéal permet de répartir harmonieusement la charge de calcul sur les processeurs pour minimiser le délai total d’exécution. Notons au passage que les critères retenus pour déterminer ce découpage peuvent être plus complexes que simplement le nombre de nœuds du sous domaine (par exemple la quantité de calculs à effectuer pour chaque élément du maillage peut différer d’une région à une autre, etc...).

Il doit en outre minimiser le surcoût engendré par ce découpage c’est à dire le volume des communications, relié directement à la taille des “interfaces” entre sous domaines.

S’il est relativement facile de découper un maillage structuré de type grille, il est en revanche beaucoup plus délicat de partitionner les maillages non structurés utilisés en éléments finis.

Les outils de CAO qui permettent de construire les maillages “industriels” n’incluent pas -le plus souvent- les notions nécessaires à un prétraitement du maillage compatible avec une approche parallèle. Il faut alors développer des outils complexes les “découpeurs de maillages” pour mener à bien cette tâche comme un prétraitement du calcul proprement dit.

1.2 Un exemple

Prenons un exemple simple sur un domaine cubique maillé de façon régulière ($n \times n \times n$ points) que nous découpons en parties “égales” destinées à être traitées par p processeurs.

Nous allons passer en revue plusieurs découpage possible et à chaque fois évaluer la taille des “interfaces”, c’est à dire le nombre de points “frontière” dont le traitement engendrera des communications et donc un surcoût lié à la parallélisation.

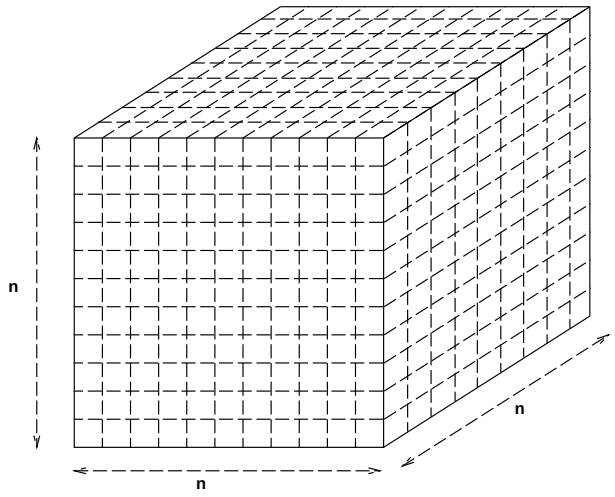


FIG. 1.1: *Un domaine cubique n x n x n*

1.2.1 Une 1re idée

L'idée la plus simple consiste à diviser le domaine en p tranches.

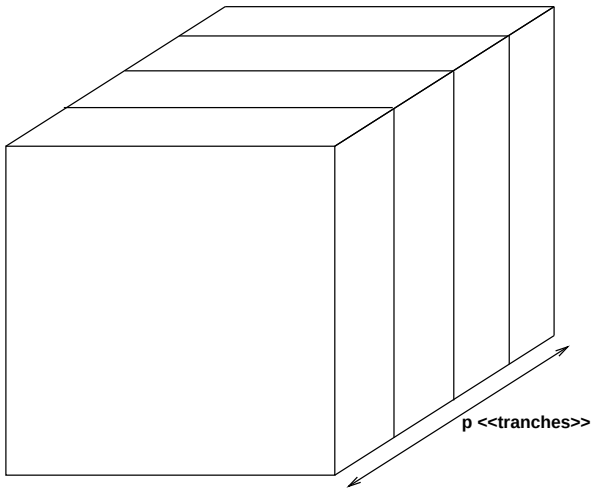


FIG. 1.2: *Domaine divisé en p "tranches"*

Évaluons (grossièrement) le nombre de points frontière :
- chaque tranche possède deux frontières de n^2 points ;
- chaque point frontière appartient à deux tranches ;

– il y a p tranches.

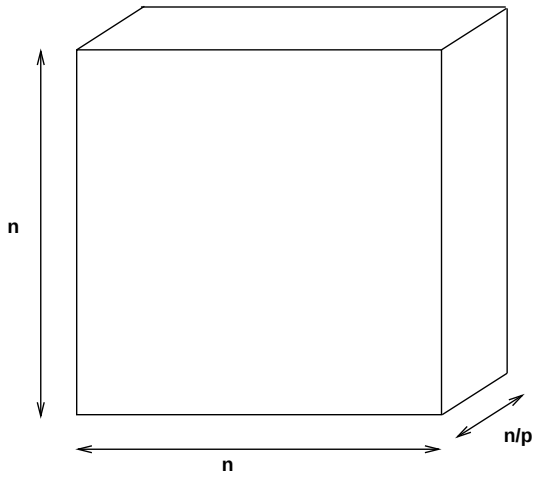


FIG. 1.3: *Une "tranche" du domaine*

Nous arrivons à un total de :

$$N_f = \frac{2pn^2}{2} = pn^2 \quad \text{sur un total de } N_T = n^3 \text{ points.}$$

C'est à dire une proportion de points frontière :

$$\frac{N_f}{N_T} = \frac{p}{n}$$

Application numérique :
Prenons $n = 64$: le maillage possède $64^3 = 262144$ points. Si nous utilisons $p = 64$ processeurs, nous obtenons $N_f/N_T = 1$, tous les points sont "frontière", c'est l'échec total.

1.2.2 Une 2^{e}idée

Essayons de faire mieux en découpant le domaines en "crayons" c'est à dire en utilisant deux dimensions d'espace au lieu d'une.

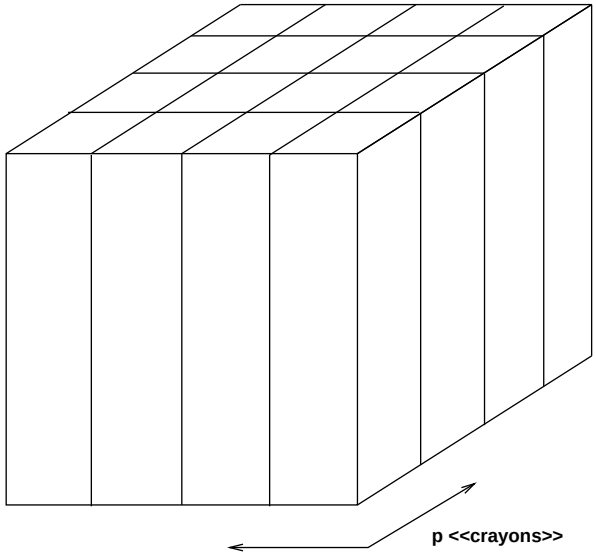


FIG. 1.4: *Domaine divisé en p "crayons"*

De même, évaluons (grossièrement) le nombre de points frontière :
– chaque tranche possède quatre frontières de $n^2/\sqrt{p}$ points ;
– chaque point frontière appartient à deux crayons ;
– il y a p crayons.

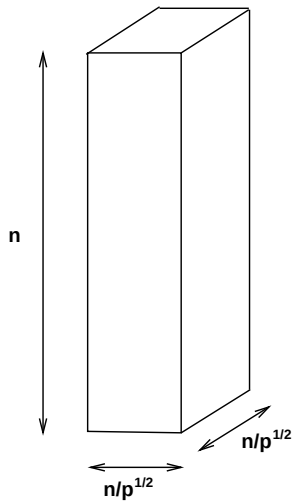


FIG. 1.5: *Un "crayon" du domaine*

Nous arrivons à un total de :

$$N_f = \frac{4pn^2}{2p^{1/2}} = 2n^2p^{1/2} \quad \text{sur un total de } N_T = n^3 \text{ points.}$$

C'est à dire une proportion de points frontière :

$$\frac{N_f}{N_T} = \frac{2p^{1/2}}{n}$$

Application numérique :
Prenons $n = 64$, donc $64^3 = 262144$ points ; si nous utilisons $p = 64$ processeurs, nous avons cette fois, $N_f/N_T \simeq 0.25$, soit une proportion qui semble plus raisonnable de points frontière.

1.2.3 Et enfin une 3^{e}idée

Encore mieux ! en découpant le domaines en "cubes", selon les trois directions d'espace.

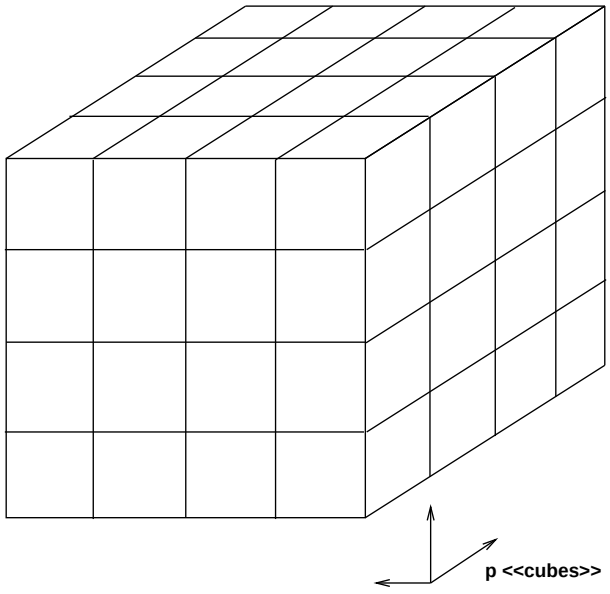


FIG. 1.6: *Domaine divisé en p "cubes"*

De même, évaluons (grossièrement) le nombre de points frontière :
– chaque tranche possède six frontières de $n^2/(\sqrt[3]{p})^2$ points ;

– chaque point frontière appartient à deux cubes ;
– il y a p cubes.

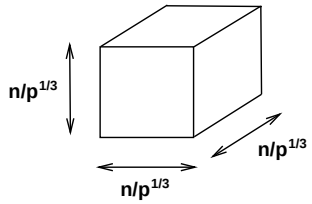


FIG. 1.7: *Un “cube” du domaine*

Nous arrivons à un total de :

$$N_f = \frac{6pn^2}{2p^{2/3}} = 3n^2p^{1/3} \quad \text{sur un total de } N_T = n^3 \text{ points.}$$

C’est à dire une proportion de points frontière :

$$\frac{N_f}{N_T} = \frac{3p^{1/3}}{n}$$

Application numérique :
Prenons $n = 64$, soit $64^3 = 262144$ points, si nous utilisons $p = 64$ processeurs. Cette fois nous obtenons $N_f/N_T \simeq 0.18$, soit une proportion de points frontière qui peut paraître tout à fait acceptable.

1.3 En conclusion

Manque de chance ! d’après ce qui précède, ce n’est pas le découpage le plus simple qui sera optimal et les communications entre processeurs seront d’autant plus difficiles à gérer qu’il y aura de sous domaines “voisins”.

Dans le dernier cas de l’exemple précédent (cf 1.2.3) un sous-domaine situé à l’intérieur du grand cube “communique” potentiellement avec six voisins. Une consolation néanmoins, une fois le “patron des communications” entre processeurs établi, il reste figé pour le reste du calcul[3] et pourra donc être optimisé.

De plus, en découpant selon plusieurs directions, chaque processeur communiquera avec plusieurs autres. À l’exécution, l’application utilisera au mieux les ressources matérielles de la machine parallèle (réseau de communication, etc...).

[3] pourvu que l’on ne remaille pas...

Enfin, il ne faut pas toutefois pas oublier de prendre en compte les caractéristiques géométriques du domaine. Ainsi le domaine ci-dessous, très allongé, peut-il être découpé simplement en “tranches” de façon très efficace, les “interfaces” restant de petite taille.

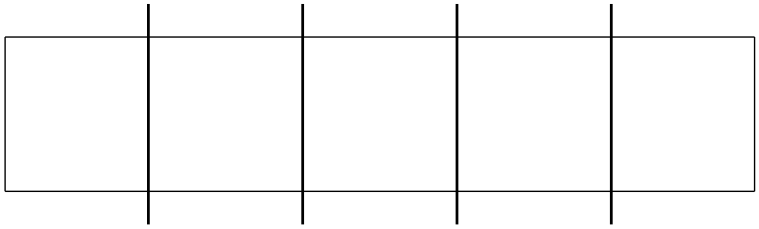


FIG. 1.8: *Découpage selon une direction*

Chapitre 2

Contexte "différences finies"

2.1 Généralités

Pour résoudre un problème dans ce contexte nous considèrons un maillage régulier, défini par une origine et un *pas de grille* dans chaque direction d'espace. Il n'y a pas lieu de stocker le maillage. Pour paralléliser, il faut s'attacher simplement à partitionner les inconnues du problème.

2.2 Un problème modèle

Nous cherchons à résoudre l'équation de Laplace, sur un domaine rectangulaire Ω de bord Γ.

$$\begin{cases} \Delta u = 0 & sur\, \Omega \\ u = g & sur\, \Gamma \end{cases}$$

Cette équation intervient dans de multiples problèmes de la physique.
Nous utilisons un schéma de différences finies centré à cinq points pour discrétiser le laplacien Δu :

$$\Delta u = \frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2}$$

Le schéma nous donne la façon d'approximer les dérivées sur la grille en fonction des valeurs des inconnues au nœuds. Par exemple en supposant que les pas (h) sont identiques en x et y :

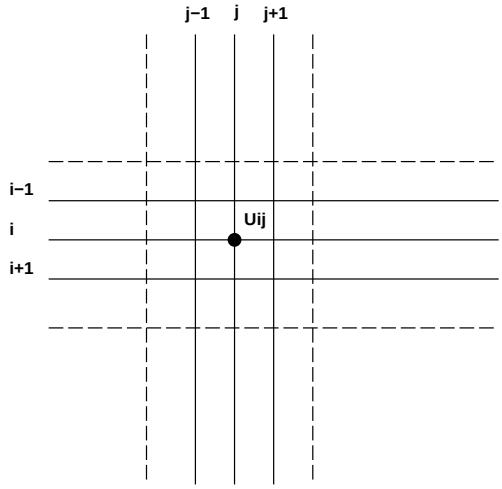


FIG. 2.1: *Discrétisation du laplacien*

Les dérivées en x à gauche et à droite au point $u_{(i,j)}$ sont approximées par :

$$\left(\frac{\delta u}{\delta x}\right)_g = \frac{u_{(i,j)} - u_{(i-1,j)}}{h} \quad \text{et} \quad \left(\frac{\delta u}{\delta x}\right)_d = \frac{u_{(i+1,j)} - u_{(i,j)}}{h}$$

et la dérivée seconde par le saut de la dérivée première en $u_{(i,j)}$ donc :

$$\frac{\delta^2 u}{\delta x^2} = \frac{u_{(i+1,j)} + u_{(i-1,j)} - 2u_{(i,j)}}{h^2}$$

en faisant de même pour la dérivée en y nous obtenons :

$$\Delta u = \frac{u_{(i+1,j)} + u_{(i-1,j)} + u_{(i,j+1)} + u_{(i,j-1)} - 4u_{(i,j)}}{h^2}$$

Une meilleure approximation serait obtenue en utilisant, par exemple, un schéma à 9 points etc...

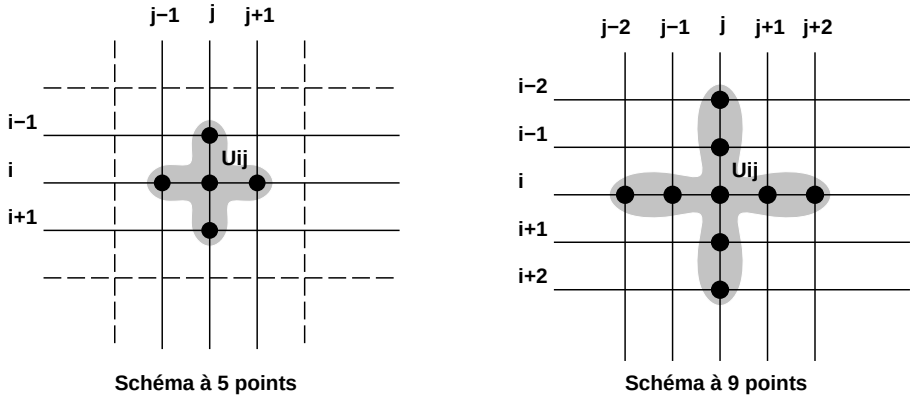


FIG. 2.2: *Différents schémas*

Nous voyons apparaître ici la taille du voisinage nécessaire au calcul en un point. Cette notion sera importante, par la suite, lors de la parallélisation.

Prenons des constantes commes conditions aux limites pour le problème global. Par exemple, en fixant des valeurs sur les différents cotés du domaine *haut*, *bas*, *droit* et *gauche*.

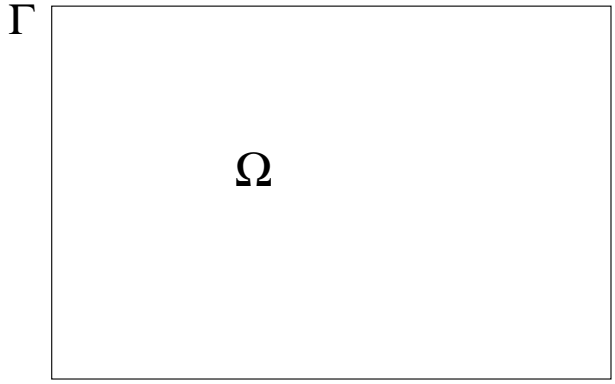


FIG. 2.3: *Le domaine de calcul*

Le domaine est maillé en utilisant une grille régulière et l'inconnue u est discrétisée sur les nœuds de la grille. Si nous écrivons le tableau des valeurs de la variable u, celles-ci sont connues sur les bords. Si le nombre des inconnues est `dim_1 x dim_2`, la dimension totale de la matrice sera `dim_g1 x dim_g2` où `dim_g`*i*`=dim_`*i*`+2`.

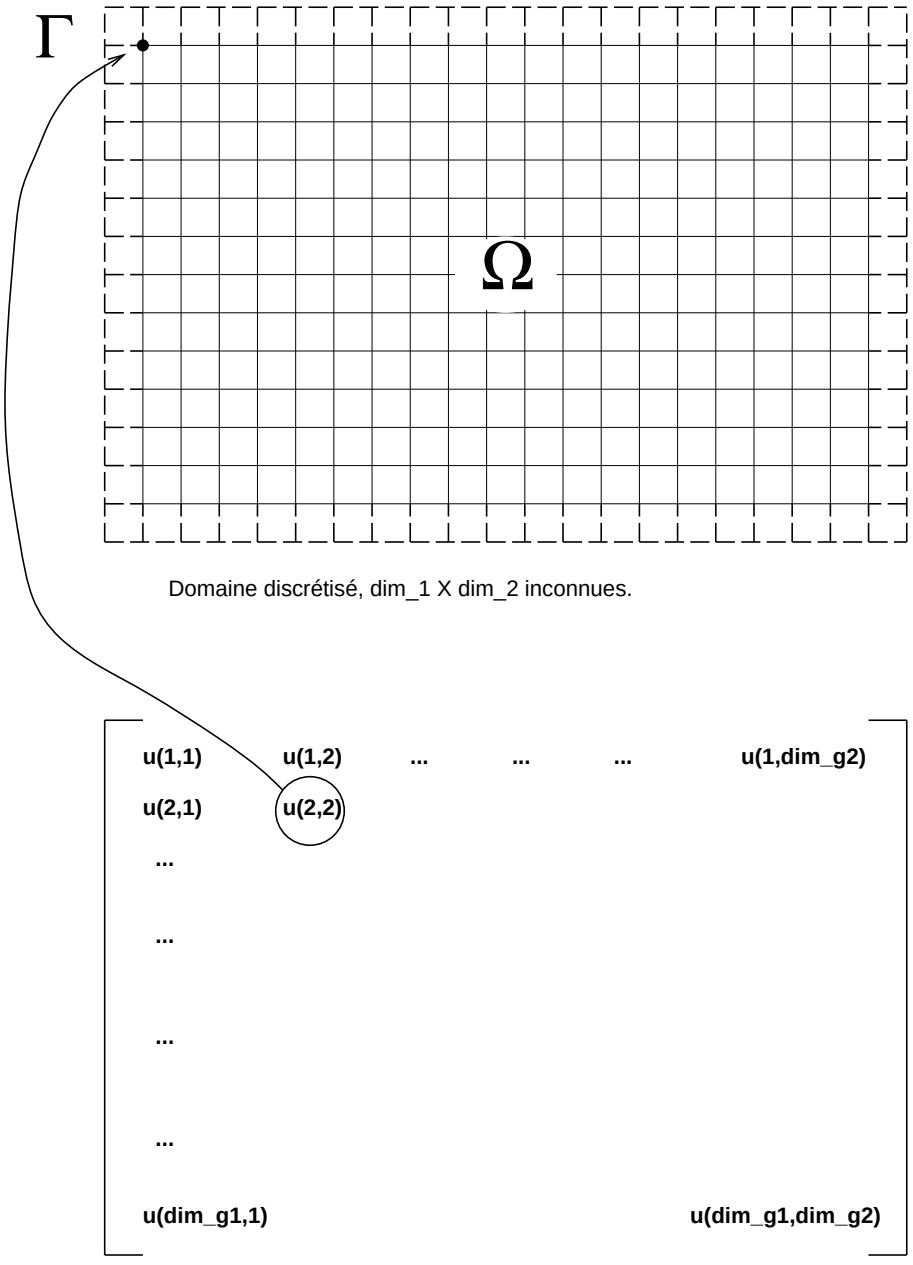


FIG. 2.4: *Les inconnues u[i,j]*

Enfin, pour la résolution nous utiliserons, ici, une méthode explicite aussi simple que possible (...) c'est à dire peu efficace mais cela n'enlève rien à la

généralité de l'approche.
Par exemple, l'algorithme (de Gauss) suivant :

```
{
// initialisation
//
u[i,j]=0             // au centre
u[i,j]=valeur_fixee // sur les bords
//
// boucle de calcul
//
jusqu'a convergence
   {
   residu=0
   pour i=2, dimg1-1 faire
      pour j=2, dimg2-1 faire
         {
         u_tmp =u (i,j)
         u[i,j]=(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1])/4.0
         //
         // residu: se méfier quand même si u[i,j]=0 ...
         //
         residu=max(residu, ||u[i,j]-u_tmp||/||u[i,j]||)
         }
   si (residu < epsilon) alors convergence
   }
//
// epilogue
//
ecrire_resultats (...)
}
```

FIG. 2.5: *Un algorithme "séquentiel" de résolution.*

2.3 Mise en œuvre de la parallélisation

Pour partitionner simplement le domaine nous choisirons la stratégie "domaine allongé" du paragraphe précédent (cf figure 1.8) : le découpage est effectué selon une seule direction. Dans chacune des "tranches" ainsi obtenues, nous résolvons de façon explicite, par le même algorithme, le problème local en allouant une "tranche" par processeur de calcul.

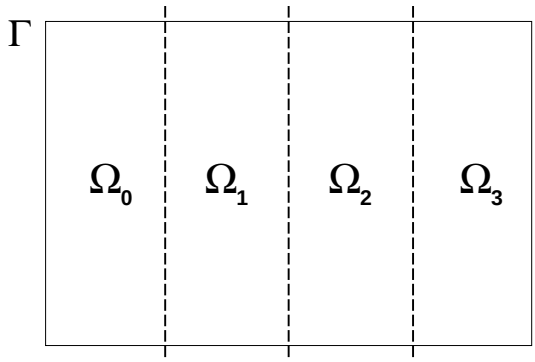


FIG. 2.6: *Découpage du domaine de calcul*

Pour chaque problème local, les valeurs sur les bords seront données selon les cas soit par les conditions aux limites du problème global soit par le recolement avec le sous-domaine voisin, ce qui assurera la continuité du domaine de calcul et donc de la solution.
Concrètement, pour les tableaux des inconnues locales, il y aura recouvrement entre “tranches” voisines. Ce recouvrement sera fonction du schémas utilisé pour discrétiser le laplacien. Avec le schéma à 5 points, il suffira de recouvrir d’une colonne ; il faudrait 2 colonnes si l’on utilisait le schéma à 9 points de la figure 2.2. C’est donc ici que la notion de voisinage intervient.

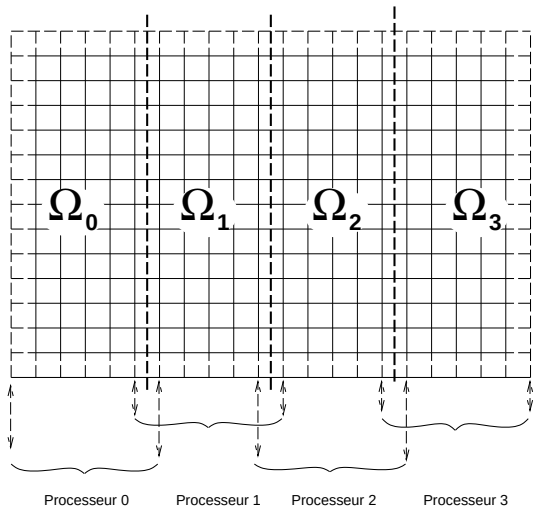


FIG. 2.7: *Recouvrement des domaines traités localement*

Pour bien comprendre ce qui se passe, nous pouvons “visualiser” la correspondance “globale-locale”, ici pour le processeur de calcul traitant le premier sous-domaine. De même que pour le problème global si chaque processeur traite `dim_1l x dim_2l` inconnues, il possède `diml_1 x diml_2` valeurs avec `dim_l`*i*`=dim_`*i*`l+2`.

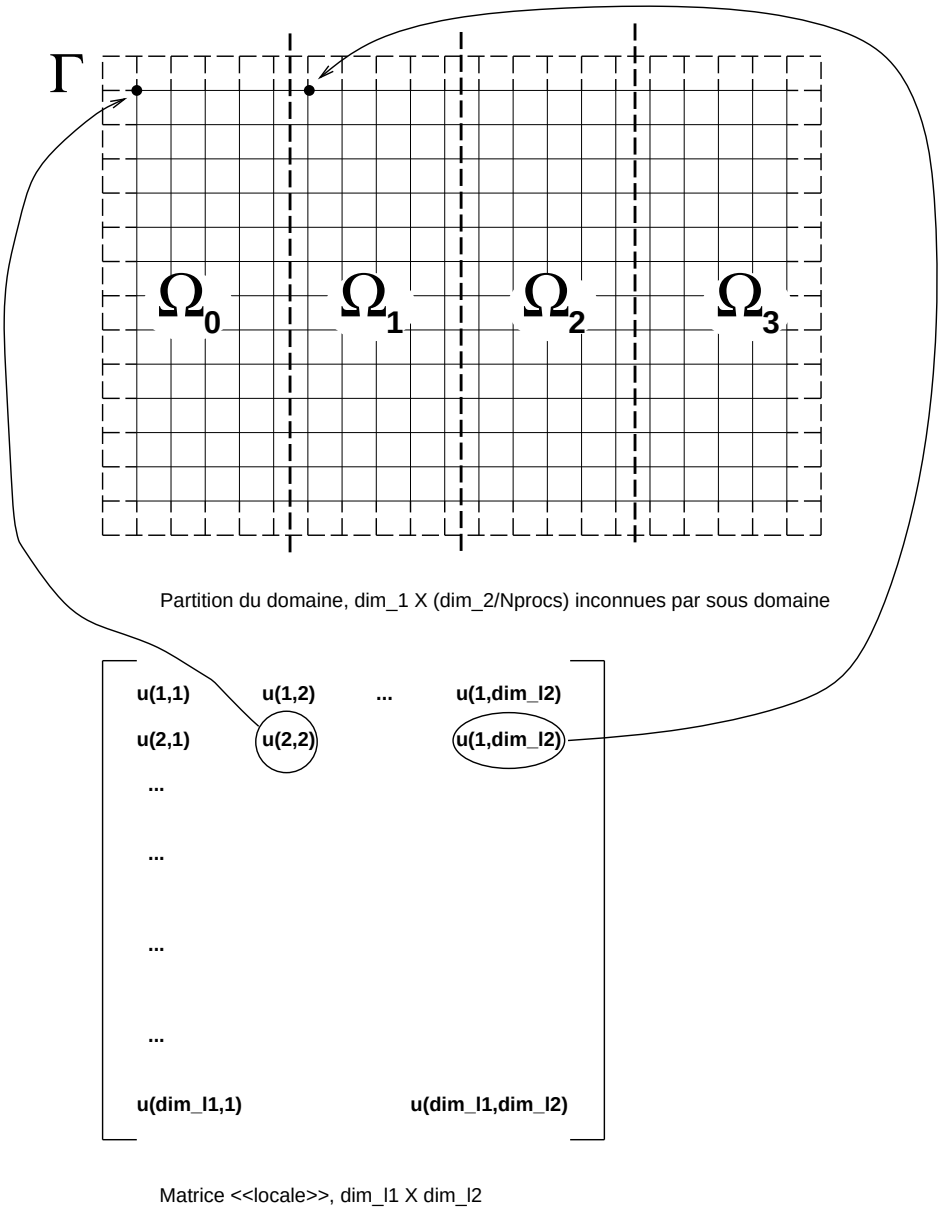


FIG. 2.8: *Domaine global, matrice locale*

À chaque itération de la méthode explicite, les calculs sont donc effectués en parallèle sur les sous-domaines. Il faut ensuite mettre à jour les données aux frontières des sous-domaines, dans les zones de recouvrement des inconnues locales.

Nous pouvons maintenant écrire l'algorithme, qui s'exécutera **dans chaque processeur** :

```
{
// initialisation
//
INITIALISATION_LOCALE (...)
//
// boucle de calcul
//
jusqu'a convergence
   {
   //
   MISE_A_JOUR [...]
   //
   residu=0
   pour i=2, diml1-1 faire
      pour j=2, diml2-1 faire
         {
         u_loc =u (i,j)
         u[i,j]=1/4(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1])
         //
         // residu: se méfier quand même si u[i,j]=0 ...
         //
         residu=max(residu, ||u[i,j]-u_loc||/||u[i,j]||)
         }
   //
   TEST_DE_CONVERGENCE (...)
   //
   }
//
// epilogue
//
RESULTATS (...)
}
```

FIG. 2.9: *L'algorithme de chaque processeur*

Cet algorithme local est très proche du code séquentiel qui résoud le même

problème. Les modifications “parallèles” sont indiquées en majuscules, elles concernent les quatre sous-programmes suivants :

- `INITIALISATION_LOCALE` : chaque processeur initialise sa “tranche” ;
- `MISE_A_JOUR` : les échanges aux frontières des sous domaines au début de chaque itération (sauf peut être la première) ;
- `TEST_DE_CONVERGENCE` : où l’on récupère, sur chaque nœud, le résidu global ;
- `RESULTATS` : l’écriture des résultats ne peut se faire par tous les processeurs simultanément.

Ces quatre sous-programmes donnent un bon aperçu des problèmes posés par la parallélisation d’un tel code. Pour bien les résoudre (indépendamment du nombre de processeurs), il est impératif de raisonner localement. Une fois “dans un processeur de calcul”, *que dois-je faire ?, qui m’envoie des informations ?, à qui dois-je en transmettre ?*. Pour réaliser ces sous-programmes nous disposons des fonctions “de base” de toute bibliothèque d’échanges de messages :

- identification : qui suis-je ? parmi combien ? ;
- communication “point à point” ;
- mais aussi opérations globales : synchronisation, réduction, etc... ? dont nous verront rapidement l’utilité.

2.3.1 Initialisation Locale

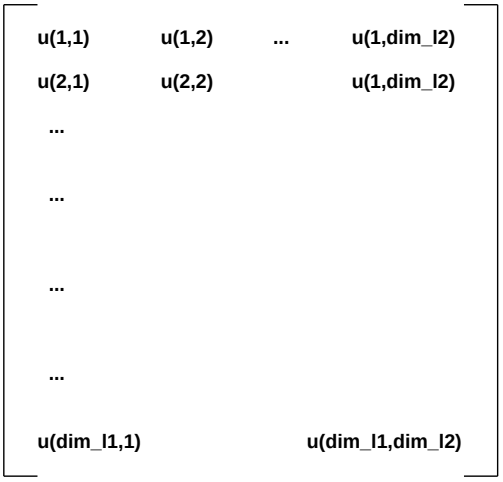
Dans chaque processeur, les inconnues du problème local sont initialisées à zéro. Puis l’on impose les conditions aux limites. Sur les bords haut et bas ce sont les mêmes pour tous les processeurs. Seul le processeur `0` qui possède le sous-domaine “de gauche” imposera les conditions du bord gauche et enfin le processeur `nb_procs-1` qui possède le sous-domaine “de droite” imposera les conditions du bord droit. Il est donc important que chaque processeur puisse s’identifier (qui suis-je ? parmi combien ?) pour savoir quelle partie des données du probléme il doit traiter.

2.3.2 Mise à Jour

Il s’agit, avant chaque itération, (sauf peut être la première) d’échanger les valeurs sur les zones de recouvrement des sous domaines pour mettre à jour les “conditions aux limites locales”. Pour un processeur “courant”, il y a deux échanges. Pour les processeurs des bords (`0` et `nb_procs-1`) il n’y en a qu’un.

En examinant la figure ci-dessous (2.10) nous voyons que la dernière colonne **d’inconnues** pour le sous-domaine de gauche devient la condition à gauche pour le sous-domaine de droite et, de même, la première colonne **d’inconnues** du sous-domaine de droite devient la condition à droite pour le sous domaine de gauche.

Nous obtenons le schémas suivant qui se généralise facilement pour un nombre quelconque de processeurs :



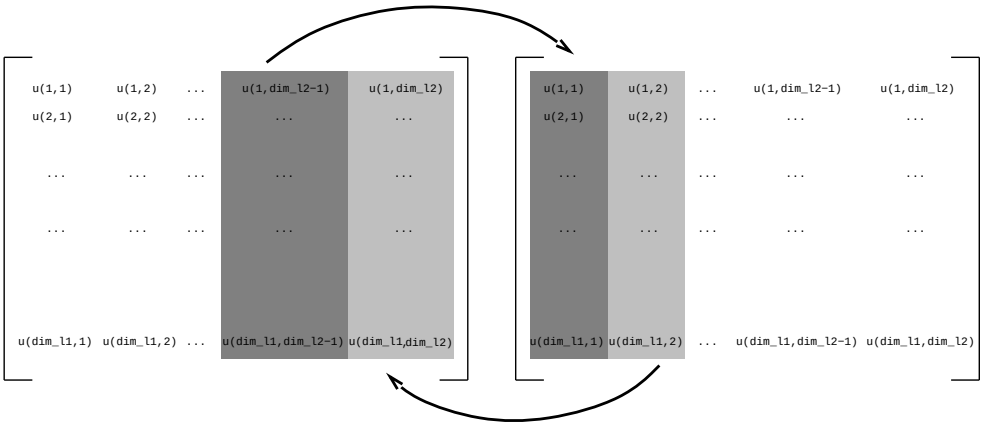


FIG. 2.10: *Les données à échanger*

La figure 2.10 donne la structure du code réalisant ces échanges. Il faut ici distinguer deux cas : le processeur a un voisin à droite ou à gauche. Si l'on découpe dans plusieurs directions, il faudrait envisager d'autre cas. Notons qu'il faut s'assurer que le programme fonctionnera même si il n'y a qu'un seul processeur.

```
si (moi >1)         // j'ai un voisin à gauche
    {
    envoyer la 1er colonne d'inconnues à "moi-1"
    recevoir,de "moi-1", la nouvelle condition aux limites
    }
si (moi <nb_procs-1)// j'ai un voisin à droite
    {
    envoyer la dernière colonne d'inconnues à "moi+1"
    recevoir,de "moi+1", la nouvelle condition aux limites
    }
```

FIG. 2.11: *Échanges aux frontières*

2.3.3 Test de Convergence

Le code de la boucle de calcul n'est pas modifié par la parallélisation de l'algorithme. Le seul soucis concerne bien le test de convergence qui se doit d'être global : "tous les processeurs doivent décider de s'arrêter en même temps". Pour apprécier globalement la convergence il faut calculer le résidu **global** : max des résidus locaux sur chaque processeur. Nous utiliserons une fonction de réduction.

```
jusqu'a convergence
   {
   ...
         {
         ...
         residu=max(residu, ||u[i,j]-u_loc||/||u[i,j]||)
         }
   //
   // max des residus locaux
   residu_glob=reduction(residu, max)
   // critère global
   si (residu_glog < epsilon) alors convergence
   }
```

FIG. 2.12: *Test de convergence global*

2.3.4 `I/O` et Résultats

Une bonne stratégie, dans un premier temps, est de laisser les entrées-sorties à la charge d'un seul des noeuds (par exemple le "noeud 0").

Pour les données d'initialisation du programme, seul ce nœud lira le fichier ou dialoguera avec l'utilisateur, puis les données lues seront diffusées aux autres nœuds.

```
{
// le noeud 0 lit les données puis les diffuse
si (moi=0)
    {
    lire_donnees
    diffusion_donnees (noeud emeteur=0) // donc emission
    }
sinon
// les autres noeuds recoivent les donnees
    {
    diffusion_donnees (noeud emeteur=0) // donc reception
    }
...
}
```

FIG. 2.13: *Lecture des données par un seul nœud*

Dans le cas de l'écriture, le nœud 0 doit ici récupérer dans le bon ordre, les résultats des autres processeurs tout en les écrivant (sur disque, à l'écran, ...) car il est clair qu'*a priori* il n'est pas possible de stocker localement le résultat global sur un seul nœud.

Néanmoins tout cela ne fonctionne que si le volume des entrées-sorties reste limité. Dans le cas contraire, il est indispensable d'utiliser le système d'entrées-sorties parallèles de la machine. On arrive alors dans le domaine du non normalisé et du non portable ; catastrophe. Il existe toutefois des "interfaces" utilisateur qui permettent d'exprimer le parallélisme des entrées-sorties. C'est le cas notamment le la partie "`IO`" de `MPI 2`[1] où l'on peut exprimer les lectures et les écritures avec une syntaxe et une sémantique proches de celles des échanges de messages. Utiliser ce type d'interface de haut niveau n'est qu'une garantie de portabilité. L'efficacité se sera au rendez-vous qu'avec une mise en œuvre optimisée de la bibliothèque `MPI`, spécifique au calculateur utilisé.

[1] Souvent appelée `MPI-IO`.

```
{
//
// le noeud 0 ecrit ses resultats puis recupere et
// ecrit dans le bon ordre ceux des autres noeuds
//
si (moi=0)
    {
    ecrire_resultats
    pour k=1 jusqu'à nb_proc-1
       {
       recevoir les résultats du processeur k
       // par exemple en ecrasant ceux que l'on
       // vient d'ecrire
       ecrire-resultats
       }
    }
sinon
//
// les autres noeuds envoient leurs resultats
// au noeud 0
//
    {
    envoyer resultats au processeur 0
    }
}
```

FIG. 2.14: *Écriture des resultats par un seul nœud*

2.3.5 Mesure de temps

Pour mesurer la performance de la parallélisation, nous nous intéresserons en général au délai total d'exécution du programme parallèle. Il s'agit donc du `max` des temps passés dans chaque processeur (opération de réduction). Pour que cette mesure sait un sens (et soit au moins reproductible) il est préférable que tous les processeurs démarrent en même temps d'où l'utilité de pouvoir les synchroniser au démarrage. Il faut penser aussi à ne pas inclure les entrées sorties dans la mesure de la performance de l'algorithme parallèle (même si bien sûr, cela a un sens de s'y intéresser par ailleurs).

La qualité de la parallélisation se mesure en comparant le "temps parallèle" au "temps séquentiel" ce qui donne l'accélération (ou "*speed-up*") obtenu sur le nombre de processeurs considéré.

```
{
...
synchronisation
debut de la mesure de temps
//
jusqu'a convergence
   {
   // boucle de calculs
   ...
   }
fin de la mesure de temps
temps_glob = reduction(temps, max) // max des temps locaux
...
}
```

FIG. 2.15: *Principe des mesures de temps*

2.4 En conclusion

Le code obtenu, dérivé simplement du code séquentiel, est bien du type SPMD. Il "passe à l'échelle" sans problème pourvu que la taille des sous domaines reste suffisante pour compenser le surcoût dû aux communications (notion de granularité).

La parallélisation très simple de ce type de problème permet d'illustrer l'utilisation des fonctionnalités de base du modèle "échanges de messages". Sept fonctions suffisent pour programmer l'application parallèle de façon efficace, c'est souvent le cas ! :

- identification du processeur, accès aux nombre total de processeurs ;
- envoi et réception de messages ;
- synchronisation, diffusion et réduction.

Bien noter aussi la façon dont doivent être traitées les entrées-sorties et les mesures de temps.

Chapitre 3

Méthodes par blocs

3.1 Un solveur direct

La résolution de nombreux problèmes de la physique conduit à résoudre des systèmes linéaires où la matrice est pleine, c'est la cas par exemple en électromagnétisme, mais la méthode numérique utilisée peut aussi impliquer la fabrication de matrices pleines (équations intégrales etc...).

Le plus souvent ces systèmes linéaires sont résolus par des méthodes directes, surtout si ils sont de grande taille.

Nous prendrons comme exemple la factorisation de Gauss (ou LU) d'une matrice en produit de deux matrices triangulaires. Pour résoudre des systèmes linéaires : $Ax = b$ avec la matrice A pleine, nous effectuons d'abord la décomposition LU de la matrice A, le problème devient : résoudre $LUx = b$.

En introduisant un vecteur auxiliaire y l'on se ramène aux deux étapes (simples) dites "descente-remontée" :

- résoudre le système triangulaire inférieur : $Ly = b$
- résoudre le système linéaire supérieur : $Ux = y$

Parmi les stratégies possibles de mise en œuvre, les méthodes par blocs se prêtent bien à la parallélisation. En effet il est possible d'exhiber, au cours des itérations, des calculs indépendants sur des blocs distincts : en résumé du parallélisme **et** de la localité.

Les méthodes directes sont particulièrement intéressantes si l'on effectue des études paramétriques. Dans ce cas, l'on "paie" une fois le coût de la factorisation pour résoudre ensuite avec plusieurs second membres - les descentes remontées- ce qui est bien sûr très avantageux (citons par exemple le calcul des signatures Radar).

3.2 Factorisation LU

Nous nous intéressons donc à la factorisation de Gauss (*A=LU*) d'une matrice carrée pleine en produit d'une matrice triangulaire inférieure *L* (pour *"lower"*) par une matrice triangulaire supérieure *U* (pour *"upper"*).

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \; se\ décompose\ en\ A = L * U \;:$$

$$A = \begin{bmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ l_{n1} & l_{n2} & \dots & 1 \end{bmatrix} * \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

FIG. 3.1: *La décomposition LU*

L'algorithme (séquentiel) classique, où $a_{ij}^{(k)}$ représente la valeur de l'élément a_{ij} à l'étape k de l'algorithme, est le suivant :

```
Pour k=1 à n-1 faire
   {
   Pour i=k+1 à n faire
      {
      a_ik^(k) = a_ik^(k-1) * (a_kk^(k-1))^-1        //(1)
      Pour j=k+1 à n faire
         a_ij^(k) = a_ij^(k-1) - a_ik^(k) * a_kj^(k-1)   //(2)
      }
   }
```

FIG. 3.2: *Décomposition LU, l'algorithme "séquentiel"*

Le diagramme ci-dessous illustre la progression de l'algorithme et les données modifiées à chaque étape *k*. Il est facile de constater que la décomposition peut se faire "en place". En effet les éléments calculés à l'étape *k* dépendent uniquement d'éléments calculés (ou non modifiés) à l'étape *k-1*.

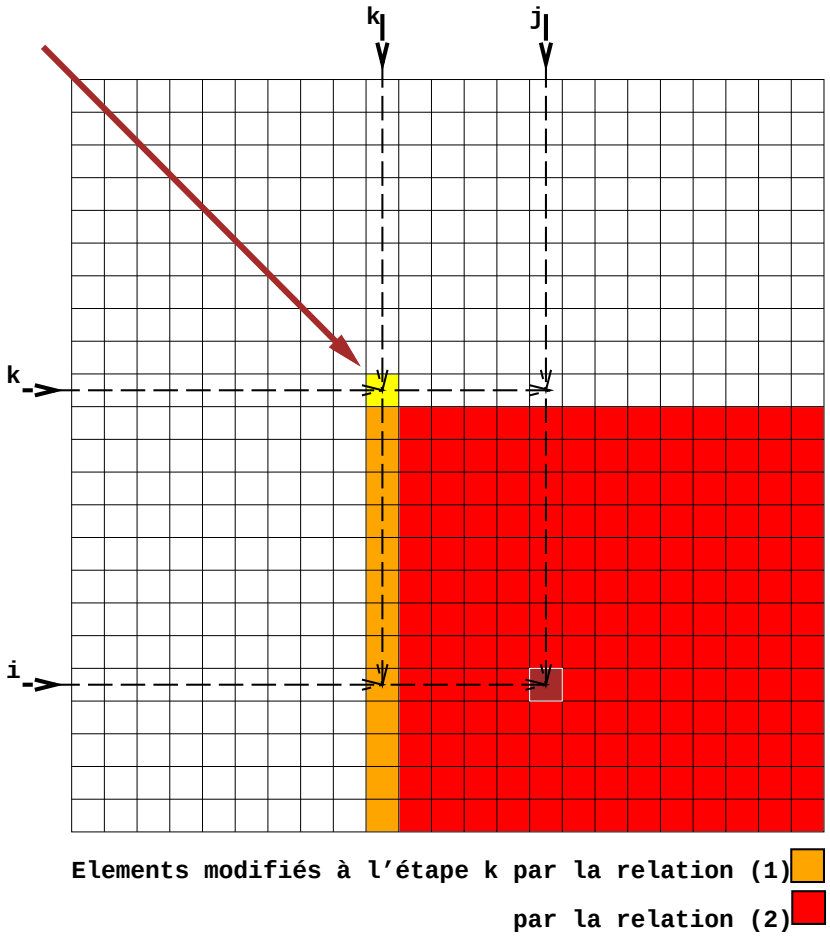


FIG. 3.3: *Factorisation LU, progression de l'algorithme*

Les nouvelles valeurs l_{ij} et u_{ij} peuvent donc prendre la place des anciens a_{ij} (l'on ne stocke pas les l_{ii} que l'on sait égaux à *1*) ce qui représente, bien sûr, une économie de place mémoire particulièrement appréciable.

Autre remarque simplificatrice, nous allons travailler, ici sur l'algorithme "sans pivotage", ce qui suppose que l'on ne trouve pas en chemin, lors de la décomposition, un élément diagonal nul, mais n'enlève rien à la généralité de l'approche.

Si un élément a_{kk} est nul (on ne peut plus diviser dans la relation **(1)**), il faut alors "pivoter" ce qui signifie permuter lignes et colonnes de la matrice jusqu'à obtentir d'un nouveau terme diagonal non nul. Il suffit de garder la trace des pivotages successifs pour reconstituer la solution du système initial en permutant les composantes de la solution du système résolu "après pivotages".

3.3 La méthode par blocs

Il est facile de voir en regardant l'algorithme séquentiel, que de nombreuses opérations peuvent être effectuées en parallèle : calcul des $a_{ik}^{(k)}$ puis des $a_{ij}^{(k)}$.

Sur un processeur, tout cela s'optimise très facilement en vectorisant ou plus simple encore en faisant appel aux bibliothèques BLAS[1]. Ainsi, nous appellerions "*dmult*" pour les opérations **(1)** de la figure 3.3 et une série de "*daxpy*" pour les opérations **(2)**. Mais pour en tirer efficacement parti dans un contexte parallèle il est nécessaire d'augmenter la taille des tâches à exécuter, c'est à dire d'augmenter la *granularité* de l'algorithme. C'est pour cela que nous envisageons une méthode par blocs.
La matrice *A*, de dimension *n* est donc décomposée en q^2 blocs de dimension *p* (avec $n=p*q$). Nous nous intéressons à la décomposition suivante :

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1q} \\ A_{21} & A_{22} & \dots & A_{2q} \\ \dots & \dots & \dots & \dots \\ A_{q1} & A_{q2} & \dots & A_{qq} \end{bmatrix} \; se\ décompose\ en\ A = L * U \ :$$

$$A = \begin{bmatrix} L_{11} & 0 & \dots & 0 \\ L_{21} & L_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ L_{q1} & L_{q2} & \dots & L_{qq} \end{bmatrix} * \begin{bmatrix} I & U_{12} & \dots & U_{1q} \\ 0 & I & \dots & U_{2q} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & I \end{bmatrix}$$

FIG. 3.4: *Décomposition LU par blocs*

Le calcul de la décomposition par blocs se fait avec le même type d'algorithme mais les calculs portent maintenant sur des matrices puisque chaque élément A_{ij}, U_{ij}, L_{ji}, est cette fois une matrice de taille *q* .

[1] "*Basic Linear Algebra Subprograms*".

Notons que, contrairement au cas scalaire, l'algorithme est écrit "par colonnes" plutôt que "par lignes". Ainsi l'on retrouve les *I* (matrices identité de taille *q*) sur la partie triangulaire inférieure *L* et non sur *U*. Il y a une raison pratique, il s'agit d'éviter d'effectuer les produits à droite par des inverses de matrice comme nous le verrons plus tard. L'algorithme est le suivant :

```
Pour k=1 à q-1 faire
   {
   Pour j=k+1 à q faire
      {
      A_kj^(k) = (A_kk^(k-1))^-1 * A_kj^(k-1)        //(3)
      Pour i=k+1 à q faire
         A_ij^(k) = A_ij^(k-1) - A_kj^(k) * A_ik^(k-1) //(4)
      }
   }
```

FIG. 3.5: *Décomposition LU par blocs, version 1*

Comme précédemment, $A_{ij}^{(k)}$ représente la valeur du bloc A_{ij} à l'étape k de l'algorithme et l'on remarque que l'algorithme peut s'effectuer "en place", les nouveaux blocs U_{ij} et L_{ij} prenant la place des anciens. Enfin les blocs U_{ii} que l'on sait égaux à I n'ont pas besoin d'être stockés.

3.4 Mise en œuvre

Pour des réels, écrire $x^{-1} * y$ ou $y * x^{-1}$ c'est la même chose, mais le problème se complique s'il s'agit de matrices !. Dans ce cas le produit "à gauche par l'inverse" est plus simple à réaliser.
En effet, effectuer

$$A_{ik}^{(k)} = \left(A_{kk}^{(k-1)}\right)^{-1} * A_{ik}^{(k-1)}$$

(relation **(3)** de l'algorithme présenté figure 3.5), revient à résoudre :

$$A_{kk}^{(k-1)} * A_{ik}^{(k)} = A_{ik}^{(k-1)} \quad où\ A_{ik}^{(k)}\ est\ l'inconnue.$$

Pour cela appelons $V_l^{(k-1)}$ et $V_l^{(k)}$ les vecteurs colonne de rang *l* des matrices $A_{ik}^{(k-1)}$ et $A_{ik}^{(k)}$ le problème devient :

$$\forall l \in [1,p] \; résoudre \quad \left(A_{kk}^{(k-1)}\right) V_l^{(k)} = V_l^{(k-1)}$$

c'est à dire que nous sommes ramené à la résolution de *p* systèmes linéaires avec pour matrice $A_{kk}^{(k-1)}$ et les vecteurs $V_l^{(k-1)}$ comme seconds membres. Bien sûr, la matrice $A_{kk}^{(k-1)}$ étant pleine, la bonne idée est de la factoriser (*LU* !) puis d'effectuer les *p* descentes remontées pour calculer les vecteurs colonne de $A_{ik}^{(k)}$. La matrice $A_{kk}^{(k-1)}$ n'étant pas de grande taille, l'algorithme séquentiel optimisé (BLAS, ...) sera tout à fait indiqué.

Ainsi l'algorithme par bloc dans sa forme finale est le suivant ; chaque bloc diagonal est factorisé, y compris le dernier pour des raisons d'homogénéité[2], ce qui justifie la boucle externe "*k=1 à q*" :

```
Pour k=1 à q faire
   {
  factorisation LU du bloc A_kk^(k-1)

   Pour i=k+1 à q faire
      {
     A_ik^(k) = (A_kk^(k-1))^-1 * A_ik^(k-1) // soit p descentes-remontées

      Pour j=k+1 à q faire
        A_ij^(k) = A_ij^(k-1) - A_ik^(k) * A_kj^(k-1)
      }
   }
```

FIG. 3.6: *Décomposition LU par blocs, version 2*

3.5 L'analyse du parallélisme

Pour analyser finement le parallélisme d'un tel algorithme, nous cherchons à isoler des tâches "élémentaires" qui seront réparties sur les nœuds de calcul, puis

[2]Ce qui simplifie les choses pour les "descentes-remontées par bloc".

à déterminer les dépendances entre ces tâches, c'est à dire "*telle tâche à besoin pour s'exécuter des résultat de telle autre etc...*".
Nous distinguons ici trois tâches élémentaires :

- tâche `Gauss (k)` : décomposition *LU* de la matrice A_{kk} (le $k^{ème}$ bloc diagonal à l'étape *k-1*) ;
- tâche `Inv_mult (k, j)` : multiplication par l'inverse du bloc diagonal A_{kk} du bloc A_{kj} (ligne *k*, colonne *j*) à l'étape *k* (le bloc A_{kk} étant factorisé, cela se ramène à la résolution de *p* systèmes linéaires) ;
- tâche `Maj (i, j, k)` : mise à jour du bloc A_{ji} (ligne *i*, colonne *j*) à l'étape *k* (produit et combinaison linéaire de matrices).

Nous pouvons alors reécrire l'algorithme en terme d'instanciations de ces tâches "élémentaires" :

```
pour k=1 à q
   {
   Gauss (k);            // (5)
   pour j=k+1 à q
      Inv_mult (k, j);   // (6)
   pour j=k+1 à q
      pour i=k+1 à q
         Maj (i, j, k);  // (7)
   }
```

FIG. 3.7: *Décomposition `LU` par blocs, version `3`*

Cette fois les tâches exécutables en parallèle -sûrement les opérations **(6)** et **(7)** mais n'excluons pas **(5)** *a priori*- présentent une *granularité* suffisante (cf figure 3.7). Mieux elles sont du "même ordre" de complexité. En effet, si la taille des blocs est de *p* :

- la factorisation *LU*, tâche `Gauss` est en $o(p^3)$ opérations ;
- les descentes remontées (résolutions de systèmes linéaires) sont en $o(p^2)$ opérations. Mais il faut en effectuer *p* ce qui donne bien $o(p^3)$ pour l'ensemble de la tâche `Inv_Mult` ;
- enfin la tâche `Maj` fait intervenir le produit de deux matrices de taille *p* qui est aussi en $o(p^3)$ opérations.

Autre bonne nouvelle, remarquons que la taille mémoire occupée par un bloc est en $o(p^2)$, ce qui augure bien de l'avenir. En effet, si le travail est réparti entre plusieurs processeurs, il faudra probablement transférer des blocs entre les différents

nœuds de la machine et il est rassurant de savoir *a priori* que l'ordre de grandeur du volume des communications ($o(p^2)$) est inférieur à celui des calculs ($o(p^3)$).

La figure 3.8 permet de visualiser la progression de l'algorithme, ici dans le cas de *q=4* :

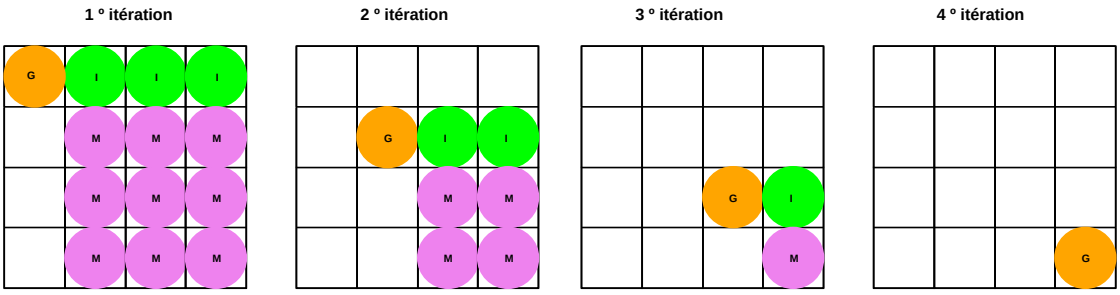


FIG. 3.8: *Factorisation LU par blocs, progression de l'algorithme*

3.6 Les dépendances

Analysons maintenant de façon plus fine les dépendances entre les tâches, l'on note `T1`→`T2` si la tâche `T2` "dépend de" la tâche `T1`. Ici cela signifie que `T1` intervient dans la production de résultats qui seront utilisés par `T2` et doit donc être exécutée avant :

1. une tâche `Gauss` ne peut avoir lieu que si le bloc sur lequel elle opère a été mis à jour à l'étape précédente ce qui donne pour tout :

   $\forall k \in [2,q]$ `Maj(k,k,k-1)`→`Gauss(k);`

2. Une tâche `Inv_mult` ne peut avoir lieu que si le bloc diagonal correspondant a éte factorisé à la même étape ce qui donne :

   $\forall k \in [1,q-1], \forall j \in [k+1,q]$ `Gauss(k)`→`Inv_mult(j,k);`

3. Une tâche `Inv_mult` ne peut avoir lieu que si le bloc sur lequel elle opère a été mis à jour à l'étape précédente ce qui donne :

   $\forall k \in [2,q-1], \forall j \in [k+1,q]$ `Maj(j,k,k-1)`→`Inv_mult(j,k);`

4. Une tâche `Maj` sur un bloc *i, j* à l'étape *k* ne peut avoir lieu que si le bloc *k, j* a subi l'opération `Inv_mult` à la même étape ce qui donne :

$\forall k \in [1, q-1], \forall (i,j) \in [k+1, q] \times [k+1, q]$ `Inv_mult(k,j)`$\rightarrow$`Maj(i,j,k);`

5. Une tâche `Maj` sur un bloc *i, j* à l'étape *k* ne peut avoir lieu que si il a déjà subi cete opération à l'étape *k-1* ainsi que le bloc *i, k* ce qui donne :

$\forall k \in [2, q-1], \forall (i,j) \in [k+1, q] \times [k+1, q]$ `Maj(i,j,k-1)`$\rightarrow$`Maj(i,j,k);`
$\forall k \in [2, q-1], \forall (i,j) \in [k+1, q] \times [k+1, q]$ `Maj(k,i,k-1)`$\rightarrow$`Maj(i,j,k);`

Nous pouvons visualiser sur un graphe orienté, appelé *graphe de dépendance*, les instances des tâches et leurs dépendances mutuelles. Pour cet algorithme, avec ce qui précède, il est clair que le graphe sera relativement complexe. La figure 3.9 en donne une représentation pour $q = 4$ (*q* est le nombre de blocs dans chaque direction).

Le traitement commence à la "racine" : $\texttt{Gauss}_{11}^{(1)}$ dans l'exemple de la figure 3.9. Sur un graphe de dépendances, les tâches exécutables en parallèle n'apparaissent pas au premier coup d'œil. Ainsi dans l'exemple figure 3.9, il est tout à fait possible de démarrer $\texttt{Inv\_Mult}_{23}^{(2)}$ avant $\texttt{Maj}_{33}^{(1)}$, pourvu que les tâches qui "précèdent" $\texttt{Inv\_Mult}_{23}^{(2)}$ dans le graphe aient aussi été exécutées.

Notons que pour ce problème, nous savons évaluer, en fonction des paramètres *p* et *q*, le coût des tâches (en *nombre d'opérations*) et des communications (en *nombre d'octets échangés*). Nous pouvons donc mettre "un coût" en face de chaque nœud et chaque arête.

L'exploitation complète de ce graphe en vue d'obtenir un délai d'exécution minimal est possible à l'aide d'outils automatiques basés sur des heuristiques et conduit à générer, sur chaque nœud de la machine parallèle, un programme spécifique :

- en fonction des valeurs de *p* et *q* ;
- et des caractéristiques de la machine cible : puissance du processeur, débit du réseau de communication.

L'idée est intéressante pour résoudre de façon optimale un problème bien précis sur une architecture matérielle donnée (dans un système embarqué par exemple) mais il est difficile d'en tirer un programme général à lancer sur chacun des nœuds de la machine, comme notre modèle de programmation SPMD nous y invite.

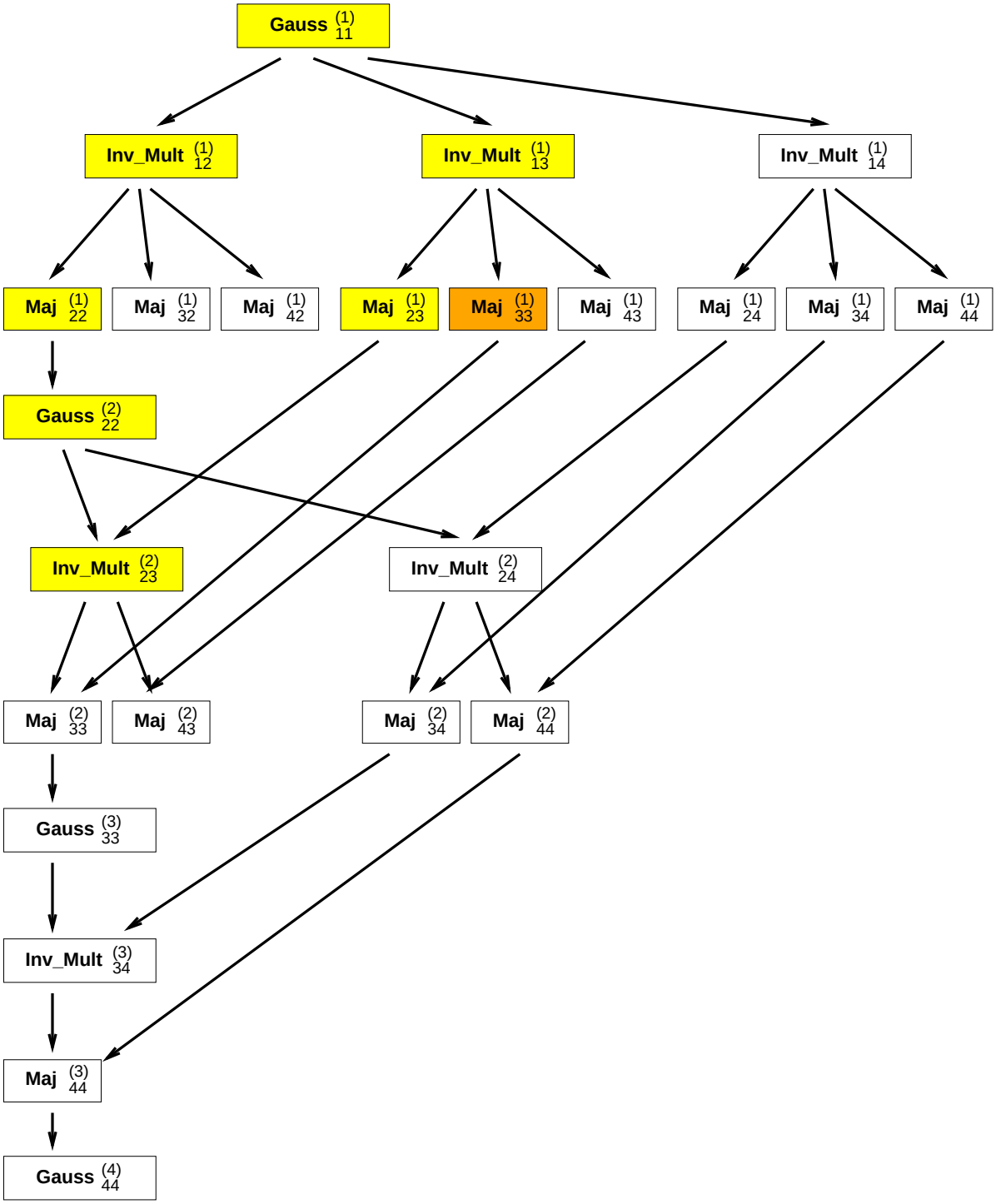


FIG. 3.9: *Factorisation LU par blocs, graphe des tâches version (1)*

Pour essayer de conclure, nous allons rajouter des contraintes supplèmentaires afin de simplifier le graphe. Une contrainte simple que l'on peut se donner est de ne pas entrelacer les itérations pour la boucle extérieure en *k*. Avons-nous vraiment simplifié les choses pour le graphe de dépendances ? Oui, car il n'y a plus que trois relations au lieu de cinq et les dépendances deviennent :

1. Une opération `Inv_mult` ne peut avoir lieu que si le bloc diagonal corespondant a éte factorisé à la même étape ce qui donne :

   $\forall k \in [1, q-1], \forall j \in [k+1, q]$ `Gauss(k)`$\rightarrow$`Inv_mult(j, k);`

2. Une opération `Maj` sur un bloc *i, j* à l'étape *k* ne peut avoir lieu que si le bloc *k, j* a subi l'opération `Inv_mult` à la même étape ce qui donne :

   $\forall k \in [1, q-1], \forall (i,j) \in [k+1, q] \times [k+1, q]$ `Inv_mult(k,j)`$\rightarrow$`Maj(i,j,k);`

3. Une opération `Gauss` ne peut avoir lieu que si tous les bloc ont été mis à jour à l'étape précédente ce qui donne

   $\forall k \in [2, q], \forall (i,j) \in [k+1, q-1] \times [k+1, q-1]$ `Maj(i,j,k-1)`$\rightarrow$`Gauss(k);`

De plus le graphe devient lui, très régulier, par exemple pour $q = 4$ sur la figure 3.10, à comparer avec la précédente 3.9. Nous voyons, cette fois-ci apparaître "horizontalement" les tâches pouvant être exécutées en parallèle. Bien sûr dans cet exemple, nous retrouvons -sans surprise- les instructions de base de l'algorithme par bloc (figure 3.7) néanmoins l'approche utilisée est beaucoup plus générale et peut s'appliquer à n'importe quel algorithme.

Les graphes de dépendances pour l'algorithme de la factorisation montrent que le parallélisme potentiel de l'algorithme augmente avec le nombre de blocs (*q*) -idéal pour le "passage à l'échelle"- mais qu'il décroit de toute façon au cours de l'algorithme et ce tout particulièrement avec la deuxième version.

Cette dernière version, en revanche, est facile à exploiter dans le cadre d'une application `SPMD` car nous avons mis en évidence des **séries d'opérations identiques** exécutables en parallèle. Ce code `SPMD` donnera (sûrement) d'honnêtes performances sur une grande gamme de problèmes mais ne sera pas optimal. L'utilisateur aura toujours la possibilité d'adapter au mieux sa résolution aux performances de la machine cible en jouant, pour une taille de matrice n donnée, sur les valeurs de q et de p : le nombre de blocs et leur taille.

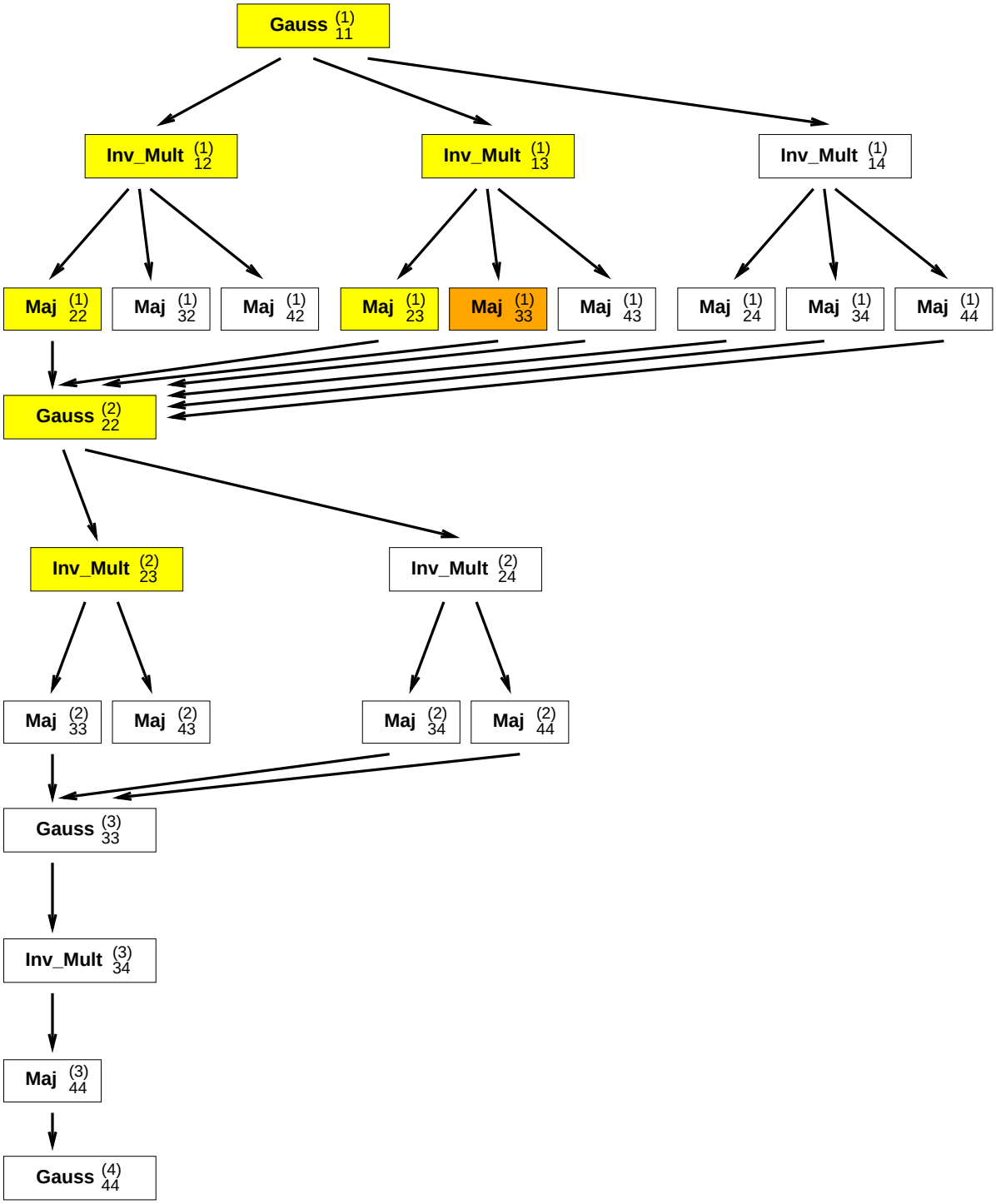


FIG. 3.10: *Factorisation LU par blocs, graphe des tâches version (2)*

3.7 Répartition des données

De nombreuses stratégies de répartition des données sont possibles, il y a un lien avec ce qui précède car le placement des blocs sur les mémoires locales conditionnera leurs...déplacements éventuels. Nous détaillerons la stratégie suivante : chaque processeur traite complètement une colonne de blocs de la matrice initiale (i.e. un découpage plutôt "orthogonal aux tâches de calcul").

Un point important, nous nous donnons aussi une contrainte de **répartition finale** : la répartition des blocs après factorisation doit être la même que la répartition initiale, ce qui facilite la suite des opérations (car il faudra bien effectuer après des descentes remontées...). Pour simplifier -encore- la programmation, nous supposerons que le nombre de processeurs disponibles sur la machine parallèle (`nb_proc`) est au moins égal au nombre de colonnes (`q`) à traiter et nous placerons une colonne de blocs par processeur. Dans le cas général il faudra éventuellement *replier* la matrice, chaque processeur pouvant traiter plusieurs colonnes de blocs, nous y reviendrons car il y a là matière à optimisation.

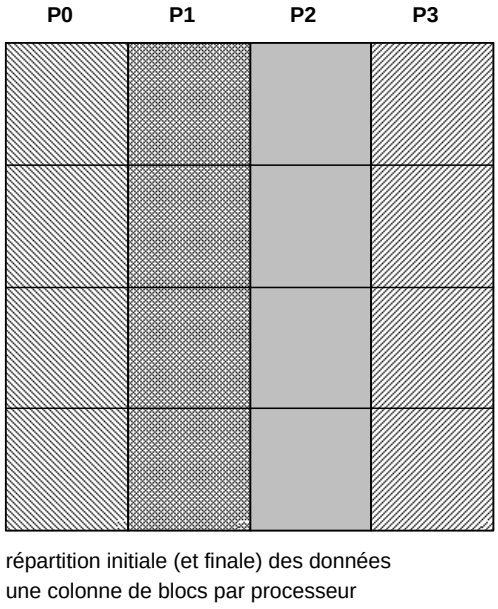


FIG. 3.11: *4 fois 4 blocs sur 4 processeurs*

À chaque itération, un processeur effectue la tâche `Gauss` puis les autres effectuent en parallèle les tâches `Inv_mult` et `Maj`. Les schémas qui suivent (figures 3.12 et 3.13) explicitent les deux premières itérations, avec la répartition de données choisie, pour un cas simple.

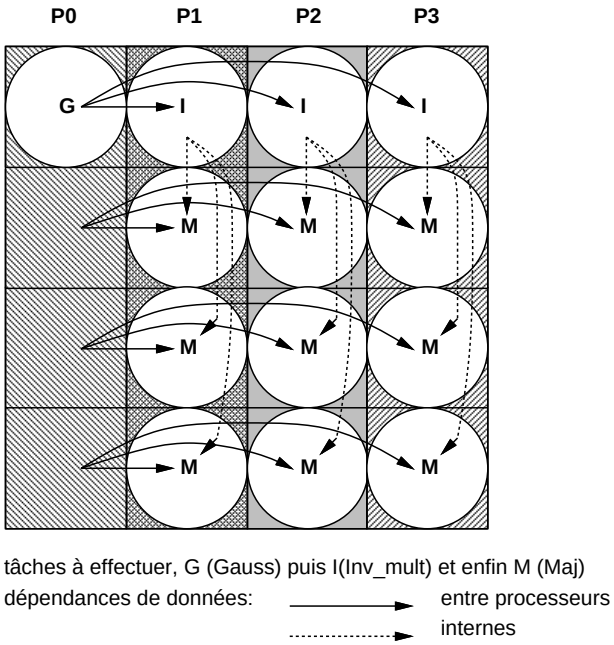


FIG. 3.12: *4 fois 4 blocs sur 4 processeurs, itération 1*

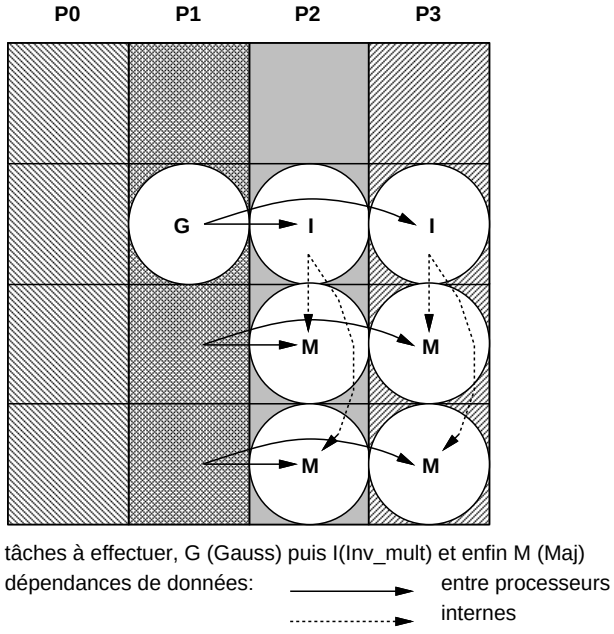


FIG. 3.13: *4 fois 4 blocs sur 4 processeurs, itération 2*

Ces schémas mettent bien en évidence les dépendances de données. À chaque itération, le processeur qui effectue la tâche `Gauss` doit transmettre **à tous les processeurs impliqués dans l'itération** :

- le bloc diagonal nécessaire aux tâches `Inv_Mult` ;
- **les blocs sous-diagonaux** nécessaires aux tâches `Maj`.

La nature et le volume des échanges dépendent donc clairement du placement des blocs sur les différents processeurs de la machine.

Une stratégie par lignes en tenant compte de la contrainte de "répartition finale" exposée ci dessus conduit à moins d'échanges (seuls les blocs ayant subi `Inv_Mult` sont à envoyer) mais aussi à moins de parallélisme : le processeur qui effectue `Gauss` devant aussi effectuer les tâches `Inv_Mult` si l'on ne veut pas multiplier les échanges de données.

3.8 Programmation

Notre modèle étant `SPMD`, nous écrirons donc un seul programme qui s'exécutera sur tous les processeurs de la machine. À "l'intérieur" de ce programme, l'on pourra décider en fonction de l'environnement : "qui suis-je ? parmi combien ?" la part de travail à effectuer à chaque itération *k* par le processeur courant "*moi*".

Nous supposons, bien sûr, que l'on dispose des programmes séquentiels effectuant les tâches "de base" `Gauss`, `Inv_Mult`, et `Maj` :

Le fragment de programme de la figure 3.14 représente la boucle centrale de l'algorithme qui s'exécute **sur chaque processeur**.

Pour bien comprendre ce qui se passe, il faut raisonner "localement" en s'appuyant par exemple sur les schémas 3.12 et 3.13. Quelles opérations doit effectuer à l'itération *k* le processeur qui détient initialement une colonne de blocs donnée. Traditionnellement la numérotation des processeurs commence à zéro, aussi dans notre cas, le processeur *moi* détient la colonne *moi+1*.

Notons qu'avec cette stratégie, les envois de blocs peuvent toujours se faire sous forme de diffusions (i.e. envoi à tous les autres nœuds). Globalement, en effet cela prend moins de temps d'effectuer des diffusions (optimisées selon l'architecture) que d'envoyer les messages un par un aux processeurs destinataires. Les processeurs qui, au cours des itérations, cessent de travailler (par exemple le processeur `P0` figure 3.13) se contentent donc de réceptionner les messages sans les traiter.

Attention dans le pseudo code de la figure 3.14, les numeros des blocs se réfèrent à la numérotation globale des blocs de la matrice. Il y a bien sur lieu de l'adapter en fonction du mode de stockage local au nœud.

```
// moi:processeur courant, q: nombre de blocs
//
pour k=1 à q faire
   {
    si (k==moi+1)
       {
       //  factorisation du K eme bloc diagonal
       //  diffusion du bloc factorise
       //  et des blocs "sub-diagonaux"
       Gauss (bloc[k,k]);
       if (k< q)
          pour i= k, q
             diffuser le bloc [i,k];
       }
    sinon
       {
       // recevoir le bloc diagonal factorise
       // si moi+1 > k  effectuer la tache I
       recevoir le bloc [k,k]
       si (moi+1 >k)  alors
          Inv_Mult (bloc[k, moi+1]);
       // recevoir les blocs sous diagonaux
       // si moi+1 > k  effectuer la tache M
       pour i=k+1, q faire
          {
          recevoir le bloc [i,k];
          si (moi+1 >k)  alors
             Maj(bloc[i,moi+1];
          }
       }
   }
```

FIG. 3.14: *Factorisation par blocs parallèle : Algorithme du processeur k*

Cet algorithme est très proche de l'algorithme séquentiel : nous nous sommes contenté de "sélectionner" les tâches effectuées par le processeur courant et de rajouter les communications. Il fonctionne quelque soit le nombre de processeurs.

Si le nombre de blocs est bien supérieur au nombre de processeurs, chaque processeur traitera plusieurs colonnes de blocs et il suffira d'adapter les tests et les communications en conséquence. Dans ce cas nous pouvons pallier au manque de parallélisme de l'algorithme en gérant la distribution des colonnes sur les processeurs. Si l'on opte pour une distribution cyclique plutôt que par blocs (figure 3.15), tous les processeurs auront des tâches à exécuter à chaque itération. Ce repliage peut donc être utiliser pour gagner du parallélisme. Nous voyons ici une

autre façon d'adapter l'algorithme à l'architecture de la machine parallèle.

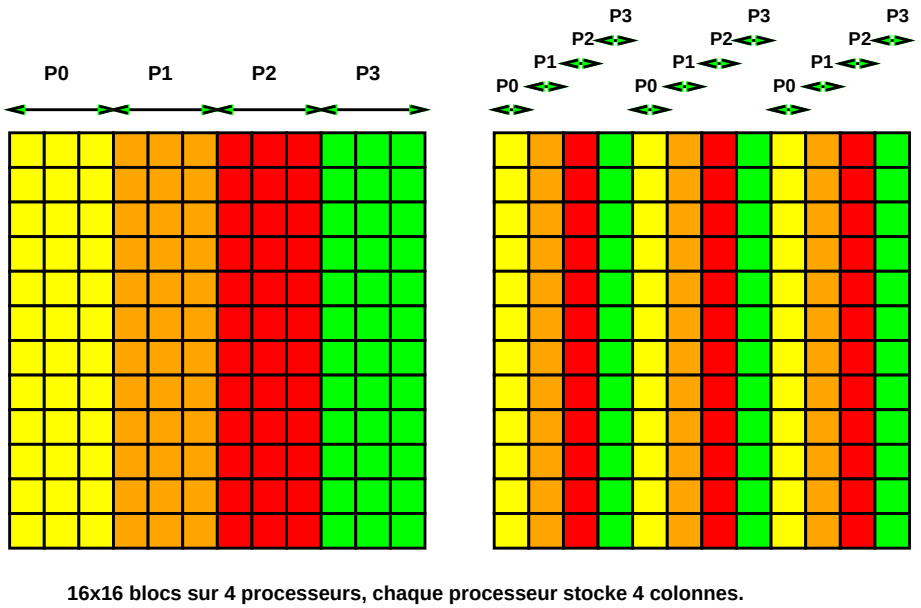


FIG. 3.15: *Distribution par blocs et distribution cyclique*

Enfin si l'on travaille avec une très grande matrice comme cela peut être le cas en électromagnétisme, une colonne de blocs ne "tient pas" forcément dans la mémoire locale du nœud. Il faudra alors lire et écrire les blocs sur disque au fur et à mesure du traitement. L'analyse doit alors montrer quels sont à un moment donné les blocs "actifs". En pratique, l'on peut mener à bien l'algorithme si l'on est capable de stocker 3 blocs sur chaque nœud. Si l'on dispose d'un peu plus de place, il est alors possible de programmer habilement les entrées sorties pour qu'elles s'exécutent en même temps que les calculs afin de ne pas ralentir la partie calcul de l'algorithme.

3.9 En conclusion

Comme au chapitre précédent, nous obtenons du code SPMD, dérivé simplement du code séquentiel, qui "passe à l'échelle" dans la mesure où la granularité demeure suffisante. Le code parallèle, de même n'utilise que les fonctionnalités de base de l"'échange de messages". Cet exemple montre que l'analyse du parallélisme peut devenir très complexe, il illustre aussi le lien non trivial entre placement des données et répartition des calculs.

Chapitre 4

Contexte "éléments finis"

4.1 généralités

Il n'est pas question, ici, de détailler la méthode des éléments finis. Nous supposons connues et maîtrisées les différentes étapes, que l'on peut résumer très schématiquement par :

- maillage ;
- discrétisation ;
- assemblage ;
- résolution.

Ce qui est certain, c'est que tout cela se termine le plus souvent par la résolution d'un système linéaire dont les principales caractéristiques sont d'être :

1. creux ;
2. de grande taille.

Pour ces deux raisons, la matrice du système n'est pas stockée entièrement, diverses stratégies de stockage permettent d'éviter de conserver les éléments nuls (stockage "profil", "Morse"). Il s'agit toujours de solutions de compromis : plus le stockage est compact, plus le coût d'accès à un élément peut se révéler élévé car il necessite des tests.

De toutes façons, dans le cas de stockage compact ("Morse"), il n'est plus question d'utiliser des méthodes directes qui "remplissent" la matrice en faisant apparaître de nouveaux éléments non nuls. La structure de la matrice en terme de représentation informatique, doit rester figée au cours du calcul.

Il ne reste qu'une utilisation possible pour ces matrices, c'est de les utiliser pour effectuer des produits matrice-vecteur dans les méthodes de résolution.

Les méthodes de résolution les plus utilisées dans ce contexte sont les méthodes de descente et notamment la méthode du gradient conjugué.

4.2 Rappel : le gradient conjugué

Pour résoudre $Ax = b$ avec A symétrique définie positive, nous considèrons la fonctionnelle :

$$J(x) = \frac{1}{2} < Ax, x > - < b, x >$$

où $< x, y >$ désigne le produit scalaire de x par y.
Dans ce contexte (A symétrique définie positive), le minimum de $J(x)$ est atteint si le gradient de J s'annule. Or

$$\nabla J(x) = Ax - b$$

et

$$\nabla J(x) = 0 \Leftrightarrow Ax = b$$

Qu'est-ce que la méthode du gradient conjugué :

- la méthode est une méthode itérative dans la mesure où il s'agit, à partir de x_0 donné, de fabriquer une suite d'itérés x_1, x_2,..., x_i ;
- la méthode est une méthode de descente, à chaque itération i, nous choisissons une direction de descente, ω_{i-1}[1] et nous minimisons ∇J le long de cette direction à partir de l'itéré courant x_i (c'est à dire pour tous les x tels que $x = x_{i-1} + \rho_i \omega_{i-1}$) ;
- la méthode du gradient (simple) consisterait à choisir comme direction de descente...le gradient lui même, ici nous choisissons de plus de "conjuguer" la nouvelle direction par rapport à celle précédente. À l'itération i, ω_i est calculée à partir du gradient g_i et de ω_{i-1} de telle manière que ω_i et ω_{i-1} soient orthogonaux par rapport au produit scalaire de A, c'est à dire que l'on ait $< A\omega_i, \omega_{i-1} >= 0 >$;

Les directions de descente successives étant 2 à 2 conjuguées, la méthode converge théoriquement en exactement n itérations si n est la dimension de A. Dans la pratique, heureusement, le nombre d'itérations nécessaires est petit devant n, c'est là tout l'intérêt de la méthode.

Rappelons que la convergence dépend du conditionnement de la matrice A, rapport de la plus grande valeur propre (en module) sur la plus petite (en module) et aussi de la répartition de ces valeurs propres, il faut mieux qu'elles se rapprochent de l'unité.

[1] Appelée $i - 1$ pour être cohérent avec l'algorithme présenté par la suite.

On améliore généralement tout cela en effectuant un préconditionnement du système linéaire à résoudre. Soit P une inverse approchée de A, nous résolvons alors le système :

$$(PA)x = (Pb)$$

P une inverse approchée dans le sens “calculée de façon économique”. Les techniques les plus utilisées sont le “préconditionnement par la diagonale” où P est diagonale et ses éléments sont simplement les inverses des termes diagonaux de A et “LU incomplet” où l'on applique l'algorithme de factorisation en négligeant le remplissage.

Notons enfin que l'on essaie souvent de dépasser le cadre théorique d'applications de ces méthodes en les utilisant pour des systèmes dont la matrice A, réputée inversible, n'est (par exemple) pas symétrique -ou pas hermitienne en complexe-. Il convient alors d'effectuer quelques aménagement. Par exemple, conserver les directions de descente successives pour orthogonaliser la nouvelle direction par rapport à toutes (ou une partie) des précédentes, etc...

Tout cela ne change pas l'esprit et...les problèmes posés par la mise en œuvre de ces méthodes et donc leur parallélisation.

4.2.1 Calcul du coefficient de descente

A l'étape i de l'algorithme, nous disposons de l'itéré x_{i-1} et de la direction de descente ω_{i-1} (calculés à l'itération précédente $i-1$). Le nouvel itéré x_i doit minimiser la fonctionnelle $J(x)$ dans la direction ω_{i-1}. Nous pouvons écrire :

$$x_i = x_{i-1} + \rho_i \omega_{i-1}$$

où ρ_i minimise le gradient dans la direction ω_{i-1}, c'est à dire la fonction scalaire :

$$f(\rho) = J(x_{i-1} + \rho\omega_{i-1})$$

Nous obtenons en développant :

$$\begin{aligned} f(\rho) &= \frac{\rho^2}{2} < A\omega_{i-1}, \omega_{i-1} > + \rho(< Ax_{i-1}, \omega_{i-1} > - < b, \omega_{i-1} >) \\ &\quad + \frac{1}{2} < Ax_{i-1}, x_{i-1} > - < b, x_{i-1} > \end{aligned}$$

et le minimum est obtenu si la dérivée par rapport à ρ s'anulle, ρ_i est donc donné par :

$$\rho < A\omega_{i-1}, \omega_{i-1} > + < Ax_{i-1}, \omega_{i-1} > - < b, \omega_{i-1} >= 0$$

En tenant compte du fait que $Ax_{i-1} - b$ est justement la valeur g_{i-1} du gradient dont nous disposons car elle a été calculée à l'itération $i - 1$ nous obtenons :

$$\rho_i = \frac{- < g_{i-1}, \omega_{i-1} >}{< A\omega_{i-1}, \omega_{i-1} >}$$

4.2.2 Calcul du coefficient de conjugaison

La nouvelle direction de descente est calculée à partir du nouveau gradient à l'itération i g_i et et le la direction de descente précédente ω_{i-1}. Nous devons avoir :

$$< A\omega_i, \omega_{i-1} >= 0$$

Soit γ_i le coefficient de conjugaison tel que :

$$\omega_i = g_i + \gamma_i \omega_{i-1}$$

on obtient :

$$< A\omega_i, \omega_{i-1} >=< A\omega_{i-1}, g_i > +\gamma_i < A\omega_{i-1}, \omega_{i-1} >= 0$$

et enfin :

$$\gamma_i = -\frac{< A\omega_{i-1}, g_i >}{< A\omega_{i-1}, \omega_{i-1} >}$$

4.2.3 L'algorithme

Notons qu'il faut bien sûr compléter tout cela par un test de convergence. Ce dernier s'effectue, le plus souvent, sur le gradient.

initialisation :

x_0	// donné
$g_0 = Ax_0 - b$	// premier gradient
$\omega_0 = g_0$	// première direction de descente

jusqu'a convergence (itération i) :

(1)	$v = A\omega_{i-1}$	
(2)	$sc =< v, \omega_{i-1} >$	// c'est à dire $< A\omega_{i-1}, \omega_{i-1} >$
(3)	$loc =< g_{i-1}, \omega_{i-1} >$	
(4)	$\rho_i = -loc/sc$	// coef. de descente
(5)	$x_i = x_{i-1} + \rho_i \omega_{i-1}$	// nouvel itéré
(6)	$g_i = g_{i-1} + \rho_i v$	// nouveau gradient
		// ($Ax_i - b = Ax_{i-1} - b + \rho_i A\omega_{i-1}$)
(7)	test de convergence	// si négatif, on continue
(8)	$loc =< v, g_i >$	
(9)	$\gamma_i = -loc/sc$	// coef. de conjugaison
(10)	$\omega_i = g_i + \gamma_i \omega_{i-1}$	// nouvelle direction de descente

FIG. 4.1: *L'algorithme "séquentiel" du gradient conjugué*

Un (bon) reflexe de numéricien-informaticien est de regarder le coût d'une itération. Rapellons que la matrice A est creuse. Le nombre moyen d'éléments non nuls par ligne p est réputé "petit" devant sa dimension n. En fait p est lié au nombre moyen de voisins d'un sommet dans le maillage.

Si l'on regarde l'algorithme, les principales étapes "coûteuses" sont :

- un produit matrice vecteur (étape (1)) dont le coût est en $O(pn^2)$;
- trois produits scalaires (étapes (2), (3) et (8)). Le coût d'un produit scalaire est en $O(n^2)$;
- trois combinaisons linéaires de vecteurs (étapes (5), (6) et (10)). Le coût d'une combinaison linéaire est en $O(n)$.

Ainsi même si la matrice est creuse, le produit matrice vecteur reste l'opération la plus coûteuse c'est à dire celle que l'on s'attachera à paralléliser en priorité. Rappelons, de plus, que le stockage "creux" peut induire des surcoûts d'accès aux éléménts de cette matrice.

4.3 Une première approche

4.3.1 Découper la matrice

La première idée qui vient à l'esprit est de répartir la matrice A sur les mémoires locales pour paralléliser le produit matrice vecteur. Une façon simple consiste à couper A en "tranches" et à allouer une "tranche" par processeur de la machine utilisée.

Si la machine comporte $Nproc$ processeurs, ils seront numérotés de 0 à $Nproc-1$. Chaque processeur $k = 0, Nproc - 1$ reçoit une "tranche" $A^{(k)}$ de la matrice :

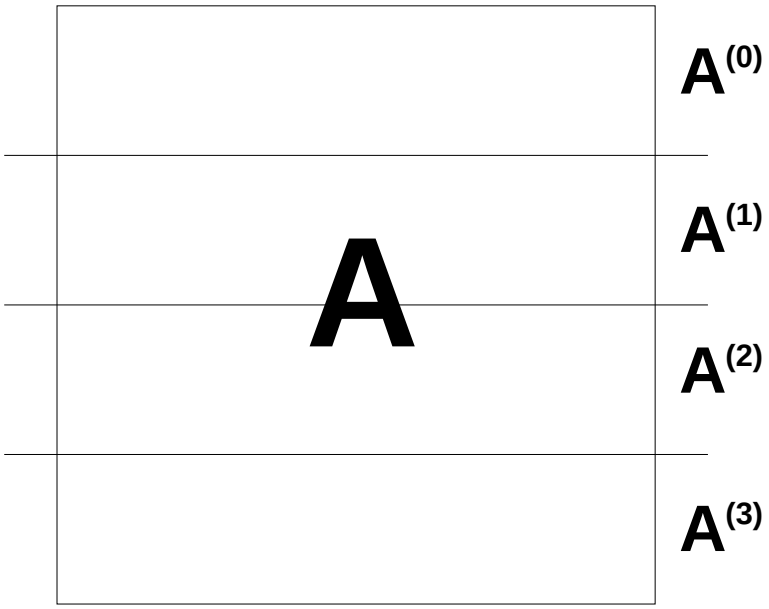


FIG. 4.2: *Matrice "en tranches", un exemple sur 4 processeurs.*

Une remarque initiale, il faut mieux laisser de coté le caractère symétrique de la matrice...et ne pas trop optimiser le stockage, n'oublions pas que l'on veut paralléliser de façon simple un produit matrice vecteur.

Ensuite il nous faut analyser ce qui se passe pour les vecteurs manipulés dans l'algorithme à savoir : v, ω_i, g_i et x_i en souhaitant, bien sûr, pouvoir aussi les répartir de la même façon. On notera $v^{(k)}$, $\omega_i^{(k)}$, $g_i^{(k)}$ et $x_i^{(k)}$ les portions de vecteurs qui correspondent aux "tranches" de la matrice A.

4.3.2 Impact sur le produit matrice vecteur

Ça commence mal ! Il est facile de voir qu'à l'étape `(1)`, pour calculer la partie $v^{(k)}$ de v, il faut que le vecteur ω_{i-1} soit présent en entier dans la mémoire du processeur k :

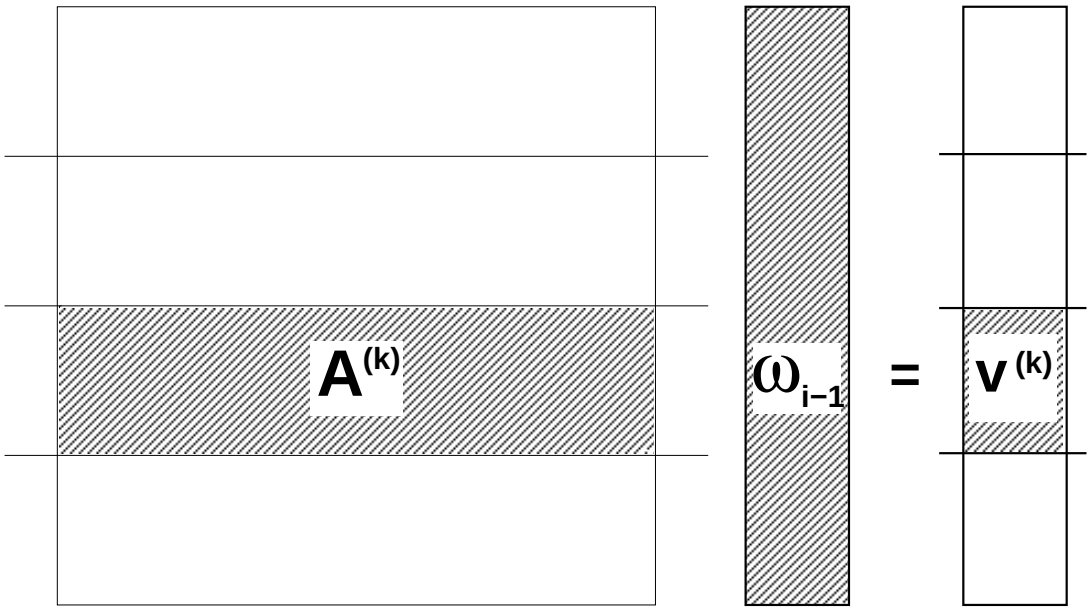


FIG. 4.3: *Produit matrice vecteur, sur le processeur k.*

La seule solution -sauf à rentrer dans un processus complexe de communications entre nœuds- consiste à dupliquer les vecteurs ω_i sur tous les processeurs. Si l'on fait cela, le produit matrice vecteur se trouve parfaitement parallélisé. Dupliquer quelques données semble acceptable mais pas toutes !, voyons si nos ennuis s'arrêtent là.

4.3.3 Impact sur le produit scalaire

Pour travailler sur un exemple, prenons le produit scalaire $< v, g_i >$ de l'étape `(7)`. Pour ce type d'opération, si l'on possède une tranche de chacun des vecteurs, il est possible de calculer sur chaque processeur k, une partie du produit scalaire qui se décompose en :

$$< v, g_i >= \sum_{k=0}^{Nproc-1} < v^{(k)}, g_i^{(k)} >$$

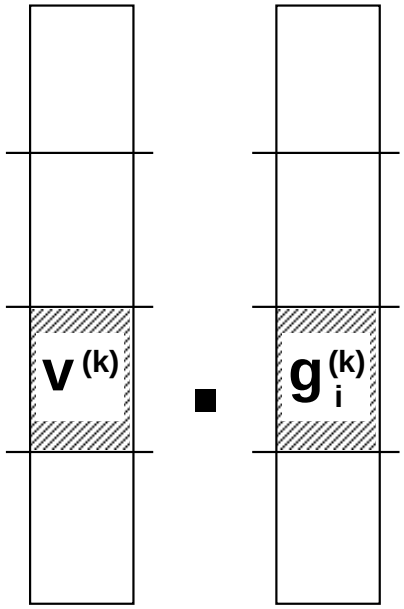


FIG. 4.4: *Produit scalaire, sur le processeur k.*

Il suffit donc d'effectuer, en parallèle sur tous les processeurs, les produits scalaires sur les "tranches" puis de faire la somme "globale" des résultats obtenus sur chaque processeur (il s'agit d'une opération de type réduction).

Ça se passe mieux ! le produit scalaire en parallèle permet de conserver seulement les "tranches" des vecteurs qui interviennent, il ne coûtera, en plus, qu'une réduction sur des variables scalaires.

4.3.4 Impact sur les combinaisons linéaires de vecteurs

De même, pour voir ce qui se passe, choisissons de travailler sur un exemple : l'étape `(6)` de l'algorithme : $g_i = g_{i-1} + \rho_i v$.

Ici, si l'on possède une "tranche" de chacun des vecteurs apparaissant à droite du signe d'affectation, l'on peut calculer la "tranche" correspondante du vecteur résultat, pourvu que l'on dispose sur chaque processeur des coefficients scalaires.

Or les variables en question (voir les étapes `(5)`, `(6)` et`(10)` sur la figure 4.1) sont les coefficients ρ_i et γ_i qui sont calculées à partir de produits scalaires que l'on sait effectuer en parallèle et reconstituer "globalement" (voir le paragraphe précédent).

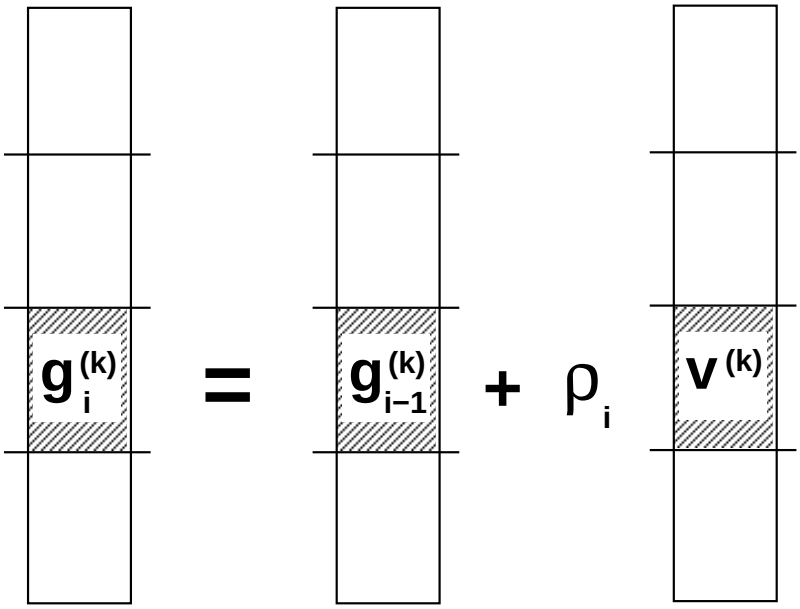


FIG. 4.5: *Combinaison linéaire, sur le processeur k.*

Ainsi ces opérations sont-elles parfaitement paralléles.

4.3.5 Reconstituer ω_i

On peut donc conserver dans les mémoires locales, les “tranches” correspondantes de chaque vecteur sauf pour ω_i qu’il faudra, à chaque itération, reconstituer en entier sur chaque processeur. Or l’étape `(9)` dans l’algorithme parallélisé pourra s’écrire :

$$\omega_i^{(k)} = g_i^{(k)} + \gamma_i \omega_{i-1}^{(k)}$$

Chaque processeur calculera donc une “tranche” de la nouvelle direction de descente. Pour obtenir le vecteur ω_i en entier, chaque processeur devra diffuser sa “tranche” aux autres processeurs. Tous pourront ensuite reconstituer ω_i, il s’agit de l’opération appelée `all_gather` dans la terminologie `MPI`.

4.3.6 L’algorithme parallèle

Chaque processeur possède une “tranche” $A^{(k)}$ de la matrice A, la portion correspondante de chacun des vecteurs v, g_i, et x_i soit $v^{(k)}$, $g_i^{(k)}$, et $x_i^{(k)}$ ainsi que le vecteur ω_i en entier.

Algorithme du processeur k

initialisation :

x_0	// donné et dupliqué
$g_0^{(k)} = Ax_0 - b^{(k)}$	// premier gradient
$\omega_0^{(k)} = g_0^{(k)}$	// première direction de descente
$all\_gather(\omega_0^{(k)})$	// reconstitution de ω_0

jusqu'a convergence (itération i) :

(1)	$v^{(k)} = A^{(k)}\omega_{i-1}$	
(2)	$sc^{(k)} =< v^{(k)}, \omega_{i-1}^{(k)} >$	// i.e. $< A\omega_{i-1}^{(k)}, \omega_{i-1}^{(k)} >$
(2.1)	$sc = réduction(sc^{(k)}, +)$	// somme globale $\Rightarrow sc$
(3)	$loc^{(k)} =< g_{i-1}^{(k)}, \omega_{i-1}^{(k)} >$	
(3.1)	$loc = réduction(loc^{(k)}, +)$	// somme globale $\Rightarrow loc$
(4)	$\rho_i = -loc/sc$	// coef. de descente
(5)	$x_i^{(k)} = x_{i-1}^{(k)} + \rho_i\omega_{i-1}^{(k)}$	// nouvel itéré
(6)	$g_i^{(k)} = g_{i-1}^{(k)} + \rho_i v^{(k)}$	// nouveau gradient
(7*)	$test\ de\ convergence$ **global**	// si négatif, on continue
(8)	$loc^{(k)} =< v^{(k)}, g_i^{(k)} >$	
(8.1)	$loc = réduction(loc^{(k)}, +)$	// somme globale $\Rightarrow loc$
(9)	$\gamma_i = -loc/sc$	// coef. de conjugaison
(10)	$\omega_i^{(k)} = g_i^{(k)} + \gamma_i\omega_{i-1}^{(k)}$	// direction de descente
(10.1)	$all\_gather(\omega_i^{(k)})$	// reconstitution de ω_{i+1}

FIG. 4.6: *1^er algorithme parallèle du gradient conjugué*

Les étapes supplémentaires (2.1), (3.1), (9.1), (10.1), correspondent aux opérations de communication nécessaires pour assurer que l'on converge bien vers la solution du problème global.

Le test de convergence étape (7*) doit bien sûr être global. Si l'on calcule sur chaque processeur un ϵ, il sera basé sur le **max** global des ϵ locaux (opération de réduction) ce problème a été détaillé dans un chapitre précédent (cf 2.3.3).

4.3.7 Bilan

Répartition des données

Dans cette approche, les données sont bien réparties sur les mémoires locales : seul un vecteur doit être dupliqué totalement sur tous les processeurs. Rappelons néanmoins que l'on stocke toute la matrice bien que celle-ci soit symétrique mais l'approche à l'avantage de se généraliser à tous types de matrice. Stocker seulement la moitié de la matrice ne pose pas de problèmes mais le produit matrice vecteur impose alors de nombreuses communications qu'il est possible d'optimiser, le reste de l'algorithme est alors inchangé.

Répartition des calculs

Si le découpage de la matrice est effectué en "tranches" régulières, la charge de calcul sera bien équilibrée sur les processeurs.

Surcoût lié aux communications

Les étapes de communications sont bien localisées. À chaque itération, il faut effectuer :

- une reconstition (`all_gather`) d'un vecteur global ;
- trois réductions sur des variables scalaires (pour les produits scalaires) ;
- enfin, il faut rajouter une réduction supplémentaire dans le test de convergence, pour le calcul d'un résidu global.

Toutes ces opérations sont déjà programmées de façon optimisées dans des bibliothèques de communication de type `MPI`.

Sur l'algorithme

Nous avons obtenu un programme de type `SPMD`, dérivé simplement de l'algorithme séquentiel. Cette approche est indépendante du nombre de processeur, elle "passe à l'échelle" dans la mesure où l'on conserve une granularité suffisante (le surcoût du aux communications ne devenant pas trop pénalisant).

Sur l'intérêt de cette approche

Si cette approche est efficace, elle n'est pas générale car nous nous sommes attaché à paralléliser uniquement un solveur. Cela implique que la matrice A du système a été assemblée avant d'être découpée. Nous passons donc sous silence la parallélisation éventuelle des étapes précédentes et notamment la phase très importante de l'assemblage.

4.4 Partition de domaine

4.4.1 Un mot sur l'assemblage

Regardons donc ce qui se passe au moment de l'assemblage. Pour effectuer cette opération il est nécessaire de posséder un tableau donnant la correspondance numéro local numéro global pour tous les nœuds du maillage.

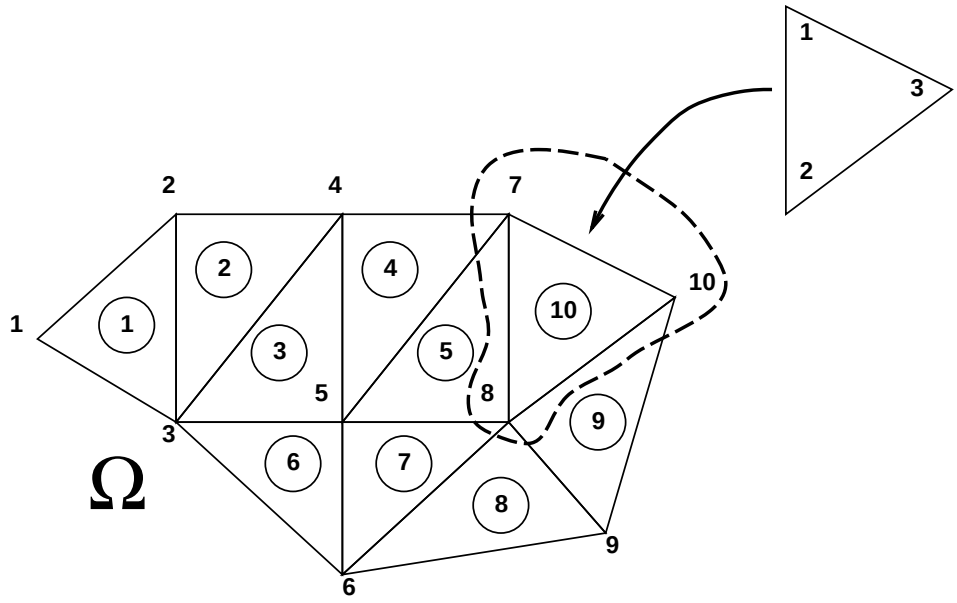


FIG. 4.7: *Correspondance "global → local"*

Dans l'exemple ci-dessus, les nœuds `1`, `2`, `3` de l'élément numéro `10` correspondent respectivement aux nœuds `7`, `8`, `10` du maillage de Ω. Appelons `loc_gl` ce tableau dont la première dimension est égale au nombre d'éléments du maillags. Nous obtenons pour la figure 4.7 :

- `loc_gl(10,1)=7`
- `loc_gl(10,2)=8`
- `loc_gl(10,3)=10`

Cette correspondance n'est, bien sûr, pas injective. À chaque fois que des éléments ont un nœud en commun, nous retrouverons le même numéro global. Il s'ensuit que l'algorithme séquentiel d'assemblage tel qu'il est décrit dans la figure 4.8 ci dessous n'est pas simplement parallélisable. Appelons `nb_el` est le nombre d'éléments du maillage, `nb_eloc` le tableau qui donne le nombre de sommets de chaque élément, `A` la matrice "globale" et `A_LOC` les matrices élémentaires :

```
pour nel=1, nb_el                          // éléments du maillage
 {
 pour j=1, nb_loc[nel]                     // i,j parcourent les sommets
  {                                        // de l'élément nel
  j_glob=loc_gl(nel,j)                      // j local ⇒ j global
  pour i=1, nb_loc[nel]
   {
    i_glob=loc_gl(nel,i)                    // i local ⇒ i global
    A(i_glob,j_glob)+=a_loc(i,j)            // += accumulation
   }
  }
 }
```

FIG. 4.8: *L'assemblage*

4.4.2 Paralléliser l'assemblage

Comment faire pour paralléliser l'assemblage. Il faut partager les éléments à traiter entre les processeurs. Si l'on travaille sur une machine `MIMD` à mémoire **partagée**, il nous faudra empécher les accès simultanés à un même élément de `A`. Cela peut se faire avec des techniques de "coloriage" des éléments qui permettent de gérer les conflits mais ce n'est pas le propos de ce document. En revanche, si la machine est à mémoire **distribuée**, nous pouvons facilement imaginer que la matrice sera *a priori* distribuée sur les mémoires locales et que les accès à un même élément seront réglés par des communications entre processeurs.

Les difficultés de mise en œuvre d'un tel "assemblage distribué" nous conduisent naturellement à envisager de ne pas assembler totalement la matrice. Après avoir partitionné le domaine de calcul, nous assemblerons "localement", sur chaque processeur, processus entièrement parallèle et sans communications, en reportant "plus loin" le problème du recollement des sous-domaines.

4.4.3 Découpage du domaine

Pour remonter le plus en amont possible, la parallélisation, nous allons découper le domaine de calcul Ω. Par exemple, sur deux processeurs, nous obtiendrions :

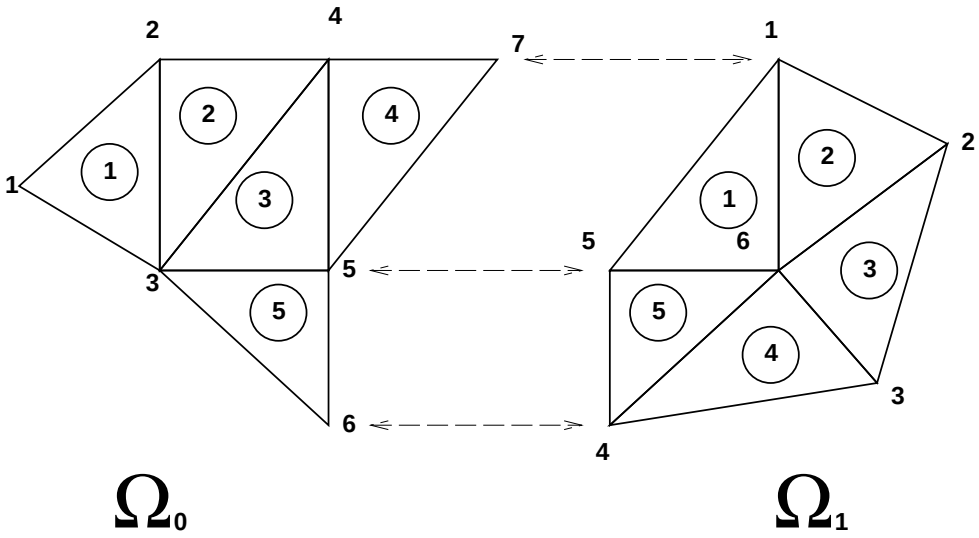


FIG. 4.9: *Partition du domaine* Ω

Les deux sous-domaines Ω_0 et Ω_1 sont traités au départ de façon indépendante. Ils ont chacun leur numérotation. Il n'y a plus de numérotation globale de Ω. Sur les deux processeurs, l'assemblage en local produit deux matrices locales A_0 et A_1. Si l'on voulait obtenir la matrice globale A il faudrait comme dans le paragraphe précédent assembler les matrices A_0 et A_1 en tenant compte d'une correspondance globale $\Rightarrow$ locale (tout cela se fait classiquement dans des codes effectuant de la "sous-structuration").

Observons la figure 4.10, le nœud Q est situé "strictement" dans un des sous-domaine, toute l'"information" nécessaire au calcul de la solution en ce point est contenu dans la matrice assemblée localement. En revanche, pour le nœud P situé Ìa frontière de deux sous-domaines, cette information est répartie dans plusieurs matrices locales. Ainsi, seules les inconnues aux points frontière donneront lieu à des communications dans la phase de résolution.

L'idée maintenant est d'appliquer l'algorithme du gradient conjugué, en parallèle sur chaque processeur, en utilisant la matrice assemblée localement et des vecteurs locaux. Comme dans les paragraphes précédents, nous allons examiner l'impact de cette stratégie sur les principales étapes de l'algorithme pour obtenir une solution au problème global.

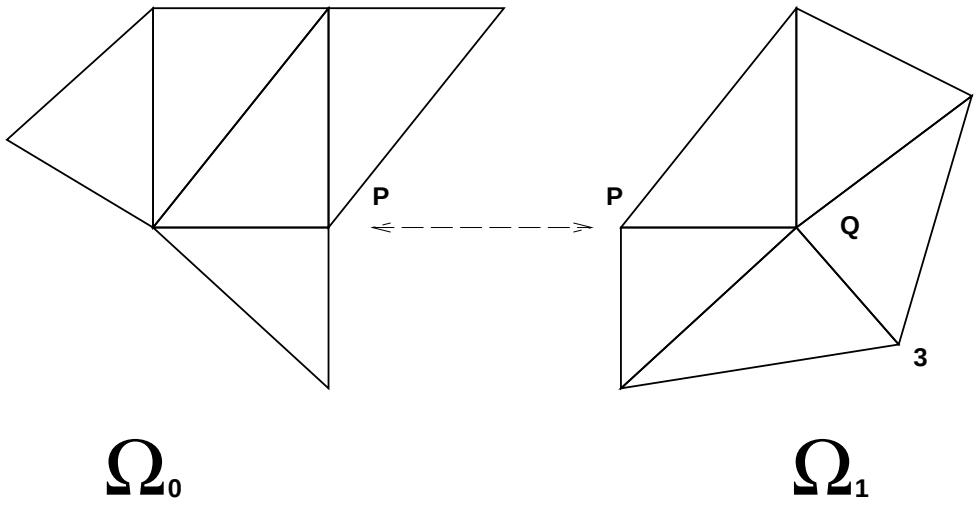


FIG. 4.10: *Point intérieur, frontière*

4.4.4 Impact sur le produit matrice-vecteur

Pour comprendre ce qui se passe, le plus facile est de traiter un exemple complet. Pour cela nous allons choisir un domaine Ω_s le plus simple possible :

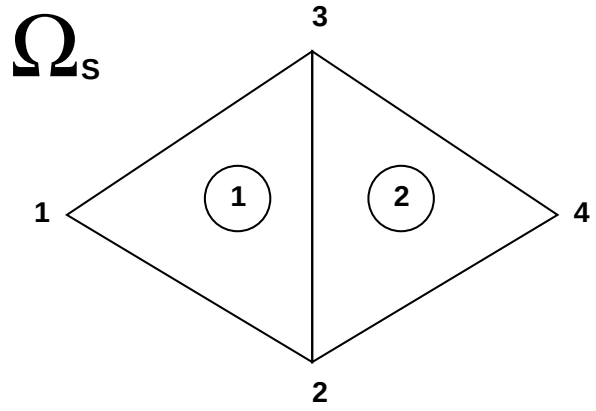


FIG. 4.11: *Un domaine "simple"* Ω_s

où le maillage est constitué de deux éléments. Nous découpons ce maillage en deux sous-domaines Ω_{s0} et Ω_{s1} :

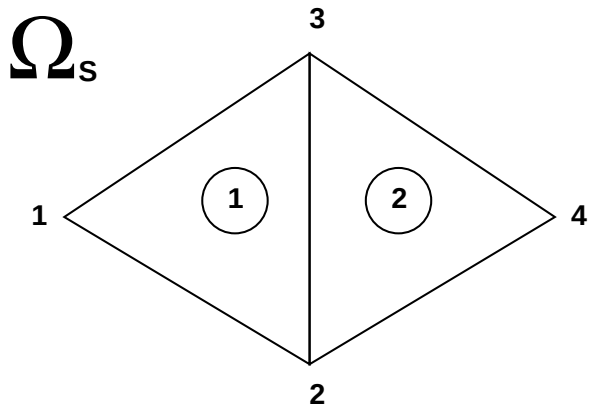


FIG. 4.12: *Partition de* Ω_s

Raisonnons sur un exemple, les matrices locales A_0 et A_1 résultat de l'assemblage en local (trivial ici mais cela n'enlève rien à la généralité de l'approche) ont, par exemple cette forme :

$$A_O = \begin{bmatrix} 1 & 3 & 0 \\ 3 & 2 & 4 \\ 0 & 4 & 1 \end{bmatrix} \quad A_1 = \begin{bmatrix} 3 & 1 & 0 \\ 1 & 4 & 2 \\ 0 & 2 & 1 \end{bmatrix}$$

L'assemblage de ces deux matrices en tenant compte du recollement des deux domaines -i.e. la correspondance "local $\rightarrow$ global"- **donnerait** la matrice A, mais rappel le but est de ne pas calculer A, ni même de générer une telle correspondance.

Notons que pour la contribution de A_1, les lignes et les colonnes sont permutées à cause de cette correspondance "local $\rightarrow$ global".

Contribution de A_0 : local $\rightarrow$ global $\begin{cases} 1 \longrightarrow 1 \\ 2 \longrightarrow 2 \\ 3 \longrightarrow 3 \end{cases}$

Contribution de A_1 : local $\rightarrow$ global $\begin{cases} 1 \longrightarrow 2 \\ 2 \longrightarrow 4 \\ 3 \longrightarrow 3 \end{cases}$

$$A = \begin{bmatrix} 1 & 3 & 0 & 0 \\ 3 & 5 & 4 & 1 \\ 0 & 4 & 2 & 2 \\ 0 & 1 & 2 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 0 & - \\ 3 & 2 & 4 & - \\ 0 & 4 & 1 & - \\ - & - & - & - \end{bmatrix} + \begin{bmatrix} - & - & - & - \\ - & 3 & 0 & 1 \\ - & 0 & 1 & 2 \\ - & 1 & 2 & 4 \end{bmatrix}$$

Étant donné un vecteur W défini sur Ω_s, nous considérons les traces W_0 et W_1 de ce vecteur sur les sous domaines Ω_{s0} et Ω_{s1}. Le problème est de reconstituer les traces du produit AW à partir des produits locaux A_0W_0 et A_1W_1. Choisissons donc un vecteur W, les traces W_0 et W_1 sur Ω_{s0} et Ω_{s1} sont obtenues, encore une fois, en utilisant l'hypothétique correspondance "local $\rightarrow$ global".

$$W = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \qquad W_0 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \qquad W_1 = \begin{bmatrix} 3 \\ 4 \\ 3 \end{bmatrix}$$

Comparons maintenant A_0W_0 et A_1W_1 et les traces du produit AW, $(AW)_0$ et $(AW)_1$. Nous obtenons :

$$A_0W_0 = \begin{bmatrix} 1 & 3 & 0 \\ 3 & 2 & 4 \\ 0 & 4 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 7 \\ 19 \\ 11 \end{bmatrix}$$

$$A_1W_1 = \begin{bmatrix} 3 & 1 & 0 \\ 1 & 4 & 2 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 3 \end{bmatrix} = \begin{bmatrix} 10 \\ 24 \\ 11 \end{bmatrix}$$

Puis :

$$AW = \begin{bmatrix} 1 & 3 & 0 & 0 \\ 3 & 5 & 4 & 1 \\ 0 & 4 & 2 & 2 \\ 0 & 1 & 2 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 7 \\ 29 \\ 22 \\ 24 \end{bmatrix}$$

et enfin avec la correspondance "local $\rightarrow$ global" :

$$(AW)_0 = \begin{bmatrix} 7 \\ 29 \\ 22 \end{bmatrix} \qquad (AW)_1 = \begin{bmatrix} 29 \\ 24 \\ 22 \end{bmatrix}$$

On peut voir sur cet exemple, qu'il suffit d'"assembler" les vecteurs A_0W_0 et A_1W_1 c'est à dire d'aditionner les contributions de chaque sous-domaine aux points situés sur les frontières, pour retrouver $(AW)_0$ et $(AW)_1$. Pour cela il n'est pas utile d'avoir effectué une numérotation globale du domaine Ω_S, il suffit de conserver les correspondances "local $\leftrightarrow$ local" lors de la construction du maillage directement en sous-domaine ou lors du découpage du maillage global de Ω_s.

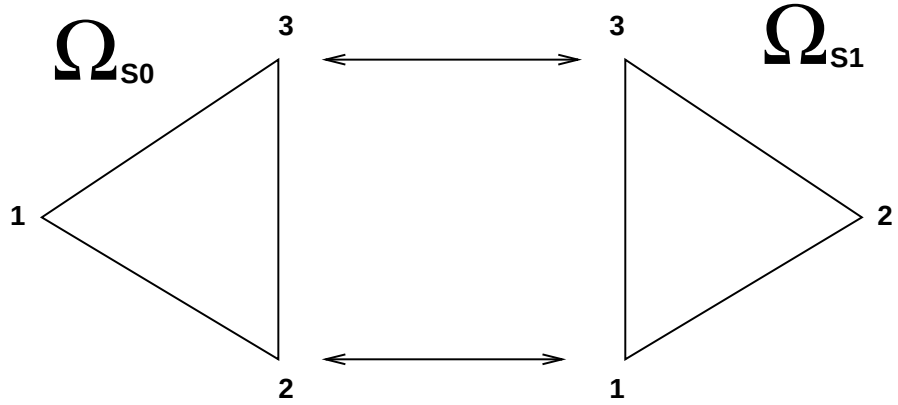


FIG. 4.13: *Correspondances "local ↔ local"*

Les tables (locales) de correspondance sont les suivantes :

Table pour Ω_{s0}			Table pour Ω_{s1}		
numéro	associé à	sous domaine	numéro	associé à	sous domaine
1	aucun		1	2	Ω_{s0}
2	1	Ω_{s1}	2	aucun	
3	3	Ω_{s1}	3	3	Ω_{s0}

Ces tables permettent, sur chaque processeur, de savoir où envoyer les informations nécessaires et...comment utiliser celles qui vont être reçues des autres processeurs.

Une fois l'échange effectué il suffit d'additionner, en local, les contributions des autres sous-domaines pour obtenir les traces du vecteur AW.

$$(AW)_0 = A_0W_0 + \begin{bmatrix} - & \\ 10 & \texttt{(le 1}^{\texttt{er}}\texttt{de } \Omega_{s1}\texttt{)} \\ 11 & \texttt{(le 3}^{\texttt{e}}\texttt{de } \Omega_{s1}\texttt{)} \end{bmatrix} = \begin{bmatrix} 7 \\ 29 \\ 22 \end{bmatrix}$$

$$(AW)_1 = A_1W_1 + \begin{bmatrix} 19 & \texttt{(le 2}^{\texttt{e}}\texttt{de } \Omega_{s0}\texttt{)} \\ - & \\ 11 & \texttt{(le 3}^{\texttt{e}}\texttt{de } \Omega_{s0}\texttt{)} \end{bmatrix} = \begin{bmatrix} 29 \\ 24 \\ 22 \end{bmatrix}$$

En conclusion, effectuer les produits en local puis la mise à jour donne le résultat recherché. Cette mise à jour peut sembler complexe mais en fait ces échanges se font selon des "patrons" fixes et déterminés -une fois pour toutes donc- dans la phase précédente de construction ou de découpage du maillage.

4.4.5 Impact sur le produit scalaire

De même qu'au paragraphe précédent, nous allons utiliser un exemple. Considérons encore le domaine Ω_s et le vecteur W défini sur ce domaine ainsi que ses traces W_0 et W_1 sur les sous domaines Ω_{s0} et Ω_{s1}. Comme précédement, nous allons chercher à retrouver le produit scalaire global à partir des produits scalaires locaux.

Effectuons le produit scalaire global :

$$< W, W >= \left\langle \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \right\rangle = 30$$

puis comparons avec les produits locaux :

$$< W_0, W_0 >= \left\langle \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \right\rangle = 14 \qquad < W_1, W_1 >= \left\langle \begin{bmatrix} 2 \\ 4 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 4 \\ 3 \end{bmatrix} \right\rangle = 29$$

Il est clair que contrairement à la première approche (cf 4.3.3), le produit salaire global n'est pas égal à la somme des produits locaux. Que s'est il passé ? eh bien, nous avons tout simplement compté deux fois les contributions des points frontière (composantes `2` et `3` pour Ω_{s0}, `1` et `3` pour Ω_{s1}).

Définissons alors pour chaque point de chaque sous domaine, un coefficient de pondération qui prend en compte le nombre de sous domaine auquel "appartient" ce point dans le maillage global. Dans nos exemple, ce coefficient est toujours égal :

- à `1`, le point est strictement contenu dans un sous domaine ;
- à `1/2`, le point est à la frontière des deux sous-domaines.

Dans des exemples plus complexes, un point pourrait être situé à la frontière de trois sous domaines (ou plus...) :

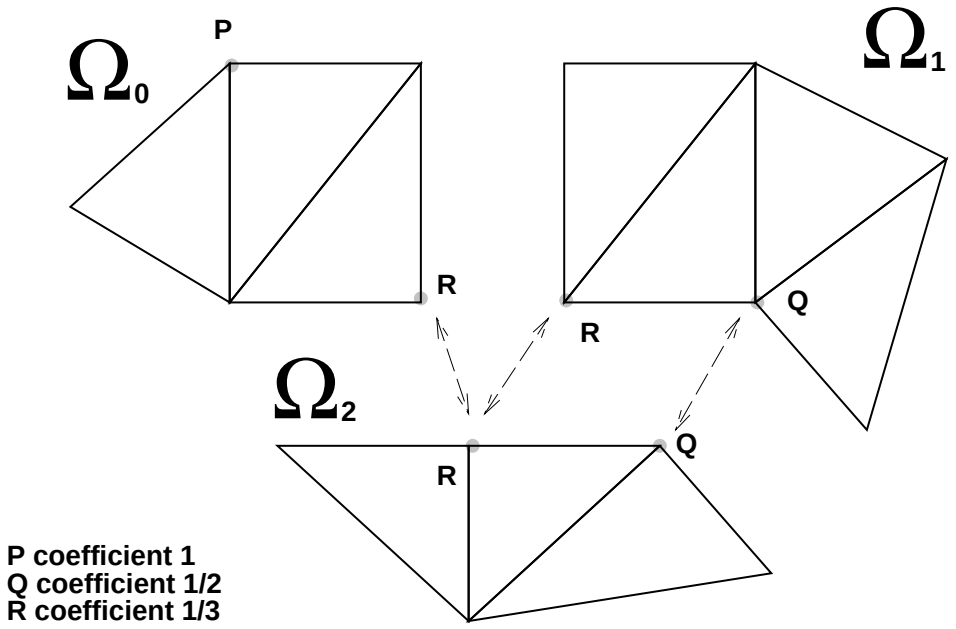


FIG. 4.14: *Coefficients de pondération*

L'ensemble des coefficients définit, pour chaque sous domaine, un vecteur de pondération. Si l'on considère de nouveau les domaines Ω_{s0} et Ω_{s1} de la figure 4.13, nous obtenons les vecteurs de pondérations suivants :

$$P_0 = \begin{bmatrix} 1 \\ 1/2 \\ 1/2 \end{bmatrix} \quad pour\ les\ inconnues\ de\ \Omega_{s0}$$

$$P_1 = \begin{bmatrix} 1/2 \\ 1 \\ 1/2 \end{bmatrix} \quad pour\ les\ inconnues\ de\ \Omega_{s1}$$

Il suffit maintenant de "pondérer" les produits scalaires en effectuant, sur chaque sous domaine k, l'opération $< P_k.W_k, W_k >$ au lieu de $< W_k, W_k >$. Nous désignons par $u.v$ la multiplication composante à composante de deux vecteurs u et v. Nous obtenons :

$$< P_0.W_0, W_0 >= \left\langle \begin{bmatrix} 1 \\ 1/2 \\ 1/2 \end{bmatrix} . \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \right\rangle = \left\langle \begin{bmatrix} 1 \\ 1 \\ 1.5 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \right\rangle = 7.5$$

et

$$< P_1.W_1, W_1 >= \left\langle \begin{bmatrix} 1/2 \\ 1 \\ 1/2 \end{bmatrix} . \begin{bmatrix} 2 \\ 4 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 4 \\ 3 \end{bmatrix} \right\rangle = \left\langle \begin{bmatrix} 1 \\ 4 \\ 1.5 \end{bmatrix}, \begin{bmatrix} 2 \\ 4 \\ 3 \end{bmatrix} \right\rangle = 22.5$$

Cette fois, le produit scalaire global est bien la somme des produits scalaires pondérés locaux. Il s'obtient en effectuant une réduction comme dans la première partie 4.3.3. Un exemple pour l'étape `(7)` de l'algorithme :

$$< v, g_i >= \sum_{k=0}^{Nproc-1} < p^{(k)}.v^{(k)}, g_i^{(k)} >$$

Dernière remarque, ces vecteurs de pondérations sont calculés dans les étapes initiales de construction ou de découpage du maillage global Ω_s en même temps que les tableaux de recollement local $\leftrightarrow$ local.

4.4.6 Impact sur les combinaisons linéaires de vecteurs

Ces opérations, elles, restent complètement locales. Si l'on dispose sur chaque processeur des coefficients scalaires identiques à ceux du problème global il n'y aura pas de correction à apporter au résultat. Or comme dans la première approche (cf 4.3.4) les variables en question (voir les étapes `(5)`, `(6)` et `(10)` sur la figure 4.1) sont les coefficients ρ_i et γ_i qui sont calculées à partir de produits scalaires que l'on sait effectuer en parallèle et reconstituer "globalement" (voir le paragraphe précédent).

4.4.7 L'algorithme parallèle

Chaque processeur possède une matrice assemblée localement $A^{(k)}$ et des vecteurs locaux $v^{(k)}$, $g_i^{(k)}$, $x_i^{(k)}$ et $\omega_i^{(k)}$, la dimension de ces vecteurs est celle de la matrice locale. Il n'y a plus aucune référence à des données globales. On suppose, en revanche, que l'on dispose des tableaux de correspondance local$\leftrightarrow$local et des vecteurs de pondération $p^{(k)}$.

Algorithme du processeur k

initialisation :

$x_0^{(k)}$	// donné
$v^{(k)} = A^{(k)} x_0^{(k)}$	
$mise\_a\_jour(v^{(k)})$	// => échanges de messages
$g_0^{(k)} = v^{(k)} - b^{(k)}$	// premier gradient
$\omega_0^{(k)} = g_0^{(k)}$	// première direction de descente

jusqu'a convergence (itération i) :

(1)	$v^{(k)} = A^{(k)} \omega_{i-1}^{(k)}$	
(1.1)	$mise\_a\_jour(v^{(k)})$	// => échanges de messages
(2)	$sc^{(k)} =< p^{(k)}.v^{(k)}, \omega_{i-1}^{(k)} >$	// $p^{(k)}$ pondération
(2.1)	$sc = réduction(sc^{(k)}, +)$	// somme globale $\Rightarrow sc$
(3)	$loc^{(k)} =< p^{(k)}.g_{i-1}^{(k)}, \omega_{i-1}^{(k)} >$	// $p^{(k)}$ pondération
(3.1)	$loc = réduction(loc^{(k)}, +)$	// somme globale $\Rightarrow loc$
(4)	$\rho_i = -loc/sc$	// coef. de descente
(5)	$x_i^{(k)} = x_{i-1}^{(k)} + \rho_i \omega_{i-1}^{(k)}$	// nouvel itéré
(6)	$g_i^{(k)} = g_{i-1}^{(k)} + \rho_i v^{(k)}$	// nouveau gradient
($7^{(*)}$)	$test\ de\ convergence$ **global**	// si négatif, on continue
(8)	$loc^{(k)} =< p^{(k)}.v^{(k)}, g_{i+1}^{(k)} >$	// $p^{(k)}$ pondération
(8.1)	$loc = réduction(loc^{(k)}, +)$	// somme globale $\Rightarrow loc$
(9)	$\gamma_i = -loc/sc$	// coef. de conjugaison
(10)	$\omega_i^{(k)} = g_i^{(k)} + \gamma_i \omega_{i-1}^{(k)}$	// direction de descente

FIG. 4.15: *2^e algorithme parallèle du gradient conjugué*

Les étapes supplémentaires (1.1), (2.1), (3.1), (8.1), correspondent aux communications nécessaires pour assurer que l'on converge bien vers la solution du problème global.

Comme dans le premier algorithme (cf figure 4.6), le test de convergence étape (7^*) doit bien sûr être global. Si l'on calcule sur chaque processeur un ϵ, il sera basé sur le **max** global des ϵ locaux (opération de réduction) ce problème a été détaillé dans un chapitre précédent (cf 2.3.3).

4.4.8 Bilan

Les données

Les données sont bien réparties sur les mémoires locales si le découpage du maillage fait apparaître des sous-domaines “équilibrés”. Un stockage supplémentaire est nécessaire pour la description des frontières :

– tableau de correspondance “local → local” ;
– vecteur de pondération.

La matrice assemblée localement est une “vraie matrice” et l’on peut bien sûr tirer parti du caractère symétrique pour optimiser le stockage.

Les calculs

Ils s’équilibrent si le découpage de maillage est bien effectué. Le surcoût est lié à l’utilisation des vecteurs de pondération pour les produits scalaires. En revanche, comme l’on manipule de “vraies” matrices, il est possible d’utiliser pour le produit matrice vecteur des outils standards et optimisés.

Surcoût lié aux communications

Les communications sont bien localisées. À chaque itération l’on trouve : :

– la mise à jour des vecteurs $v^{(k)}$ qui implique un échange d’information entre les processeurs. Cet échange peut être optimisé car il est toujours effectué selon le même “patron” en fonction des tables de correspondance précalculées “local → local” ;
– trois réductions sur des variables scalaires (pour les produits scalaires) ;
– il faut rajouter enfin une réduction supplémentaire dans le test de convergence, pour le calcul d’un résidu global.

Sur l’algorithme

Nous avons obtenu un programme de type `SPMD`, dérivé simplement de l’algorithme séquentiel. Cette approche est indépendante du nombre de processeur, elle “passe à l’échelle” dans la mesure où l’on conserve une granularité suffisante (le surcoût du aux communications ne devenant pas trop pénalisant).

Sur l’intérêt de cette approche

Cette approche est bien meilleure car elle prend en compte la parallélisation complète du problème. De plus elle se généralise assez facilement à un nombre quelconque de sous domaines.

4.5 Approche “sous domaine”

Dans le paragraphe précédent, nous avons appelé “partition de domaine” une approche consistant à utiliser la même méthode numérique sur des sous domaines. Nous réserverons le nom d’approche par sous domaines à des technique mettant en œuvre d’autres méthodes numériques sur le domaine de calcul.

4.5.1 Condenser sur la frontière

Comme toujours, nous allons considérer un domaine de calcul Ω que nous partitionnons en deux sous domaines Ω_0 et Ω_1. Mais cette fois-ci nous allons nous intéresser plus particulièrement à la frontière Γ_2 qui les sépare.

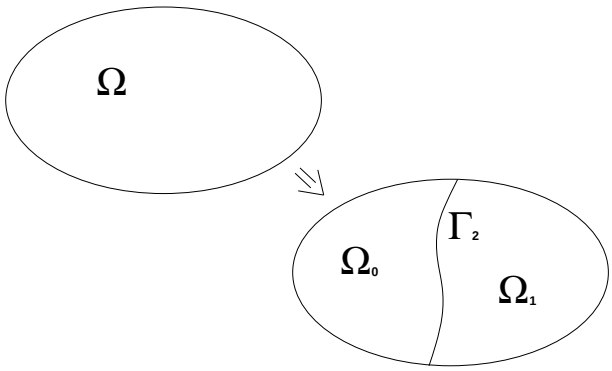


FIG. 4.16: *Frontière entre sous-domaines*

Une inconnue définie sur Ω_0 (resp. Ω_1) n’est en relation qu’avec d’autres inconnues de Ω_0 (resp. Ω_1) ou de la frontière Γ_2. Ainsi, si l’on assemble la matrice A du système après avoir **renuméroté les inconnues** de la façon suivante :

1. En premier, celles du sous domaine Ω_0.
2. Puis, celles du sous domaine Ω_1.
3. Et enfin, celles de la frontière Γ_2.

Nous obtenons une matrice qui a une structure par blocs :

$$A = \begin{bmatrix} A_{00} & 0 & A_{02} \\ 0 & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}$$

En particulier, nous avons fait apparaître deux blocs de zeros. Si $x^{(0)}$ représente les inconnues de Ω_0, $x^{(1)}$ celles de Ω_1 et $x^{(2)}$ celles de Γ_2, nous obtenons après avoir partitionné de même le vecteur second membre le système suivant :

$$\begin{bmatrix} A_{00} & 0 & A_{02} \\ 0 & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x^{(0)} \\ x^{(1)} \\ x^{(2)} \end{bmatrix} = \begin{bmatrix} b^{(0)} \\ b^{(1)} \\ b^{(2)} \end{bmatrix}$$

Si le problème est “bien posé” au départ, les matrices A_{00} et A_{11} sont réputées inversibles. Nous pouvons alors dans ce système exprimer $x^{(0)}$ et $x^{(1)}$ en fonction des inconnues sur la frontière $x^{(2)}$.
Soit le système suivant :

$$\begin{cases} A_{00}x^{(0)} + A_{02}x^{(2)} = b^{(0)} & (1) \\ A_{11}x^{(1)} + A_{12}x^{(2)} = b^{(1)} & (2) \\ A_{20}x^{(0)} + A_{21}x^{(1)} + A_{22}x^{(2)} = b^{(2)} & (3) \end{cases}$$

Les deux premières équations donnent :

$$\begin{aligned} x^{(0)} &= A_{00}^{-1}(b^{(0)} - A_{02}x^{(2)}) \quad (1.1) \\ x^{(1)} &= A_{11}^{-1}(b^{(1)} - A_{12}x^{(2)}) \quad (2.1) \end{aligned}$$

Si l’on sait calculer $x^{(2)}$ à l’aide de (3), l’on peut en déduire $x^{(0)}$ puis $x^{(1)}$ et ainsi reconstituer la solution du problème initial. C’est ce que l’on appelle condenser le système.

4.5.2 La méthode du complément de Schur

Nous allons présenter une méthode possible pour résoudre le problème condensé avec dans l’idée d’exhiber une méthode parallèle. Examinons plus particulièrement l’équation (3) du paragraphe précédent. Nous obtenons, en remplaçant $x^{(0)}$ et $x^{(1)}$ par leurs valeurs données dans (1.1) et (2.2) :

$$(A_{22} - A_{20}A_{00}^{-1}A_{02} - A_{21}A_{11}^{-1}A_{12})x^{(2)} = b^{(2)} - A_{20}A_{00}^{-1}b^{(0)} - A_{21}A_{11}^{-1}b^{(1)}$$

Soit en posant :

$$S = A_{22} - A_{20}A_{00}^{-1}A_{02} - A_{21}A_{11}^{-1}A_{12} \qquad \text{et}$$

$$f = b^{(2)} - A_{20}A_{00}^{-1}b^{(0)} - A_{21}A_{11}^{-1}b^{(1)}$$

$$Sx^{(2)} = f$$

S est appelée la matrice du complément de Schur. Quelles sont ses qualités et …ses défauts :

1. S est de petite taille par rapport à la matrice A initiale. Les inconnues $x^{(2)}$ étant définies sur la frontière, nous gagnons une dimension en espace.
2. L'on peut montrer que S est symétrique définie positive.
3. L'on peut aussi montrer que si le découpage est "bien effectué"[2] alors elle est bien conditionnée.
4. Mais S semble impossible à calculer !

L'idée bien sûr est de tenter de résoudre le système $Sx^{(2)} = f$ en appliquant la méthode du gradient conjugué puisque S semble présenter toutes les "bonnes propriétés. La convergence, compte tenu de ce qui précède, devrait être obtenue en un très petit nombre d'itérations. Malheureusement il n'est pas question de calculer S mais ce n'est pas forcément utile car l'on a seulement besoin de savoir calculer le produit de S par un vecteur c'est là que cette méthode devient intéressante car ce produit -très coûteux, l'on s'en doute- s'avère "parallèlisable".

Comme le système $Sx^{(2)} = f$ est de petite taille, nous allons ici le résoudre en même temps sur deux processeurs qui exécuterons donc le même code sur les mêmes données **sauf** au moment du produit matrice-vecteur, seule étape parallèle-où le travail sera partagé.

4.5.3 Parallélisation

Comme dans l'approche partition de domaine, nous allons dès le départ partitionner le domaine Ω en deux sous domaines Ω_0 et Ω_1. Il n'y aura pas de référence à des quantités globales.

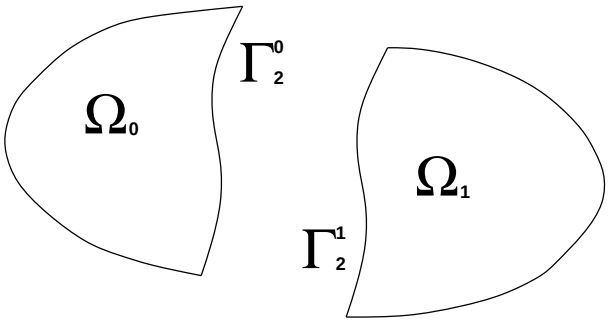


FIG. 4.17: *Partition du domaine*

Sur chaque sous domaine k, nous allons assembler la matrice locale après avoir renuméroté comme précédement les inconnues de la manière suivante :

[2]Notion difficilement quantifiable, voir l"'art du découpage" au début de ces notes.

1. Celles du sous domaine Ω_k.
2. Celles de la frontière $\Gamma_2^{(k)}$.

Nous obtenons sur chaque processeur (i.e. ici pour Ω_0 et Ω_1) :

$$A_0 = \begin{bmatrix} A_{00} & A_{02} \\ A_{20} & A_{22}^{(0)} \end{bmatrix} \quad \text{et} \quad A_1 = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22}^{(1)} \end{bmatrix}$$

Dans ces matrices, les blocs A_{00}, A_{02}, A_{20} et A_{11}, A_{12} et A_{21} sont identiques à ceux que l’on avait obtenus au paragraphe 4.5.1 sur le domaine global en renumérotant avant de condenser le système.

De plus, il est facile de voir que le bloc A_{22} “global” du paragraphe 4.5.1 s’obtiendrait par assemblage des deux blocs $A_{22}^{(0)}$ et $A_{22}^{(1)}$ c’est à dire, ici, en tenant compte de la manière dont les inconnues sont numérotées sur chaque sous domaine par simple addition.

$$A_{22} = A_{22}^{(0)} + A_{22}^{(1)}$$

La matrice S définie au paragraphe précédent, peut ainsi s’exprimer comme la somme de deux termes $S^{(0)}$ et $S^{(1)}$ que l’on peut calculer localement sur chaque processeur :

$$S = S^{(0)} + S^{(1)} \qquad \text{avec}$$

$$\begin{aligned} S^{(0)} &= A_{22}^{(0)} - A_{20}A_{00}^{-1}A_{02} \qquad \text{et} \\ S^{(1)} &= A_{22}^{(1)} - A_{21}A_{11}^{-1}A_{12} \end{aligned}$$

Ainsi effectuer du produit $Sx^{(2)}$ revient à calculer $S^{(k)}x^{(2)}$ sur chaque processeur k :

$$Sx^{(2)} = S^{(0)}x^{(2)} + S^{(1)}x^{(2)}$$

puis à effectuer la somme composante à composante des vecteurs ainsi obtenus sur chaque processeur, c’est à dire effectuer une opération de réduction.

4.5.4 Détail du calcul $S^{(k)}x^{(2)}$

Comment calculer le produit $S^{(k)}x^{(2)}$ sur chaque processeur ? Nous devons effectuer :

$$v^{(k)} = S^{(k)}x^{(2)} \text{ c'est à dire}$$
$$v^{(k)} = (A_{22}^{(k)} - A_{2k}A_{kk}^{-1}A_{k2})x^{(2)}$$

Pour cela posons d'abord :

$$loc^{(k)} = -(A_{kk}^{-1}A_{k2})x^{(2)}$$

ce qui revient à résoudre le système linéaire :

$$A_{kk}loc^{(k)} = -A_{k2}x^{(2)}$$

il suffit ensuite d'effectuer une combinaison linéaire de vecteurs :

$$v^{(k)} = (A_{k2}loc^{(k)}) + (A_{22}^{(k)}x^{(2)})$$

Avec la remarque du paragraphe précédent, A_{kk} est inversible et la résolution du système ne devrait pas poser de problème. Mieux ! comme l'on aura à résoudre ce système avec plusieurs second membres et que la dimension de A_{kk} n'est *a priori* pas trop grande (on a partitionné le domaine) l'on peut envisager de stocker A_{kk} sous forme (par exemple) profil et d'effectuer une factorisation "LU" de cette matrice. Les résolutions de système linéaires se ramèneront alors à de simples descentes remontées.

En conclusion, ces produits $Sx^{(2)}$ non seulement se parallélisent très bien mais de plus s'optimisent facilement sur chaque processeur.

4.5.5 L'algorithme parallèle

Chaque processeur possède une matrice assemblée localement $A^{(k)}$ et résoud dans un premier temps $Sx^{(2)} = f$ problème de petite taille car posé sur la frontière. La seule étape parallèle étant le calcul des produits $S^{(k)}x^{(2)}$. Le calcul des itérés $x_i^{(2)}$ et des autres quantités s'effectuera localement (et simultamément) sur chaque processeur.

Algorithme du processeur k

initialisation :

$x_0^{(2)}$	// donné
$v^{(k)} = produit(S^{(k)}, x_0^{(2)})$	// premier gradient
$g_0 = réduction(v^{(k)}, +)$	// somme globale $\Rightarrow g_0$
$\omega_0 = g_0$	// première direction de descente

jusqu’a convergence (itération i) :

(1)	$v^{(k)} = produit(S^{(k)}, \omega_{i-1})$	// voir description ci-dessus
(1.1)	$v = réduction(v^{(k)}, +)$	// somme globale $\Rightarrow v$
(2)	$sc =< v, \omega_{i-1} >$	// c’est à dire $< A\omega_i, \omega_i >$
(3)	$loc =< g_{i-1}, \omega_{i-1} >$	
(4)	$\rho_i = -loc/sc$	// coef. de descente
(5)	$x_i^{(2)} = x_{i-1}^{(2)} + \rho_i \omega_{i-1}$	// nouvel itéré
(6)	$g_i = g_{i-1} + \rho_i v$	// nouveau gradient
(7)	$test\ de\ convergence$	// si négatif, on continue
(8)	$loc =< v, g_i >$	
(9)	$\gamma_i = -loc/sc$	// coef. de conjugaison
(10)	$\omega_i = g_i + \gamma_i \omega_{i-1}$	// nouvelle dir. de descente

reconstitution de la solution sur le sous domaine k :

FIG. 4.18: *3^e algorithme parallèle du gradient conjugué*

Attention la partie gradient conjugué ne permet que de calculer $x^{(2)}$ sur la frontière séparant les deux sous domaines. Il faut ensuite sur chaque processeur k reconstituer la solution locale $x^{(k)}$.

En utilisant l’analyse du début (cf 4.5.1), il vient :

$$x^{(k)} = A_{kk}^{-1}(b^{(k)} - A_{02}x^{(2)})$$

Comme pour les produits $S^{(k)}x^{(2)}$, il y a un système linéaire à résoudre mais la matrice A_{kk} a été factorisée et cela ne coûte encore une fois qu’une descente-remontée. Cette dernière étape est parfaitement parallèle car la reconstitution se fait sur chaque processeur sans communication.

4.5.6 Bilan

Les données

Comme en “partition de domaine”, les données sont bien réparties sur les nœuds si le découpage du maillage est “équilibré”. De même, la matrice assemblée localement est une “vraie matrice” et nous pourrons tirer parti de la symétrie pour le stockage et utiliser les bibliothèques standards d’algèbre linéaire. Les données dupliquées sont celles nécessaire á la résolution sur chaque processeur du même problème, mais de petite taille, $Sx = f$.

Les calculs

Une itération de gradient sur la matrice de Schur coûte très cher, mais l’on se rassure en constatant que l’on parallélise parfaitement la plus coûteuse des étapes, le produit $Sx^{(2)}$ et que l’on effectue *a priori* peu d’itérations.

Surcoût lié aux communications

Une seule étape de communication à chaque itération, où nous effectuons une réduction sur un vecteur de “petite” taille (nombres d’inconnues sur la frontière).

Sur l’algorithme

Nous avons obtenu un programme de type SPMD, dérivé simplement de l’algorithme séquentiel. Le “passage à l’échelle” ne pose aucun problème pourvu que la granularité soit suffisante

Sur l’intérêt de cette approche

Comme dans la deuxième méthode (partition de domaine) cette approche est très bonne car elle prend en compte la parallélisation complète du problème initial. La démonstration a été effectuée ici sur deux sous domaines mais cette approche se généralise à un plus grand nombre de sous domaines.

4.6 En conclusion

Cette dernière partie justifie un des postulats de l’introduction : c’est bien en travaillant sur les aspects modélisation que l’on arrivera à concevoir des applications parallèles efficaces. La troisième version du gradient conjugué parallèle illustre une approche beaucoup plus générale de décomposition de domaines qui

ne s'applique pas seulement à la parallélisation d'une application unique. De nombreux problèmes de couplages faisant intervenir plusieurs codes de calcul sont d'abord posés comme des problèmes aux "interfaces" afin d'être résolus efficacement sur des calculateurs parallèles.

Annexe A

Quelques pointeurs WWW

Parallélisation par Directives

Un travail sur le code, à réserver *a priori* pour l'optimisation locale sur le nœud de calcul (tout comme la vectorisation sur le processeur) :

– OpenMP, un standard sur les machines MIMD à mémoire distribuée, *"Simple, Portable, Scalable SMP Programming"* :
  http ://www.openmp.org/

Bibliothèques d'échanges de messages

Deux bibliothèques très utilisées. Si MPI était plus orientée "calcul parallèle" et PVM "application distribuée", les différences tendent a s'amenuiser :

– MPI *"The Message Passing Interface standard"*
  http ://www-unix.mcs.anl.gov/mpi/
– PVM *"Parallel Virtual Machine"*
  http ://www.epm.ornl.gov/pvm/pvm_home.html/

Langages

La plupart des bibliothèques et outils sont utilisables en Fortran, C et C++. Il existe de nombreux langages de programmation parallèles dont certains sont simplement des extensions des langages les plus courants. Il y a des problèmes plus spécifiques avec Java qui est dès le départ un langage parallèle :

– *"The JavaNumerics page"*
  http ://math.nist.gov/javanumerics/

Bibliothèques numériques

Bibliothèques d'algèbre linéaire parallélisées (ex : `scalapack`), etc. . .consulter :

– `Netlib` *" a collection of mathematical software, papers, and databases"*
`http ://www.netlib.org/`

Programmation Objet

La programmation objet tend à s'imposer, lentement certe, en calcul scientifique, inutile de tout reécrire. . . :

– *"The Object-Oriented Numerics Page"* (surtout `C++`)
`http ://www.oonumerics.org/oon/`