Composition
Mineure Calcul Parallèle

E. Goubault

30 mars 1998

Introduction

Au cours d'une résolution numérique de l'équation des ondes par un schéma implicite, on a à trouver la solution d'un système d'équations linéaires $Ax = b$. Après un algorithme de type pivot de Gauss que l'on n'étudiera pas ici, on se retrouve avec un système triangulaire supérieur $Ux = b$. On se propose ici de programmer la résolution parallèle de ce dernier système sous diverses hypothèses.

Soit donc à résoudre le système triangulaire supérieur

$$Ux = b$$

avec,

$$U = \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} & \cdots & U_{1,n} \\ 0 & U_{2,2} & U_{2,3} & \cdots & U_{2,n} \\ & & & \cdots & \\ 0 & 0 & \cdots & 0 & U_{n,n} \end{pmatrix}$$

et $U_{i,i} \neq 0$ pour tout $1 \leq i \leq n$.

On procède par "remontée" c'est à dire que l'on calcule successivement,

$$\begin{array}{rcl} x_n & = & \frac{b_n}{U_{n,n}} \\ x_i & = & \frac{b_i - \sum_{j=i+1}^{n} U_{i,j} x_j}{U_{i,i}} \end{array}$$

pour $i = n-1, n-2, \cdots, 1$.

A) Remontée en mode SIMD

On écrit les équations précédentes en pseudo-code C de la façon suivante:

```
x[n]=b[n]/U[n,n];
for (i=n-1;i>=1;i--)
   {
      x[i]=0;
      for (j=i+1;j<=n;j++)
         L: x[i]=x[i]+U[i,j]*x[j];
      x[i]=(b[i]-x[i])/U[i,i];
   }
```

où `n` est la taille de la matrice U représentée par le tableau `U[i,j]`, où `x[i]` (`i` entre 1 et `n`) représente le vecteur "inconnu" x et `b[i]`, le vecteur second membre du système d'équations linéaires.

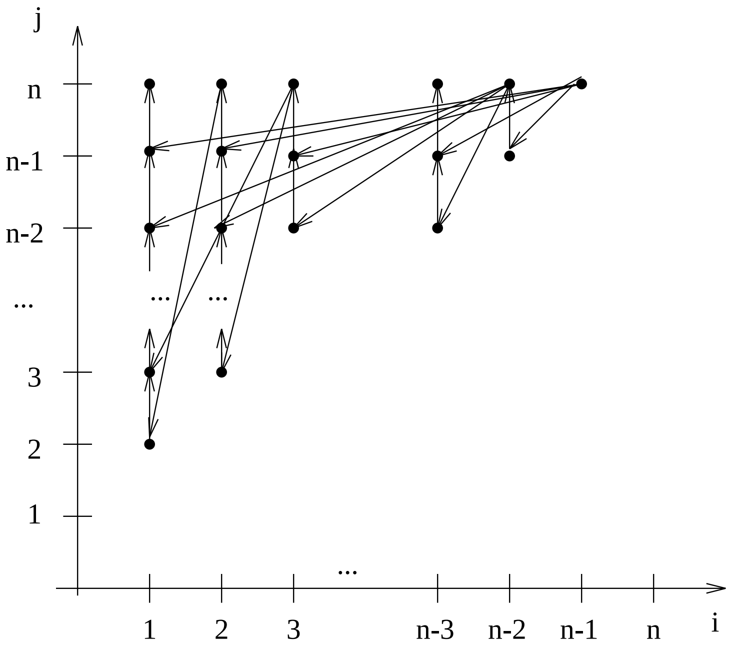

Figure 1: Le graphe de dépendances de la question A)1)

On suppose dans cette première partie que l'on dispose de $n-1$ processeurs.

Graphe de dépendance dans l'espace d'itérations

1) Quel est le graphe de dépendance dans l'espace d'itérations du code C dans les boucles imbriquées, calculant la remontée ? On rappelle que ce graphe est le graphe de dépendance dans l'espace des `(i,j)` entre les différentes instances du corps de la boucle `L`. De quel(s) type(s) de dépendance(s) s'agit-il ?

Correction Le graphe de dépendances est donné figure 1. Il s'agit essentiellement de dépendances de flot. On a aussi des anti-dépendances (qui n'ont absolument aucune importance en ce qui concerne la parallélisation et que l'on n'a donc pas représenté sur la figure) des itérations (i,j) vers $(i-1,j)$ (et ainsi de suite par transitivité).

Amélioration du code

Tel que le code séquentiel est écrit il parait difficile de le paralléliser en mode SIMD. On suppose que dans ce mode, on dispose d'une instruction `forall` qui a la même sémantique que l'instruction C `for` si ce n'est que le corps de boucle qui la suit, si ce corps n'a pas de dépendance pour les indices gérés par ce `forall`, est exécuté en parallèle et de façon synchrone sur chacun des processeurs.

2) Y a t-il une "rotation de code" (et autres transformations) que l'on peut effectuer qui permettrait de paralléliser le code séquentiel donné plus haut ? Reécrivez le code en conséquence.

Indication: On pourra comme dans le cas séquentiel commencer par calculer `x[n]`. Ensuite, l'idée sera de placer chacune des composantes des tableaux rentrant en compte pour le calcul de la ième équation linéaire ($1 \leq i \leq n-1$)[1] sur le ième processeur P_i. On pourra donc propager les valeurs déjà résolues des `x[j]`, `j`$\geq$`i+1` aux processeurs P_k, $k \leq$`i`, qui effectueront une même étape de sommation élémentaire.

Correction En faisant une rotation et une distribution de boucles (cf. cours) et en rajoutant des étiquettes pour faciliter la discussion sur la complexité,

```
H': forall (i=1;i<=n-1;i++)
       x[i]=b[i];
T': x[n]=b[n]/U[n][n];
H:  for (t=1;t<=n-1;t++)
       forall (i=1;i<=n-t;i++)
          L: x[i]=x[i]-x[n-t+1]*U[i][n-t+1];
       T: x[n-t]=x[n-t]/U[n-t][n-t];
```

Complexité

3) En supposant que le coût de chaque opération élémentaire (comme +, -, /, *) est unitaire ainsi que le coût d'un accès à un élément de tableau qui n'est pas sur la mémoire que gère le processus appelant, et en négligeant le coût d'initialisation des boucles, quel est le gain entre le code parallélisé et le code séquentiel du début?

Correction Si on ne regarde que la partie H du code (la seule vraiment importante asymptotiquement), on a,

- un coût de 2 pour `L`,
- un coût de 1 pour rapatrier `x[n-t+1]`

Comme nous sommes en code SIMD, tous ces calculs se recouvrent dans le `forall`. Ils sont donc à sommer pour chaque boucle sur `t`, tout comme le coût égal à un de `T`, donc un coût de $4(n-1)$ pour `H`. A cela, il faut ajouter le temps pour `H'` égal à un puisque nous sommes en mode SIMD et encore un pour `T'`. Donc un coût total de $4n-2$.

Pour le code séquentiel, si on compte l'affectation à zéro comme une opération élémentaire de coût unitaire, le coût total est,

$$1 + 3(n-1) + 2(1 + 2 + \cdots + n-1) = (n-1)(n+3) + 1$$

Le ratio est donc d'ordre $\frac{4}{n}$ asymptotiquement.

B) Remontée en mode MIMD

Dans cette partie, on veut rendre explicite les communications entre les processeurs qui gèrent chacun une partie du tableau `U`. De fait, on suppose que l'on a <u>n processeurs</u> $P_1, \cdots, P_n$ et que l'on répartit les tableaux de la façon suivante,

P_1	P_2	P_3	$\cdots$	P_{n-1}	P_n
$b[1]$	$b[2]$	$b[3]$	$\cdots$	$b[n-1]$	$b[n]$
$U[1,n]$ $U[1,n-1]$ $\cdots$ $U[1,1]$	$U[2,n]$ $U[2,n-1]$ $\cdots$	$U[3,n]$ $U[3,n-1]$ $\cdots$	$\cdots$ $\cdots$ $\cdots$	$U[n-1,n]$ $U[n-1,n-1]$	$U[n,n]$

[1] C'est celle qui sert à déterminer `x[i]` à partir des `x[j]` (`j`$\geq$`i+1`) lors de la remontée.

Code en PVM

1) Programmer le code de remontée correspondant en PVM (on pourra s'inspirer de ce qui a été décrit en A)2)). On supposera que P_n est le maître qui contient au début tout le tableau `U` et le tableau `b` et crée ses esclaves $P_1, \cdots, P_{n-1}$. A la fin il doit attendre le résultat de tous ses esclaves.

Correction En pseudo-code PVM (et sans etre précis sur les initialisations vis-à-vis du démon pvmd3 etc.) et avec des annotations pour simplifier la décomposition en tâches,

P_n est le maître qui a pour code,

```
T(n,1): x[n]=b[n]/U[n][n];
       /* tids est le tableau d'identificateurs */
       /* esclave est le code executable des autres processus */
       pvm_spawn("esclave",NULL,PvmTaskDefault,"",n-1,tids);
       for (i=1;i<=n-1;i++)
          {
             pvm_initsend(PvmDataDefault);
             pvm_pkint(&i,1,1);
             /* on envoie la colonne i */
             pvm_pkfloat(U[i],n,1);
             pvm_pkfloat(&(b[i]),1,1);
             pvm_pkfloat(&(x[n]),1,1);
             pvm_pkint(tids,n-1,1);
             /* TAG est le filtre d'envoi d'initialisation */
             pvm_send(tids[i-1],TAG);
          }
T(n,2): for (i=1;i<=n-1;i++)
           {
              /* RES est le filtre d'envoi des resultats */
              pvm_recv(tids[i-1],RES);
              pvm_upkfloat(&(x[i]),1,1);
           }
        /* le resultat est dans le vecteur x[] */
```

Les esclaves P_i ($1 \leq i \leq n-1$) ont tous le même code "esclave" suivant (avec des annotations pour simplifier la décomposition en tâches),

```
T(i,1): pvm_recv(pvm_parent(),TAG);
        pvm_upkint(&i,1,1);
        pvm_upkfloat(U[i],n,1);
        pvm_upkfloat(&(b[i]),1,1);
        pvm_upkfloat(&(x[n]),1,1);
        pvm_upkint(tids,n-1,1);
        x[i]=U[i][n]*x[n];

for (j=n-1;j>i;j--)
   {
T(i,2,j):  /* INTER est le filtre de communication entre esclaves */
           pvm_recv(tids[j],INTER);
           pvm_upkfloat(&(x[j]),1,1);
           x[i]=x[i]+U[i][j]*x[j];
   }
```

```
T(i,3): x[i]=(b[i]-x[i])/U[i][i];
        for (j=1;j<i;j++)
           {
              pvm_initsend(PvmDataDefault);
              pvm_pkfloat(&(x[i]),1,1);
              pvm_send(tid[j],INTER);
           }

T(i,4): pvm_initsend(PvmDataDefault);
        pvm_pkfloat(&(x[i]),1,1);
        pvm_send(pvm_myparent(),RES);
```

Dans toute la suite du problème, on supposera que le coût de l'envoi et le coût de la réception (hors attente) est égal à la taille des données envoyées multipliée par α $(0 \leq \alpha \leq 1)$ et ne dépend ni du processeur ni de l'historique des calculs.

Dépendance des tâches

Chacun des P_i contient donc un processus unique (que l'on peut aussi appeler P_i), qui va se charger de calculer $x[i]$.

2) Représentez le graphe de dépendances de tâches en y ajoutant les tâches de communication. Pour ce faire, on découpera le processus P_i en sous-tâches correspondant aux réceptions, sommations et envois de résultats partiels.

Correction Le graphe se trouve à la figure 2.

Complexité

3) Quel est le temps de calcul global du code PVM ? On supposera le coût de chaque opération élémentaire comme unitaire et on négligera tout ce qui est initialisation de boucle, `spawn` etc. Comment cela se compare-t-il avec A)3) ?

Correction On somme les coûts en tirant partie du graphe des tâches et en se souvenant que les `pvm_send` sont non-bloquants et les `pvm_recv` sont bloquants. On a donc les coûts par processus suivants, où on a représenté des chevauchements dans le temps (en supposant α très petit et n pas trop élevé),

P_n	P_{n-1}	P_{n-2}	$\cdots$	P_1
$T_{n,1} : 1 + 2\alpha(n-1)(n+1)$	$--$	$--$	$\cdots$	$--$
$--$	$T_{n-1,1} : 2\alpha(n+1)+1$	$T_{n-2,1} : 2\alpha(n+1)+1$	$\cdots$	$T_{1,1} : 2\alpha(n+1)+1$
$--$	$T_{n-1,3} : 2+\alpha(n-2)$	$--$	$\cdots$	$--$
$--$	$T_{n-1,4} : \alpha$	$T_{n-2,2,n-1} : \alpha+2$	$\cdots$	$T_{1,2,n-1} : \alpha+2$
$T_{n,2} : \alpha$	$--$	$T_{n-2,3} : 2+\alpha(n-3)$	$\cdots$	$--$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
α	$--$	$--$	$\cdots$	$--$

Donc la première ligne du tableau correspond à un coût global de $1+2\alpha(n-1)(n+1)$, puis les deux premières à $2+2\alpha(n+1)$, puis les quatre premières: $6+\alpha(2n(n+1)+n-1)$, puis si on somme les lignes deux par deux, on rajoute $2+\alpha(n-i)$ $(3 \leq i \leq n)$ et $\alpha+2$. On termine enfin par ajouter α (pour $T_{1,4}$) et α (pour $T_{n,2}$). Donc le coût global est,

$$6 + 4(n-2) + \alpha(2n(n+1) + (n-1) + (n-2) + \cdots + 1) + 2\alpha$$

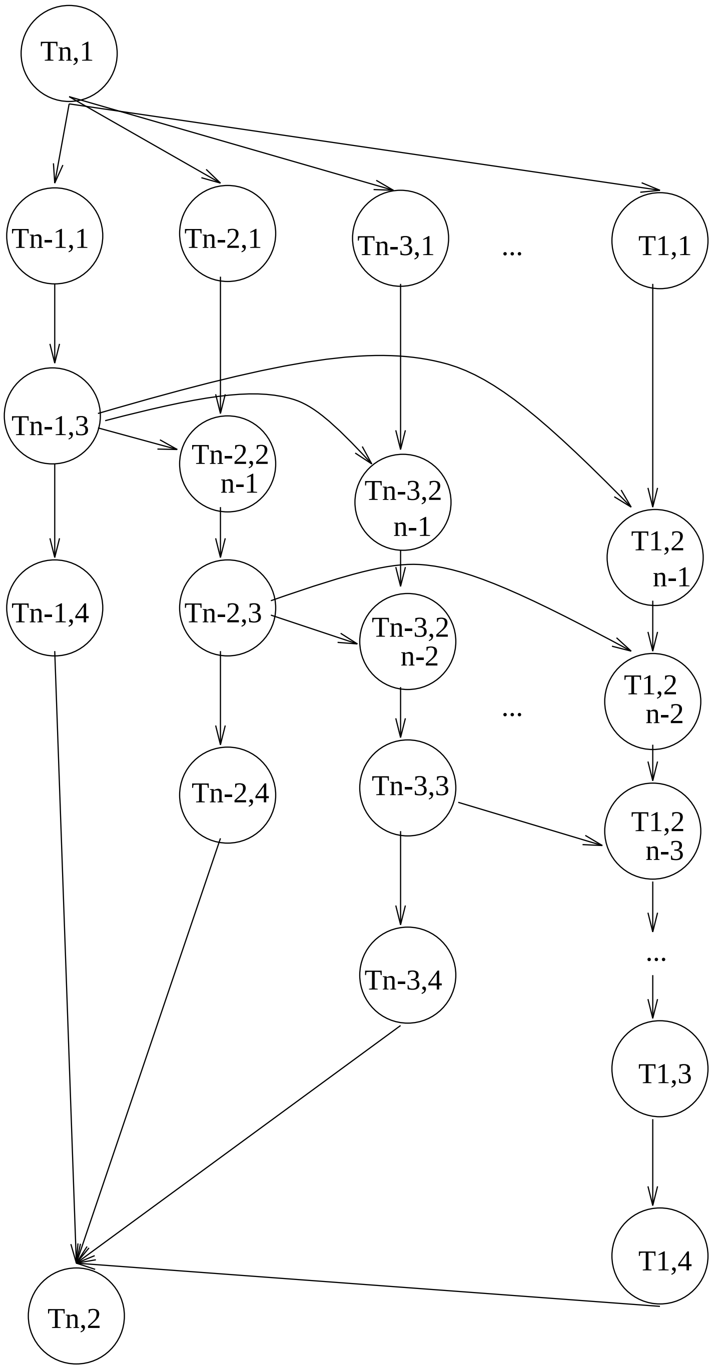

Figure 2: Graphe de dépendances pour la question B)2)

donc le coût est $4n - 2 + \frac{\alpha}{2}(5n^2 + 3n + 4)$.

Le coût est donc exactement celui du programme SIMD plus le coût explicite des communications. On aurait pu mieux ordonnancer les envois et gagner un peu sur ce temps supplémentaire, mais il serait resté en αn^2. C'est le coût des compactage et décompactage des données tel qu'il est programmé ici. On aurait eu un coût supplémentaire en αn si on avait utilisé des `broadcast`, et pour α petit on serait au même niveau de performance que le code SIMD, parce-que le code MIMD aurait été une transcription fidèle du code SIMD.

Ordonnancement dans le cas de deux processeurs

Le cas que nous avons étudié était idéal. On suppose dans cette question que l'on n'a pas autant de processeurs que de lignes dans la matrice U. Pour fixer les idées on choisit $p = 2$ processeurs. Les P_i apparaissant dans le tableau plus haut en début de la partie B) correspondent donc à des processus (ou des tâches).

4) Ordonnancez au mieux les tâches sur les 2 processeurs tout en respectant la localisation des données. On pourra s'inspirer de la méthode du cours (après avoir attribué des coûts aux sous-tâches) ou essayer a posteriori de justifier en quelques lignes l'optimalité de son choix.

Correction Un des nombreux ordonnancements possibles est

P_1	P_2
$T_{n,1}$	$--$
$T_{n-1,1}$	$T_{n-2,1}$
$\cdots$	$\cdots$
$\cdots$	$T_{1,1}$
$T_{n-1,3}$	$--$
$T_{n-2,2,n-1}$	$T_{n-3,2,n-1}$
$T_{n-4,2,n-1}$	$T_{n-5,2,n-1}$
$\cdots$	$\cdots$
$\cdots$	$T_{1,2,n-1}$
$T_{n-2,3}$	$--$
$T_{n-3,2,n-2}$	$T_{n-4,2,n-2}$
$\cdots$	$\cdots$
$\cdots$	$T_{1,2,n-2}$
$\cdots$	$\cdots$
$T_{2,3}$	$--$
$T_{1,2,2}$	$--$
$T_{1,3}$	$--$
$T_{1,4}$	$--$
$T_{n,2}$	$--$

Le taux d'occupation est en général excellent, si ce n'est à la toute fin du code.

On pouvait aussi appliquer le deuxième algorithme d'ordonnancement du cours, car nous sommes dans le cas 2 processeurs et le graphes de tâches est un graphe dirigé acyclique (bien que les différentes tâches n'aient pas de coût unitaire). On trouve alors les niveaux suivants:

$T_{n,2}$	1
$T_{n-1,4}$	2
$T_{n-2,4}$	3
$\cdots$ $T_{1,4}$	$\cdots$ $n+1$
$T_{1,3}$	$n+2$
$T_{1,2,2}$	$n+3$
$T_{2,3}$	$n+4$
$T_{1,2,3}$	$n+5$
$T_{2,2,3}$	$n+6$
$T_{1,2,4}$	$n+7$
$T_{3,3}$	$n+8$
$\cdots$	$\cdots$

qui donne un ordonnancement semblable (en terme de taux d'occupation des processeurs).

Ordonnancement statique

5) On suppose ici que la matrice U a $n = kp$ lignes (où $k \geq 2$ et p est le nombre de processeurs).

a) Sans écrire le code, justifier de la complexité (temps global d'exécution) de la solution qui consiste à répartir les lignes de `U` sur les processeurs modulo `p` (donc les lignes `U[1,*]`, `U[1+p,*]`, $\cdots$, `U[1+(k-1)*p,*]` sur P_1 etc.).

Correction Le nouveau code agit comme si on avait recopié p fois le code du maître et p fois le code des esclaves. Cela ne change rien à la complexité de l'algorithme, si on a bien fait attention à la gestion des "`broadcast`" vers les processus de numéro plus élevés.

b) On met maintenant des lignes adjacentes de `U` sur chacun des processeurs, c'est à dire que l'on découpe `U` en bandes horizontales. Ecrire un squelette de code PVM permettant de gérer cette décomposition de domaines. Comment faut-il choisir au mieux les tailles des domaines ? Comparer avec 5)a).

Correction Si on suppose que l'on fait gérer les lignes j, $1 \leq k_i \leq j \leq l_i \leq n$ au processeur i ($1 \leq i \leq n$), et que $i' > i$ implique $k_{i'} > l_{i'}$, plus $k_{i+1} = l_i + 1$, $k_1 = 1$, $l_p = n$. On a le squelette suivant, pour le maître (P_p),

```
/* resolution pour les inconnues x[k[p]] a x[n] */
T(p,1):...

        /* tids est le tableau d'identificateurs */
        /* esclave est le code executable des autres processus */
        pvm_spawn("esclave",NULL,PvmTaskDefault,"",n-1,tids);
        for (i=1;i<=n-1;i++)
           {
              pvm_initsend(PvmDataDefault);
              pvm_pkint(&i,1,1);
              /* on envoie la colonne i */
              pvm_pkfloat(U[i],n,1);
              pvm_pkfloat(&(b[i]),1,1);
              pvm_pkfloat(&(x[k[p]]),n-k[p]+1,1);
              /* on envoie k_i et l_i mis dans un tableau k[] et l[] */
              pvm_pkint(k,p,1);
              pvm_pkint(l,p,1);
```

```
            pvm_pkint(tids,n-1,1);
            /* TAG est le filtre d'envoi d'initialisation */
            pvm_send(tids[i-1],TAG);
         }
T(n,2): for (i=1;i<=n-1;i++)
          {
             /* RES est le filtre d'envoi des resultats */
             pvm_recv(tids[i-1],RES);
             pvm_upkfloat(&(x[k[i]]),l[i]-k[i]+1,1);
          }
        /* le resultat est dans le vecteur x[] */
```

Les esclaves P_i ($1 \leq i \leq p-1$) ont tous le même code “esclave” suivant,

```
T(i,1): pvm_recv(pvm_parent(),TAG);
        pvm_upkint(&i,1,1);
        pvm_upkfloat(U[i],n,1);
        pvm_upkfloat(&(b[i]),1,1);
        pvm_upkfloat(&(x[n]),1,1);
        pvm_upkint(k,p,1);
        pvm_upkint(l,p,1);
        pvm_upkint(tids,n-1,1);

/* initialisation des variables x[k[i]] a x[l[i]] */
...

for (j=p-1;j>i;j--)
   {
T(i,2,j):  /* INTER est le filtre de communication entre esclaves */
          pvm_recv(tids[j],INTER);
          pvm_upkfloat(&(x[k[j]]),l[j]-k[j]+1,1);
          ...
          /* resolution ... */
   }

T(i,3): for (j=1;j<i;j++)
          {
             pvm_initsend(PvmDataDefault);
             pvm_pkfloat(&(x[k[i]]),l[i]-k[i]+1,1);
             pvm_send(tid[j],INTER);
          }

T(i,4): pvm_initsend(PvmDataDefault);
        pvm_pkfloat(&(x[k[i]]),l[i]-k[i]+1,1);
        pvm_send(pvm_myparent(),RES);$i' > i$
```

Tel que le code est écrit, le mieux est de découper en domaines avec un même nombre de lignes, donc $k_i = 1 + k(i-1)$ et $l_i = ki$, car chacun des esclaves met le même temps (si on suppose que tous les processeurs ont la même puissance) de calcul. Si les puissances sont différentes et connues, on peut répartir en fonction des rapidités relatives.

Une meilleure solution consiste à envoyer `x[k[i]]` sitôt calculé, aux esclaves $P_{i'}$, $i' > i$, puis `x[k[i]+1]` jusqu'à `x[l[i]]` sitôt calculés. Cela permet de recouvrir bien mieux les calculs et les communications que dans la solution précédente. De même les domaines (à puissance égale de processeur) ont intérêt à être de mêmes tailles. C'est la bonne extension du code MIMD du B)1) dans le cas (courant) où on a plus de lignes que de processeurs, qui sera en général meilleure que l'algorithme du 5)a).

6) Question facultative: Donnez un principe, en quelques phrases, qui permettrait de faire de l'équilibrage de charges dynamique en PVM pour cette remontée (par exemple, si on avait autorisé d'autres applications indépendantes à consommer aléatoirement des ressources calcul pendant l'exécution du code).

Correction On peut programmer un algorithme du type client/serveur comme celui que l'on a vu en cours. Un serveur s'occupe de distribuer des lignes avec leurs inconnues à des clients. Les clients renvoient les résultats partiels au serveur et demandent de nouvelles lignes et inconnues non résolues. Cela permet un équilibrage de charge dynamique.