HAL
archives-ouvertes.fr

Factorisation parallèle de Cholesky pour matrices creuses sur une mémoire virtuelle partagée

Jocelyne Erhel, Mounir Hahad, Thierry Priol

▶ **To cite this version:**

Jocelyne Erhel, Mounir Hahad, Thierry Priol. Factorisation parallèle de Cholesky pour matrices creuses sur une mémoire virtuelle partagée. [Rapport de recherche] RR-1988, INRIA. 1993. <inria-00074684>

HAL Id: inria-00074684
https://hal.inria.fr/inria-00074684

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Factorisation parallèle de Cholesky pour matrices creuses sur une mémoire virtuelle partagée

Jocelyne Erhel, Mounir Hahad, Thierry Priol

N° 1988

Juillet 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

ISSN 0249-6399

INRIA
RENNES

Factorisation parallèle de Cholesky pour matrices creuses sur une mémoire virtuelle partagée

Jocelyne Erhel, Mounir Hahad, Thierry Priol

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet PAMPA

Rapport de recherche n ° 1988 — Juillet 1993 — 49 pages

Résumé : Nous abordons dans ce rapport différentes approches de la factorisation de Cholesky pour matrices creuses. Des algorithmes pour multiprocesseurs MIMD sont présentés, aussi bien pour un modèle de programmation par envoi de messages que par variables partagées. Notre but est d'analyser le comportement du système de gestion de mémoire virtuelle partagée (MVP) *KOAN* vis à vis d'algorithmes à accès semi-irréguliers aux données. De nouvelles approches sont proposées pour traiter ce type d'algorithmes, essentiellement en matière de synchronisation. Des résultats expérimentaux sont commentés.

Mots-clé : mémoire virtuelle partagée, matrices creuses, factorisation parallèle de Cholesky, ordonnancement de tâches, KOAN.

(Abstract: pto)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Parallel sparse Cholesky factorization on a shared virtual memory

Abstract: This paper addresses the problem of factoring large sparse matrices using the Cholesky factorization. Several approaches for MIMD architectures are depicted in both the message passing model and the shared variables one. Our goal is to analyze the behavior of the *KOAN* shared virtual memory with algorithms that exhibit semi-irregular data access patterns. A new parallel approach is presented and experimental results are provided.

Key-words: Sparse Cholesky factorization, shared virtual memory, KOAN, sparse matrices, scheduling.

Table des matières

Introduction

De nos jours, le calcul scientifique "hautes performances" couvre un large domaine d'applications. Plusieurs catégories peuvent en être distinguées selon la structure et l'évolution des calculs. Nous nous intéressons plus particulièrement aux applications en dynamique des fluides et calcul de structures, fondées sur une analyse par éléments finis ou différences finies. Généralement deux phases distinctes composent ces applications: une première phase de formulation du problème par assemblage de matrices, et une seconde phase de résolution d'un système d'équations. Afin de résoudre ce type de problèmes, des outils puissants sont mis à la disposition des chercheurs et des industriels. Toutefois, et dans le but de couvrir le besoin en puissance de calcul croissant sans cesse au fil des années (la taille des problèmes l'exige), le recours aux calculateurs parallèles semble être un passage obligé.

L'évolution des supercalculateurs a donné naissance tout récemment à une nouvelle génération de machines parallèles, à savoir les multiprocesseurs à mémoire virtuelle partagée (MVP), tels que la *KSR1* de *Kendal Square Research* (MVP mise en œuvre par dispositif matériel) et la *Paragon* d'*Intel* (MVP mise en œuvre par dispositif logiciel annoncé). Ceux-ci accumulent les avantages d'une modularité, et d'une facilité de programmation prônée par les habitués des machines à mémoire partagée. En effet, la distribution physique de la mémoire à travers les nœuds de calcul facilite la conception d'une architecture extensible, et la vision offerte au programmeur d'un espace d'adressage global unique décharge l'utilisateur de l'opération de distribution des données, et des conséquences qu'elle induit (gestion explicite des mouvements de données). Il reste à démontrer qu'en termes de performances, ces architectures n'ont rien à envier à leurs prédécesseurs.

Nous entamons donc une étude critique du comportement d'une mémoire virtuelle partagée (MVP) vis-à-vis d'algorithmes à accès "semi-irrégulier" aux données. Dans cette catégorie d'algorithmes, le schéma d'accès aux données ne peut être connu qu'à l'exécution du programme et au vu des données en entrée. Ceci limite d'ores et déjà les capacités d'un compilateur-paralléliseur à générer un code efficace. Cet aspect des choses ne sera pas abordé dans ce travail, et nous nous limiterons à une approche plus classique de la programmation parallèle.

Pour ce faire, les traitements effectués sur des matrices creuses semblent être une plate-forme intéressante. En particulier, nous choisissons une mé-

thode directe de factorisation de matrices creuses, la factorisation de *Cholesky*, pour illustrer notre discussion. Cette application se caractérise par des accès à une structure de données compactées, aussi bien en lecture qu'en écriture. De bonnes performances sont difficilement atteignables, à cause notamment des dépendances de nature non structurée, de la difficulté inhérente à l'équilibrage des charges de calcul et de communication ainsi que du coût élevé engendré par la manipulation de structures de données complexes. Il est important de savoir que les résultats en performances dépendent fondamentalement de la nature des données traitées, et que les bons résultats sont obtenus au prix d'efforts très importants. Nous montrerons que des performances satisfaisantes peuvent être facilement obtenues, en exploitant une mémoire virtuelle partagée.

L'expérimentation sera menée sur un hypercube *iPSC/2* muni du mécanisme de mémoire virtuelle partagée *KOAN* développé à l'Irisa [10]. Ce dispositif dédie 1,5 Mo de mémoire locale de chaque nœud à la mémoire virtuelle partagée, ce qui est une limitation non négligeable quant à la taille des problèmes qui pourront être traités.

1 Factorisation de Cholesky

Soit à résoudre un système d'équations linéaires, écrit sous forme matricielle :

$$Ax = b,$$

A étant une matrice carrée $n \times n$ symétrique définie positive, et b un vecteur de n composantes connues.

Deux classes de méthodes permettent de résoudre ce système linéaire : d'une part les méthodes *directes* qui donnent une solution exacte au problème, et d'autre part, les méthodes *itératives* qui donnent une approximation de la solution exacte.

La factorisation de Cholesky, qui est une méthode directe, permet d'écrire A sous la forme :

$$A = LL^T.$$

où le facteur L est une matrice triangulaire inférieure à éléments diagonaux positifs. Le système linéaire se réécrit sous la forme :

$$LL^T x = b,$$

et peut donc être traité par la résolution successive des deux systèmes triangulaires :

$$Ly = b, \ puis \ L^T x = y.$$

Le processus de factorisation procède par transformations successives de la matrice d'origine A pour aboutir au facteur de Cholesky L, en un nombre fini d'étapes.

Lorsqu'il s'agit de factoriser des matrices creuses, il faut conserver le plus possible l'aspect creux de la matrice A au fur et à mesure que la factorisation progresse. Cet objectif est essentiel pour la sauvegarde de l'espace mémoire et la minimisation du nombre de calculs à effectuer.

Dans les mises en œuvre séquentielles, la factorisation de Cholesky est décomposée en trois étapes de base :

1. renumérotation ,
2. factorisation symbolique ,
3. factorisatin numérique ;

que nous allons décrire dans les paragraphes suivants.

1.1 Renumérotation

Durant le processus de factorisation, l'élimination d'une variable au niveau d'une ligne peut entraîner, et c'est souvent le cas, un *remplissage* (un élément nul dans A devient non nul dans L) au niveau des autres lignes. Outre les avantages déjà cités (espace mémoire restreint et nombre de calculs réduit), plus la matrice est creuse, plus on a de chance d'avoir du parallélisme. Aussi, nous recherchons au cours de cette étape une nouvelle renumérotation des ligne de A qui permet de minimaliser le remplissage dans L. Afin de préserver la symétrie, toute permutation de ligne doit entraîner une permutation de colonne telle que les éléments diagonaux restent sur la diagonale. Il est à noter que nous ne prenons pas en considération les annulations éventuelles d'éléments de A dues au calcul, vu que celles-ci sont imprévisibles.

En d'autres termes, sachant que pour toute permutation P, la matrice PAP^T reste symétrique définie positive, nous pouvons choisir une permutation qui minimalise le remplissage de L, puis appliquer la factorisation de

Cholesky sur la matrice PAP^T. Malheureusement, trouver la bonne permutation est un problème complexe qui nécessite le recours à des heuristiques.

Les algorithmes de renumérotation les plus courants reposent sur des principes de la théorie des graphes. Aussi allons-nous en introduire quelques notions.

Définition 1 *Soit A une matrice symétrique d'ordre n. Le graphe G = (X,E) associé à A est un graphe non orienté tel que :*

$X = \{1,2,\ldots,n\}$, *sommets correspondant chacun à une ligne de la matrice ;*

$E = \{(i,j) \in X^2 \; i > j \; / \; A_{ij} \neq 0\}$.

□

Soit $G = (X, E)$ le graphe d'une matrice A et soit x un sommet de G.

Définition 2 *Le degré de x est défini par le nombre de voisins de x dans G. Il représente le nombre d'éléments non nuls dans la colonne x au dessous de la diagonale de A.*

□

L'élimination de GAUSS d'une variable i au niveau du système d'équations revient à :

1. supprimer le sommet i de G ;
2. supprimer tous les arcs incidents au sommet i ;
3. relier par un arc toute paire de sommets appartenant à l'ensemble des voisins de i et non eux-même voisins.

L'algorithme décrit ci-dessus permet le passage, par transformations successives, du graphe associé à la matrice A à celui associé à la matrice L. Le troisième point de cette procédure correspond au remplissage de L, et c'est ce que nous cherchons à minimiser par renumérotation de A. Pour ce faire, nous introduisons les deux algorithmes les plus utilisés.

1.1.1 Algorithme du degré minimal

Le principe de cet algorithme est de dériver une suite de graphes $G_0 \ldots G_{n-1}$ de la façon suivante :

1. $G_0 = G(X_0, E_0)$ = G(X,E) associé à A ;
   Pour $i = 1$ à $n - 1$ faire :
2. trouver $x \in X_{i-1}$ tel que $\forall y \in X_{i-1}$, $deg(x) \leq deg(y)$;
3. associer le numéro i à x ;
4. former G_i à partir de G_{i-1} en éliminant i.

Ainsi, l'élimination d'une variable entraîne le moins de remplissage possible. L'avantage de cet algorithme est qu'il donne de bons résultats pour une large classe de problèmes. Il reste cependant, pour certains d'entre eux [3, 7, 8], sensible au choix d'un sommet quand plusieurs ont le même degré, et toute tentative de rendre le choix plus intelligent s'avère coûteuse.

1.1.2 Algorithme de dissection emboîtée

Diviser pour régner. Tel est le principe de cet algorithme. Nous partons du principe que la matrice A est *irréductible*, c'est-à-dire que le système linéaire ne peut se décomposer en deux sous-systèmes complètement indépendants. En termes de graphes, cela se traduit par le fait que G est *connexe*[1]. Le but de la dissection emboîtée est de faire interagir le moins possible des parties relativement indépendantes de la matrice. Pour cela, et partant du graphe initial G, il faut trouver un sous-ensemble X_1 de sommets tel que, si X_1 est supprimé de G, celui-ci est décomposé en au moins deux sous-graphes G_1^1 et G_2^1 disjoints. X_1 est appelé *séparateur*. En numérotant les sommets des deux sous-graphes avant ceux du séparateur, aucun des sous-graphes n'engendrera de nouvel arc, à la suite d'une élimination, dans l'autre. Cette procédure est appliquée récursivement à G_1^1 et G_2^1.

Tandis que cette méthode minimise le remplissage et exhibe du parallélisme (G_1^1 et G_2^1 peuvent être évalués en parallèle), l'algorithme du degré minimal nécessite un post-traitement pour accroître le parallélisme.

[1] Il existe un chemin reliant toute paire de sommets dans le graphe G.

Un problème non résolu actuellement est de trouver de bons séparateurs. En effet, le parallélisme mis en évidence est tributaire de la taille de ces séparateurs : plus ils sont petits, meilleur est le parallélisme [2].

1.2 Factorisation symbolique

Au cours de cette phase, la structure de la matrice L est construite. Cette structure qui servira à l'allocation de l'espace mémoire, accueillera aussi bien la matrice A que L. Ce traitement allège par conséquent la factorisation numérique.

1.2.1 Quelques définitions

Soit M une matrice creuse, symétrique, définie positive.

Définition 3 *La structure d'une ligne i de M est définie par l'ensemble des indices des éléments non nuls de M, se trouvant à la ligne i et dans le triangle inférieur :*

$$Struct(M_{i*}) = \{k < i / \, m_{ik} \neq 0\}.$$

□

Définition 4 *La structure d'une colonne j de M est définie par l'ensemble des indices des éléments non nuls de M, se trouvant à la colonne j et dans le triangle inférieur :*

$$Struct(M_{*j}) = \{k > j / \, m_{kj} \neq 0\}.$$

□

Définition 5 *On appelle premier d'une colonne j de structure non vide, l'indice du premier élément non nul dans la colonne j, au dessous de la diagonale.*

$$p(j) = \begin{cases} min\{i \in Struct(L_{*j})\} & si\ Struct(L_{*j}) \neq \emptyset, \\ j & sinon. \end{cases}$$

□

Définition 6 *L'arbre d'élimination T(A) de A est défini comme suit[12] :*

- *l'ensemble de ses nœuds est l'ensemble des indices des colonnes de L ;*
- *i est le père de j (avec $i > j$) $\Longleftrightarrow i = p(j)$.* □

Notons ici que, si la matrice A est irréductible, l'arbre d'élimination est effectivement un arbre dans le sens où il a une racine. Dans le cas contraire, il est représenté par un ensemble d'arbres disjoints. La figure 1 donne les grandes lignes d'un algorithme de construction de l'arbre d'élimination. Dans la pratique, une version optimisée sera utilisée.

```
For i := 1 to n
    parent[i] := 0
    For x_k ∈ Adj(x_i) & k < i do
        r := k ;
        While parent[r] ≠ 0 & parent[r] ≠ i do
                r := parent[r] ;
        Endwhile
        if parent[r] = 0 then parent[r] := i ;
    Endfor ;
Endfor.
```

Figure 1 : Algorithme de construction de l'arbre d'élimination.

1.2.2 Structure du facteur de Cholesky

Proposition 1 *La structure d'une colonne j du facteur de Cholesky peut être construite ainsi :*

$$Struct(L_{*j}) = Struct(A_{*j}) \cup \left(\bigcup_{\{i<j \,/\, p(i)=j\}} Struct(L_{*i}) \right) - \{j\}.$$

□

Cette proposition suggère directement un algorithme de calcul de la structure de L dont les grandes lignes sont exposées figure 2.

For $j := 1$ to n **do**
    $R_j := \emptyset$;
Endfor
For $j = 1$ **to** n **do**
    $S := Struct(A_{*j})$
    For $i \in R_j$ **do**
        $S := S \cup Struct(L_{*i}) - \{j\}$
    Endfor
    $Struct(L_{*j}) := S$
    if $Struct(L_{*j}) \neq \emptyset$ **then**
        $p(j) := \min\{i \in Struct(L*j)\}$
        $R_{p(j)} := R_{p(j)} \cup \{j\}$
    endif
Endfor

Figure 2 : Algorithme de factorisation symbolique pour matrices creuses.

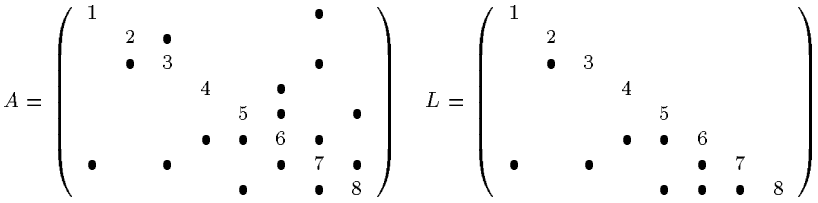


Figure 3 : Structure d'une matrice symétrique A et du facteur de Cholesky L correspondant.

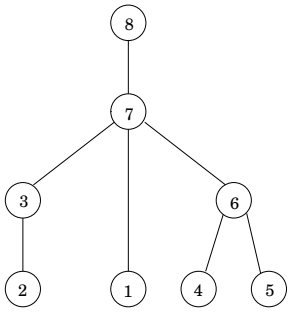


Figure 4 : Arbre d'élimination associé à L.

1.3 factorisation numérique

Pour chaque élément l_{ij} $(1 \leq j \leq i \leq n)$ du triangle inférieur de la matrice L, il faut faire les mises à jour suivantes :

$$\begin{aligned} l_{ij} &= l_{ij} - \sum_{k \in L_{j*}} l_{ik} * l_{jk} \\ l_{jj} &= \sqrt{l_{jj}} \\ et \;\; l_{ij} &= l_{ij}/l_{jj} \end{aligned}$$

L'espace des itérations est parcouru par trois boucles imbriquées sur i, j et k. Les six approches possibles pour parcourir cet espace, selon l'ordre d'imbrication des indices, sont illustrées figure 5. Ces algorithmes effectuent tous le même nombre de calculs mais diffèrent par l'ordre dans lequel ils les effectuent.

Lorsqu'une matrice creuse est rangée par colonnes, l'accès à ses éléments par ligne est couteux, et inversement. Par conséquent, les algorithmes 1, 4 et 5 de la figure 5 sont rarement utilisés.

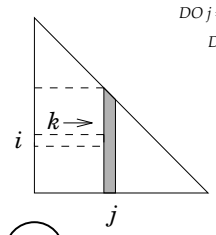
L'algorithme 3 résout, à chaque itération de la boucle externe sur i, un système triangulaire afin de calculer les éléments de la ligne i. Etant donné que ce type d'opérations exhibe un parallélisme de grain fin, sa mise en œuvre sur une machine à mémoire distribuée limite les performances [9]. Et comme nous visons une parallélisation sur une machine à mémoire distribuée, nous n'étudierons pas cet algorithme.

Nous nous intéressons donc plus particulièrement aux versions par colonnes (2 et 6) de la figure 5, et nous supposerons que les matrices creuses traitées sont rangées par colonnes. Nous appelerons ces deux versions *Fan-In* et *Fan-Out* respectivement. Ceci signifie que dans le premier cas, pour une itération de la boucle externe, la même colonne est modifiée. Dans le second cas, plusieurs colonnes sont modifiées.

Dans les deux cas, deux tâches de base sont définies :

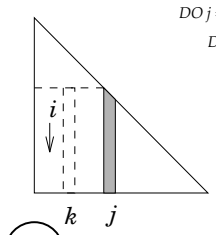
1. $cmod(j,k) \Longleftrightarrow col(j) = col(j) - l_{jk} * col(k)$.

   qui correspond à la partie du code :



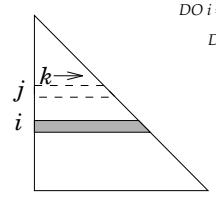
1

```
DO j = 1, n
    DO i = j, n
        DO k = 1, j-1
            A[i,j] = A[i,j] - A[i,k] * A[j,k]
        ENDDO
        If (i = j) THEN A[i,i] = SQR(A[i,i])
                   ELSE A[i,j] = A[i,j] / A[j,j]
        ENDIF
    ENDDO
ENDDO
```



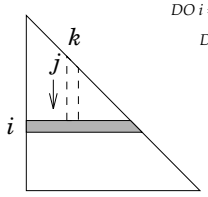
2

```
DO j = 1, n
    DO k = 1, j-1
        DO i = j, n
            A[i,j] = A[i,j] - A[i,k] * A[j,k]
        ENDDO
        A[j,j] = SQR(A[j,j])
        DO i = j + 1, n
            A[i,j] = A[i,j] / A[j,j]
        ENDDO
    ENDDO
ENDDO
```



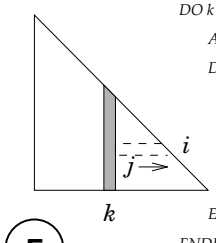
3

```
DO i = 1, n
    DO j = 1, i
        DO k = 1, j - 1
            A[i,j] = A[i,j] - A[i,k] * A[j,k]
        ENDDO
        If (i = j) THEN A[i,i] = SQR(A[i,i])
                   ELSE A[i,j] = A[i,j] / A[j,j]
        ENDIF
    ENDDO
ENDDO
```



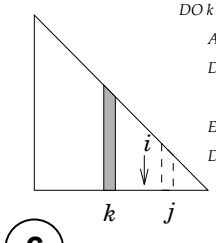
4

```
DO i = 1, n
    DO k = 1, i
        If (k = i) THEN A[i,i] = SQR(A[i,i])
                   ELSE A[i,k] = A[i,k] / A[k,k]
        ENDIF
        DO j = k + 1, i
            A[i,j] = A[i,j] - A[i,k] * A[j,k]
        ENDDO
    ENDDO
ENDDO
```



5

```
DO k = 1, n
    A[k,k] = SQR(A[k,k])
    DO i = k + 1, n
        A[i,k] = A[i,k] / A[k,k]
        DO j = k + 1, i
            A[i,j] = A[i,j] - A[i,k] * A[j,k]
        ENDDO
    ENDDO
ENDDO
```



6

```
DO k = 1, n
    A[k,k] = SQR(A[k,k])
    DO i = k + 1, n
        A[i,k] = A[i,k] / A[k,k]
    ENDDO
    DO j = k + 1, n
        DO i = j, n
            A[i,j] = A[i,j] - A[i,k] * A[j,k]
        ENDDO
    ENDDO
ENDDO
```

Figure 5 : Accès aux données.

```
DO i = j, n
        A[i,j] = A[i,j] - A[i,k] * A[j,k]
ENDDO
```

2. $cdiv(j) \Longleftrightarrow \left\{ \begin{array}{l} l_{jj} = \sqrt{l_{jj}} \\ col(j) = col(j)/l_{jj}. \end{array} \right.$

   qui correspond, dans la version Fan-In (2), à la partie du code:

```
A[j,j] = SQR(A[j,j])
DO i = j + 1, n
    A[i,j] = A[i,j] / A[j,j]
ENDDO
```

   et à la même partie du code dans la version Fan-Out en remplaçant j par k.

Cette nouvelle notation nous permet de réécrire les algorithmes Fan-In et Fan-Out comme décrits dans la figure 6.

For $j := 1$ **to** n **do**
  For $k := 1$ **to** $j-1$ **do**
  $cmod(j,k)$;
  Endfor
  $cdiv(j)$;
Endfor
	For $k := 1$ **to** n **do**
	  $cdiv(k)$;
	  For $j := k+1$ **to** n **do**
	  $cmod(j,k)$;
	  Endfor
	Endfor

Figure 6 : *Fan-In* et *Fan-Out* pour matrices denses.

Dans le cas de matrices creuses, il faut tenir compte du fait qu'il n'est pas utile de calculer $cmod(j,k)$ si l'élément l_{jk} est nul. En fait, les indices des colonnes dont l'élément l_{jk} n'est pas nul sont exactement ceux qui se trouvent dans l'ensemble $Struct(L_{j*})$. Les algorithmes de la figure 7 présentent les optimisations des algorithmes de la figure 6 qui découlent de cette constatation.

For $j := 1$ **to** n **do**
  For $k \in Struct(L_{j*})$ **do**
  $cmod(j,k)$;
  Endfor
  $cdiv(j)$;
Endfor
	For $k := 1$ **to** n **do**
	  $cdiv(k)$;
	  For $j \in Struct(L_{*k})$ **do**
	  $cmod(j,k)$;
	  Endfor
	Endfor

Figure 7 : *Fan-In* et *Fan-Out* pour matrices creuses.

L'ensemble $Struct(L_{*k})$ apparaissant dans l'algorithme Fan-Out est facilement obtenu puisque ce n'est autre que l'ensemble des indices de ligne des éléments non nuls de la colonne k (la matrice creuse est rangée par colonnes). Par contre, l'ensemble $Struct(Lj*)$ apparaissant dans l'algorithme Fan-In n'est pas obtenu aussi facilement. La façon dont il est calculé est décrite par l'algorithme de figure 8. $next(j,k)$, pour $k \leq j$, représente l'indice de la prochaine colonne après j, nécessitant une modification par la colonne k. C'est-à-dire que $next(i,k)$ est l'indice de ligne du prochain élément non nul dans la colonne k après la ligne j. Cet algorithme garantit qu'au traitement de la colonne j, l'ensemble S_j est égal à l'ensemble $Struct(L_{j*})$.

For $j := 1$ **to** n **do**
  For $k \in S_j$ **do**
  $cmod(j,k)$;
  $i := next(j,k)$;
  $S_i := S_i \cup k$
  Endfor
  $cdiv(j)$;
  $i := next(j,j)$;
  $S_i := S_i \cup j$
Endfor

Figure 8 : Détails du calcul de $Struct(L_{j*})$

Pour clore ce chapitre, rappelons que l'étape de factorisation numérique est de loin la plus importante en temps de calcul [7]. Elle sera donc la phase du processus de factorisation à paralléliser en priorité.

2 Parallélisation

Nous allons parcourir les différentes mises en œuvre parallèles des algorithmes Fan-In et Fan-Out développées jusqu'à présent. Pour les classes de machines parallèles prises en compte (machines MIMD), des implémentations existent pour deux grains de parallélisme :

- grain moyen : une tâche correspond à la réalisation d'une opération $Cmod(j,k)$ ou $Cdiv(j)$;
- grain large : une tâche correspond dans la version Fan-In à la réalisation de l'ensemble des $Cmod(j,k)$ pour $k \in Struct(L_{j*})$ et du $Cdiv(j)$ pour un j donné (Cette tâche sera notée T(j)). Dans la version Fan-Out, une tâche correspond à la réalisation du $Cdiv(k)$ et de l'ensemble des $Cmod(j,k)$, $j \in Struct(L_{*k})$ pour un k donné. Cette tâche sera notée J(k).

Deux modèles de programmation seront considérés : la distribution des données et la distribution du contrôle. La distribution des données consiste à répartir les données sur les différents processeurs puis à affecter le contrôle du programme selon cette répartition. Dans le cas particulier du modèle SPMD (Sin gle Program Multiple Data), ceci est réalisé par la règle *Owner compute* : grossièrement, cela signifie qu'un processeur n'exécute une instruction d'affectation que s'il est propriétaire de la donnée écrite par cette instruction. Quant au modèle de programmation par distribution du contrôle, seul le flux de contrôle est parallélisé (par exemple : distribution des boucles), chaque processeur étant supposé pouvoir accéder à toute donnée dont il a besoin.

2.1 Dépendances

En termes d'opérations $Cmod(j,k)$ et $Cdiv(j)$, les dépendances de données peuvent s'exprimer ainsi (figure 9) :

- *Cmod(j,k)* doit être effectué après *Cdiv(k)* ;
- *Cdiv(j)* doit être effectué après $\{Cmod(j,k) \ / \ k \in Struct(L_{j*})\}$.

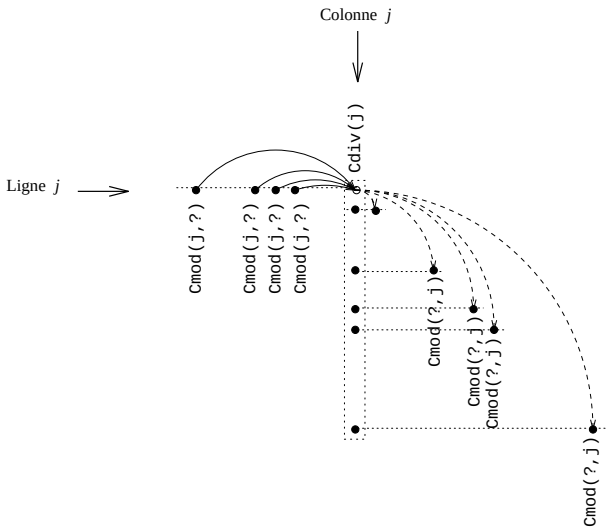


Figure 9 : Graphe des dépendances relatives à une colonne

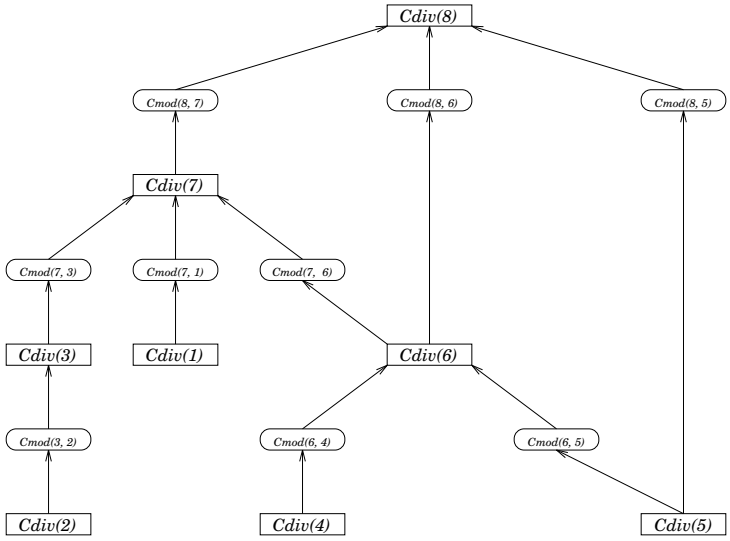


Figure 10 : Graphe des dépendances - Exemple

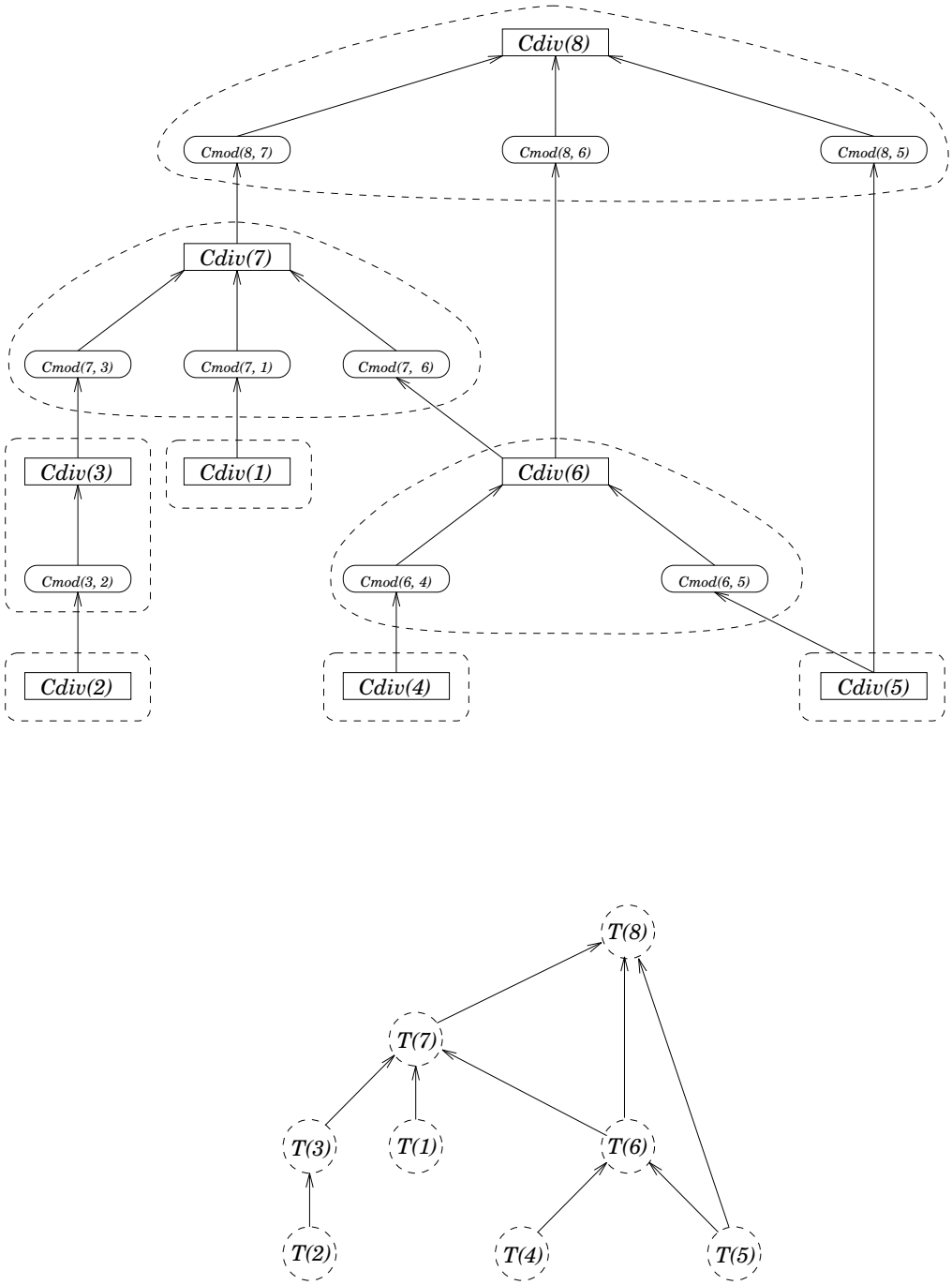


Figure 11 : Graphe des dépendances - Fan-In à grain large

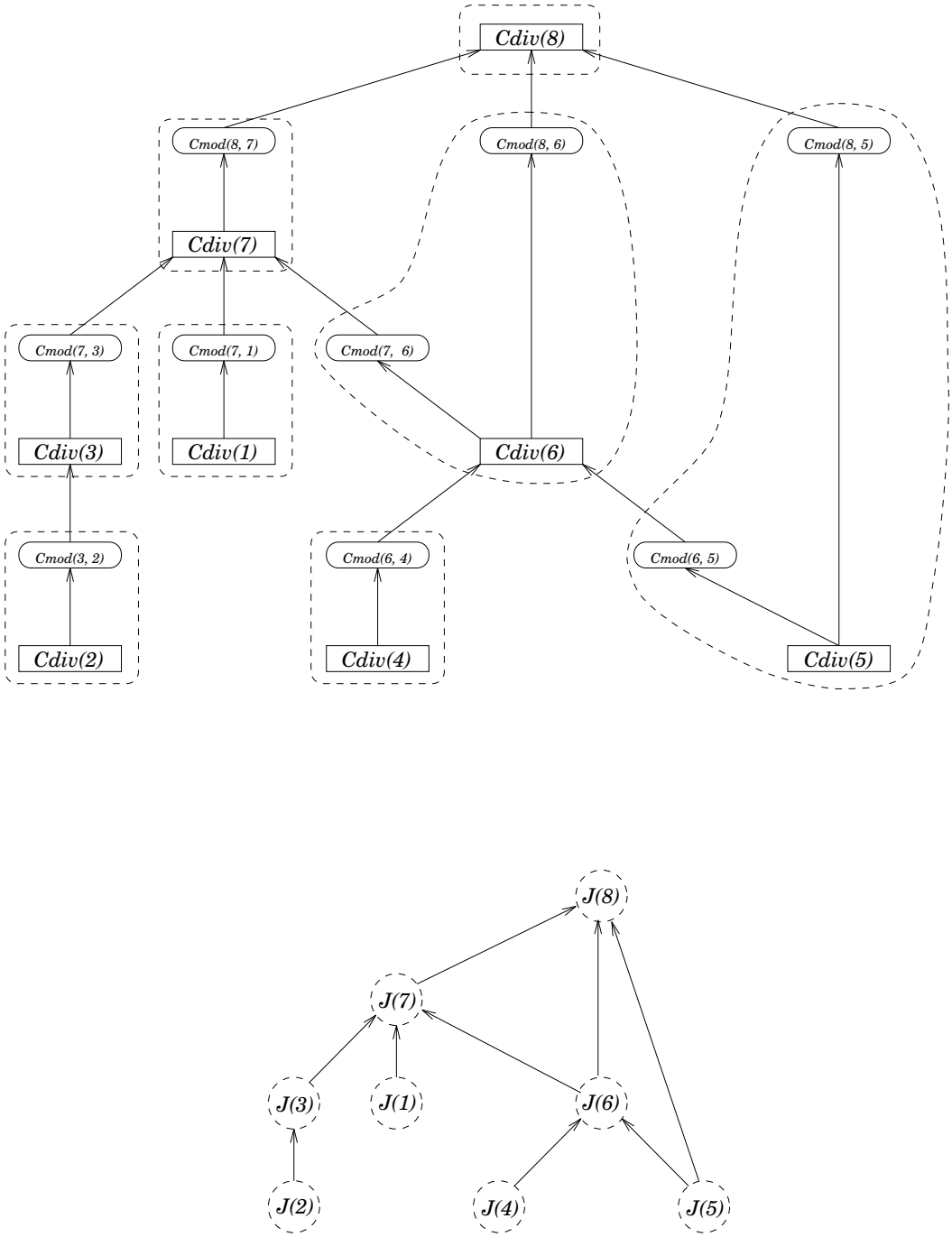


Figure 12 : Graphe des dépendances - Fan-Out à grain large

La figure 10 montre sur un exemple concret (celui de la matrice figure 3) les dépendances entre les différentes tâches de base $Cdiv$ et $Cmod$. Lorsqu'on passe à un grain plus large, la tâche de base devient T(j) dans le cas du Fan-In et J(k) dans le cas du Fan-Out. Dans ces termes, les dépendances sont illustrées par les figures 11 et 12. Une réduction transitive de ces graphes de dépendances permet de retrouver l'arbre d'élimination de la matrice.

Il a été démontré dans [12] qu'un nœud de l'arbre ne peut dépendre que de ses descendants. Nous pouvons en déduire que des nœuds (colonnes de L) n'ayant pas de relation ascendant-descendant sont indépendants l'un de l'autre, et peuvent être évalués en parallèle.

2.2 Algorithme Fan-Out

2.2.1 Distribution des données

L'approche par distribution des données consiste à répartir les données sur les processeurs selon une stratégie à définir, puis à affecter les calculs aux processeurs de manière à minimiser les transferts de données.

L'algorithme Fan-Out fut parmi les premiers algorithmes parallèles de factorisation de Cholesky pour matrices creuses introduits sur une machine à mémoire distribuée. Il exploite un parallélisme de grain moyen, en tirant profit de l'indépendance des $Cmod(j,k)$ pour un j donné. L'algorithme, introduit par George et al. [4], est repris dans la figure 13.

Les matrices utilisées pour les tests de performances, provenant de problèmes à éléments finis, sont renumérotées par l'algorithme de dissection emboitée afin de garantir un remplissage minimal du facteur de Cholesky et un bon équilibrage de l'arbre d'élimination. Les nœuds de l'arbre d'élimination sont marqués dans un ordre topologique (un nœud n'est marqué que si tous ses fils le sont déjà). Ceci revient à parcourrir l'arbre par niveaux et à affecter aux nœuds d'un niveau des numéros supérieurs à ceux du niveau précédent. Cette numérotation servira au placement des nœuds (un nœud correspond à une colonne) sur les processeurs par entrelacement simple. La figure 14 montre sur un petit exemple la numérotation d'un arbre d'élimination par entrelacement simple.

Les contraintes de précédence de tâches réalisées par cet algorithme sont illustrées par la figure 15. Ce graphe est une extension du graphe de dépendances, où certaines relations de précédence sont dues aux choix faits dans

```
For j ∈ mycols(p) do
    if j is a leaf node in T(A) do
        cdiv(j)
        send column j to processors in Struct(L_{*j})
        mycols(p) := mycols(p) - {j}
    endif
enfor

While mycols(p) ≠ ∅ do
    receive any column of L, say k
    For j ∈ Struct(L_{*k}) ∩ mycols(p) do
        cmod(j, k)
        if column j requires no more cmod's do
            cdiv(j)
            send column j to processors in Struct(L*j)
            mycols(p) := mycols(p) - {j}
        endif
    endfor
endwhile
```

Figure 13 : Algorithme *Fan-Out* pour machine à mémoire distribuée.

l'algorithme et au placement statique des données. Pour notre exemple, les seules contraintes supplémentaires introduites sont dues à la phase d'initialisation des calculs, où l'on effectue les *Cdiv(k)* pour toutes les colonnes ne nécessitant aucune modification par d'autres colonnes. La minimisation des attentes éventuelles induites par ces contraintes est un problème complexe, tributaire du placement des colonnes sur les processeurs.

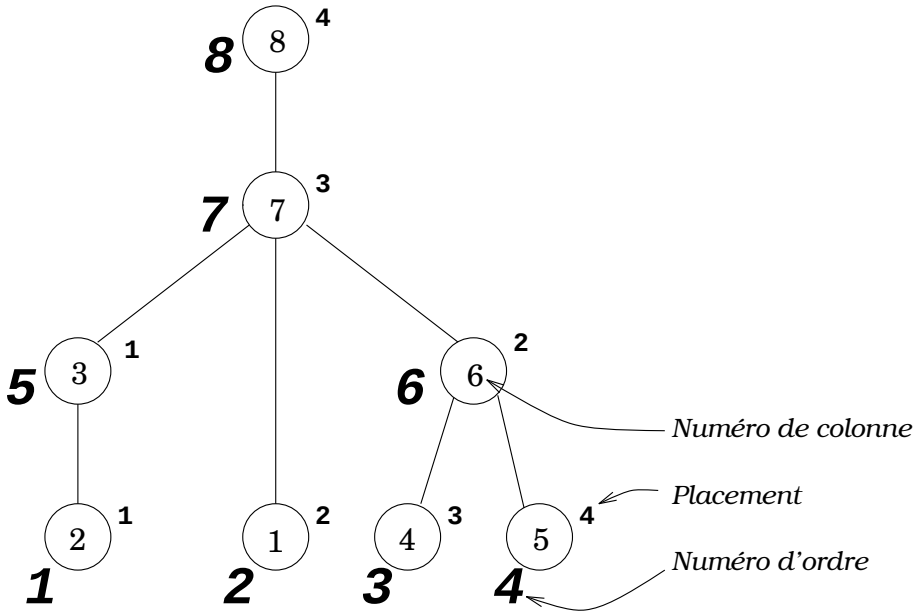


Figure 14 : Ordre topologique et entrelacement

2.2.2 Distribution du contrôle

Rotheberg et Gupta [15] ont dérivé de l'algorithme Fan-Out une version pour machines à mémoire partagée. La principale innovation est l'exploitation de supernœuds (colonnes consécutives ayant un bloc triangulaire diagonal plein et une même structure en dessous de ce bloc). Les supernœuds diminuent considérablement les accès indirects à la mémoire et augmentent le grain de parallélisme. Il y a par conséquent moins de synchronisation entre les processeurs, mais le problème d'équilibrage de la charge devient crucial. L'algorithme est décrit en détail dans le paragraphe 3.2.1.

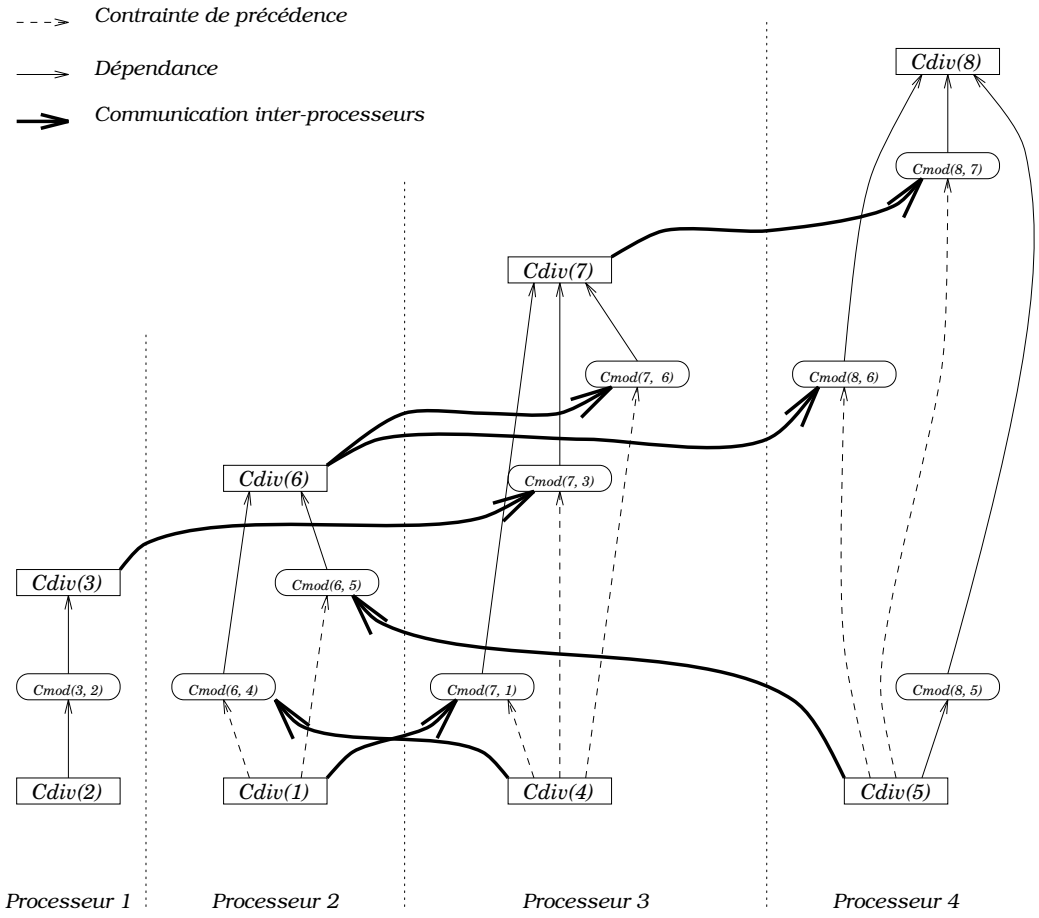


Figure 15 : Contraintes de précédence

Les résultats, en termes de performances, obtenus par Rotheberg et Gupta sont intéressants. Cependant, une étude en simulation qu'ils ont poursuivie dans [14] démontre que les performances du Fan-Out supernodal en présence d'une mémoire partagée s'estompent avec un nombre de processeurs supérieur à 8. Pour 32 processeurs, il est des cas où l'utilisation des supernœuds est un inconvénient plutôt qu'un avantage. Ceci est étroitement lié à la taille et au nombre de supernœuds. Néanmoins, cet algorithme est celui qui se comporte le mieux sur une machine à mémoire partagée et qui donne en général les meilleures performances.

2.3 Algorithme Fan-In

2.3.1 Distribution des données

La version du Fan-In pour machines à mémoire distribuée a été introduite par Ashcraft et al. [1]. C'est un des algorithmes les plus efficaces pour factoriser des matrices creuses, sur une machine à mémoire distribuée. De même que le Fan-Out, il exploite un parallélisme de grain moyen. Il est difficile de concevoir un parallélisme de grain plus large sur une machine à mémoire distribuée, étant donné qu'une tâche T(j) ou J(k) peut nécéssiter l'accès à une colonne que le processeur en charge de la tâche ne possède pas. L'algorithme est décrit dans la figure 16.

Le point critique dans cet algorithme est le choix de la répartition des colonnes sur les processeurs. Un consensus est vite survenu autour de la technique de placement du "sous-arbre à sous-cube" (*subtree to subcube mapping*) qui est illustrée par la figure 17 pour deux processeurs. En effet, cet algorithme a la faculté de réduire considérablement les communications requises avec cette distribution particulière des données. Par exemple, supposons que la colonne 8 se trouve chez le processeur 1. L'algorithme Fan-Out aurait nécessité l'envoi de la colonne 5 du processeur 2 vers le processeur 1, puis de la colonne 6. Dans le même cas de figure, l'algorithme Fan-In n'aurait engendré qu'un envoi de message accumulant la contribution des colonnes 5 et 6 à la colonne 8. Ceci explique intuitivement la supériorité des performances du Fan-In distribué sur le Fan-Out distribué.

Toutefois, cet algorithme souffre du fait que chaque processeur est sensé connaître les ensembles Struct(L_{j*}) pour toutes les colonnes de la matrice.

Cette information étant stable dans le temps, sa duplication peut être envisagée, moyennant un espace mémoire en $O(nz(L))$.

```
for j = 1 to n do
    if j ∈ mycols(p) or StructL_{j*} ∩ mycols(p) ≠ ∅ do
      u := 0
      for k ∈ Struct(L_{j*}) ∩ mycols(p) do
          u := u + l_{jk} * col(k)
      Endfor
      if map[j] ≠ p then
          send aggregate update column u to proc map[j]
      else
          col(j) := col(j) − u
          while not all contributions have been received do
                receive any aggregate update column u for column j
                from another processor and subtract u from col(j)
          Endwhile
          cdiv(j)
      Endif
    Endif
Endfor
```

Figure 16 : Algorithme *Fan-In* pour machine à mémoire distribuée.

2.3.2 Distribution du contrôle

George et al. ont développé en 1986 un algorithme pour machines parallèles à mémoire partagée[5]. Cet algorithme est une adaptation quasi-directe de l'algorithme séquentiel Fan-In pour matrices creuses (figure 8). Il est repris dans la figure 18.

Les tests de performances effectués avec cet algorithme sur une machine *Sequent* sont fort prometteurs. Toutefois, un faible nombre de processeurs (7 processeurs) a été utilisé.

Cet algorithme exploite une technique d'équilibrage dynamique de la charge de calcul. Il utilise pour cela une file d'attente de tâches où les processeurs inoccupés viennent récupérer du travail. Cette file d'attente est initialisée de manière à minimaliser les attentes et éviter les interblocages. Une numérotation selon un ordre topologique de l'arbre d'élimination permet de

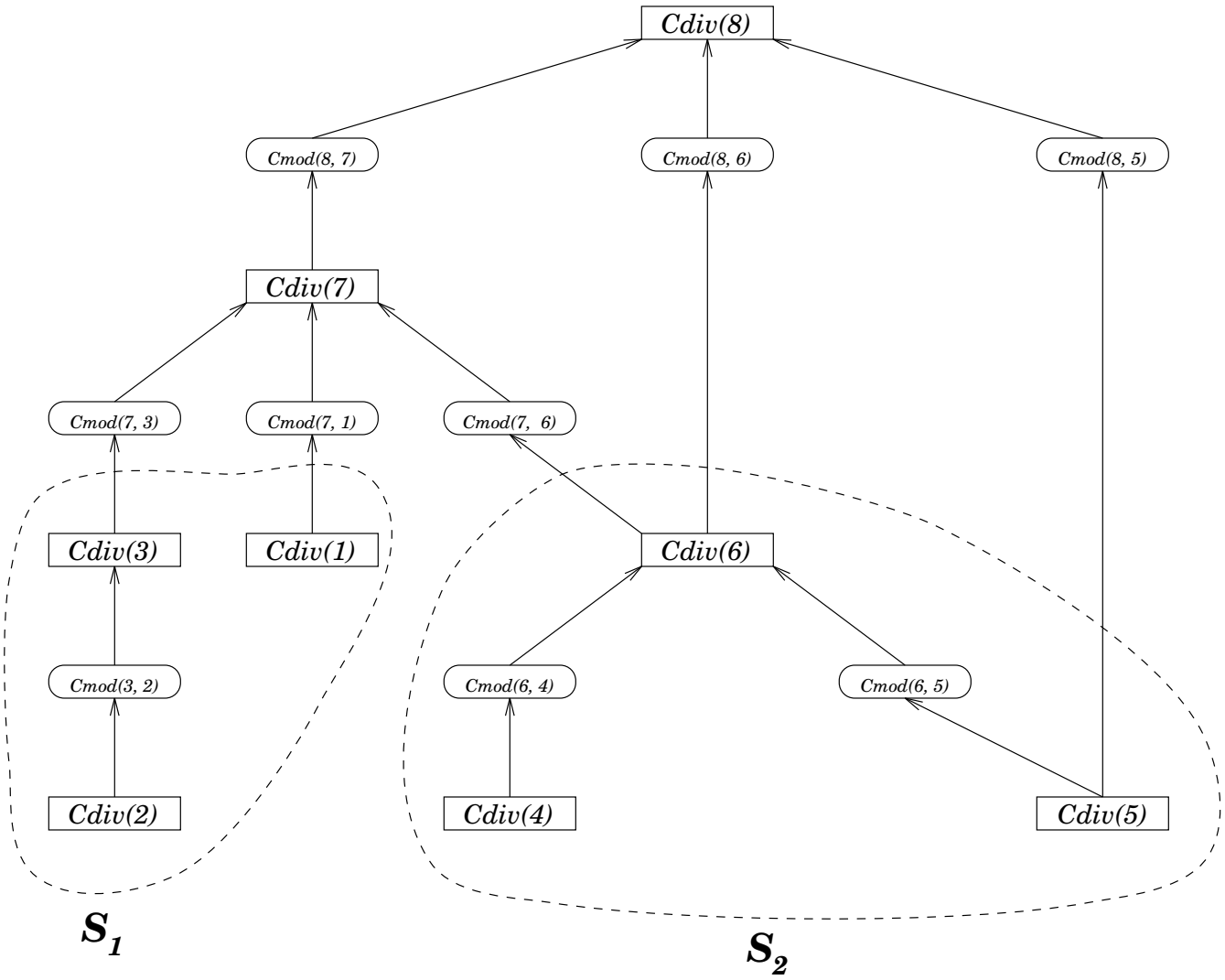


Figure 17 : Répartition par “sous-arbre à sous-cube”

réaliser ces objectifs. En l'occurence, un ordonnancement selon un parcours par niveaux de l'arbre d'élimination a été utilisé. Il est important de signaler que le parallélisme au sein de cet algorithme est à grain large, ce qui diminue les synchronisations entre processeurs.

```
Q := {T_1, T_2, ..., T_n}
While Q ≠ ∅
        pop T(j) from Q
        While column j requires further cmod's do
                if S_j = ∅
                        wait until S_j ≠ ∅
                obtain k from S_j
                cmod(j, k)
                i = next(j, k)
                lock
                S_i := S_i ∪ {k}
                unlock
        EndWhile
        cdiv(j)
        i :=next(j, j)
        lock
        S_i := S_i ∪ {j}
        unlock
EndWhile
```

Figure 18 : Algorithme Fan-In pour machines à mémoire partagée.

Etant donné que les tâches T(j) peuvent nécessiter des temps de calculs très différents, il est difficile d'étudier le comportement de l'algorithme dans le cas général. Néanmoins, nous illustrons l'extension du graphe de dépendance réalisée par cet algorithme sur la figure 19. Nous constatons que la méthode utilisée par l'algorithme pour construire les ensemble S_i introduit des contraintes de précédence supplémentaires. Ainsi, le processeur en charge de la tâche T(8) ne peut effectuer Cmod$(8,5)$ avant que le processeur en charge de la tâche T(5) n'ait effectué Cmod$(6,5)$. Il ne peut non plus effectuer Cmod$(8,6)$ avant que Cmod$(7,6)$ ne soit effectué. Le graphe initial des dépendances (figure 10) montre que ces opérations sont entièrement in-

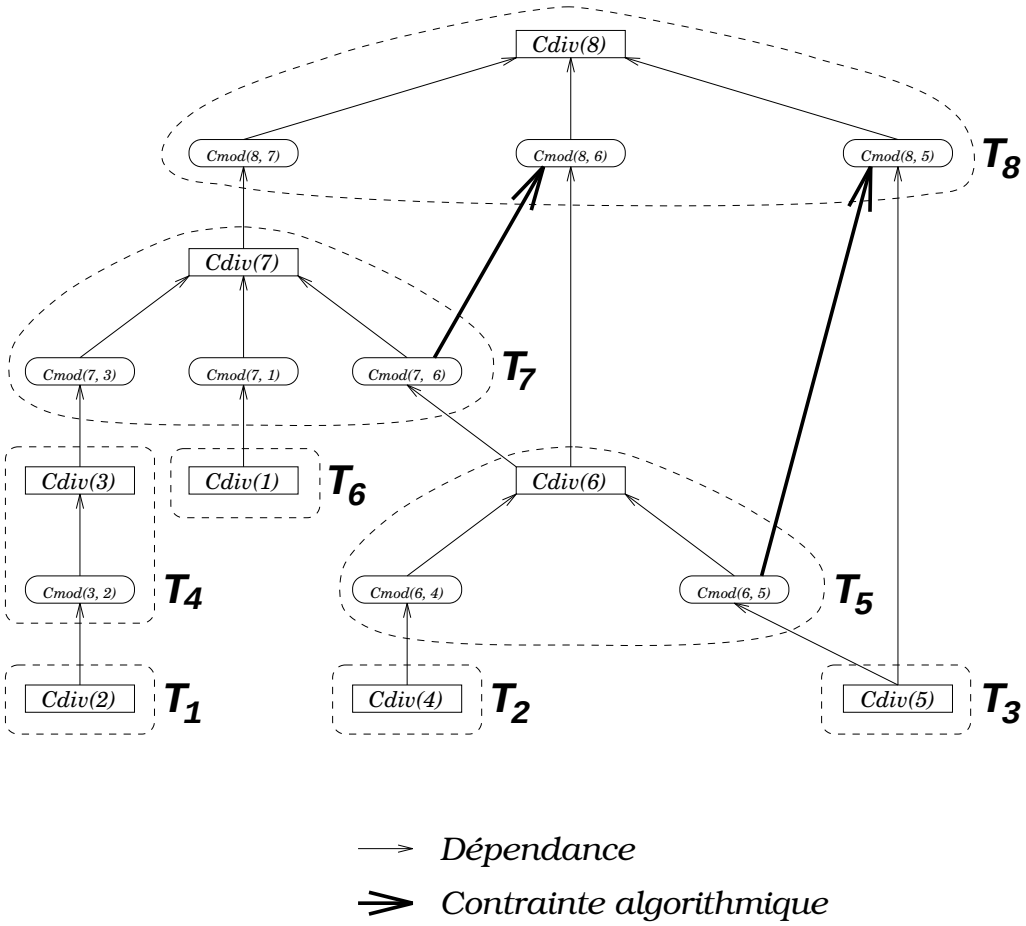


Figure 19 : Contraintes de précédence

dépendantes. Toutefois, ces contraintes peuvent être supprimées au prix d'un accroissement substantiel de la taille mémoire requise.

Le détail marquant de cet algorithme reste l'utilisation fréquente de l'exclusion mutuelle. En effet, l'accès à la file d'attente des tâches et aux ensembles S_i doit se faire en section critique pour éviter des incohérences. Le nombre d'accès en section critique est de l'ordre de $n + nz(L)$. L'algorithme est donc surtout adapté aux machines où ces mécanismes sont implémentés de manière efficace, et aux problèmes où les tâches sont de taille importante.

3 Approches sur une mémoire virtuelle partagée

Nous nous plaçons maintenant dans le contexte d'une machine à mémoire distribuée dotée d'un mécanisme de mémoire virtuelle partagée. En l'occurence, la machine est un *iPSC/2* d'*Intel* avec 32 processeurs ayant chacun 4 Mo de mémoire. *KOAN* est la mémoire virtuelle partagée pour cette machine. Il offre à l'utilisateur la vision d'un espace d'adressage unique global à partir des différentes mémoires locales [10]. *KOAN* se charge également de maintenir une cohérence forte sur les données partagées : une opération de lecture à une adresse retourne la dernière valeur écrite à cette adresse. Des développements récents permettront sans doute d'exploiter une cohérence faible : une opération de lecture à une adresse retourne la dernière valeur écrite au point de synchronisation précédent. En d'autre termes, une opération d'écriture n'est pas instantanément visible à tous les processeurs, ce qui permet plus de concurrence dans les opérations de lecture/écriture.

3.1 Approche par échange de messages

A titre de référence, nous avons mis en œuvre l'algorithme du Fan-In distribué (décrit précédemment dans la figure 16) sur l'*iPSC/2*. Les colonnes ont été distribuées sur les processeurs par un entrelacement simple. Ceci a permis de mettre en évidence certain problèmes liés à l'implémentation de cet algorithme et qui influent sur ses performances. Entre autres, deux remarques importantes sont à signaler :

1. A chaque itération j de l'algorithme, il est nécessaire de connaître l'ensemble *Struct*(L_{j*}). En effet, seuls les processeurs ayant au moins une colonne dont l'indice est dans cet ensemble ou bien possédant la colonne j exécuteront cette itération. Cet ensemble est construit à la volée, au début de l'itération. Sa construction pose un problème parce que la matrice est rangée par colonnes et que l'on a besoin d 'y accéder par lignes. Ce calcul est rendu possible grâce à une structure de données distribuée, constituée de deux vecteurs de longueur n. Le nombre d'accès à cette structure est $O(n^2)$. Si l'information *Struct*(L_{j*}) est construite durant la factorisation symbolique, elle nécessitera un espace mémoire de $O(nz(L))$ et un nombre d'accès également en $O(nz(L))$.

2. Lorsqu'un processeur entame le calcul d'un *Cmod(j,k)*, il ne connait pas nécessairement la structure de la colonne j. Dans des mises en œuvre plus classiques, il faut transmettre à chaque fois la structure de la contribution calculée sur chaque processeur, pour une colonne j donnée. Ceci augmente sensiblement le volume des communications et du temps est perdu à construire le message à transmettre. La solution que nous adoptons est la duplication de cette information stable dans le temps. Ainsi, chaque processeur connait la structure de toutes les colonnes de la matrice, et peut donc formatter les résultats partiels en conséquence.

Certaines matrices, telles que la Lshp3466, ont peu d'éléments par colonne[2]. Ceci induit un rapport communication/calculs très élevé et comme nous pouvons le constater sur la figure 20, le passage à une version parallèle sur deux processeurs introduit un surcoût important dû aux communications.

# Proc	séq	2	4	8	16	32	messages
Bcsstk14	88,16	69,52	37,57	21,63	13,62	9,45	34261
Lshp3466	37,14	92,55	49,48	27,77	16,37	10,37	35659
GH1-01	281,85	147,34	76,09	40,57	22,72	13,88	14343

Tableau 1 : Temps(s) de calcul et nombre de messages du Fan-In distribué

[2]Voir les caractéristiques de la matrice dans l'annexe A.

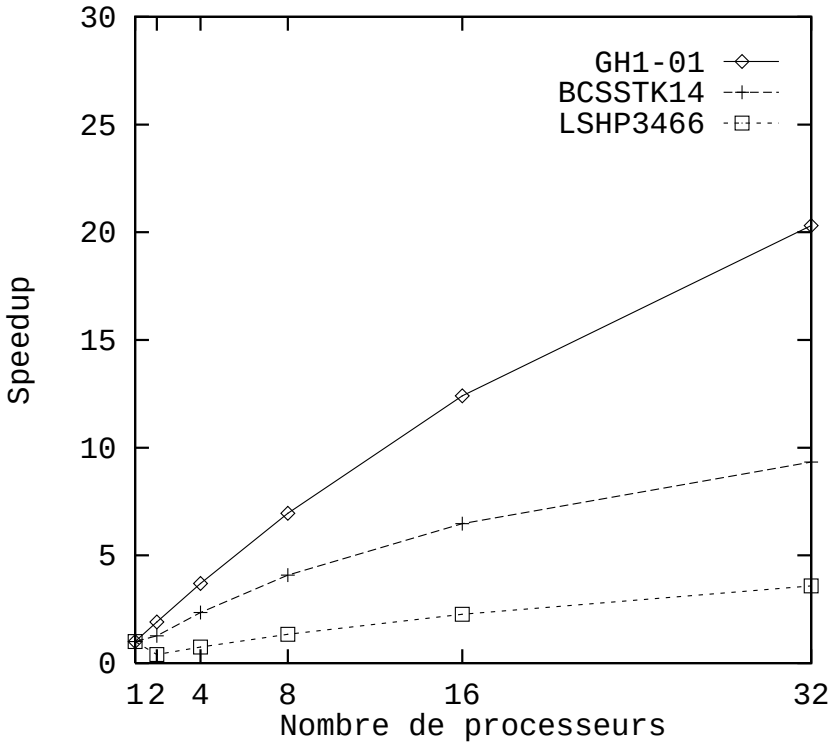


Figure 20 : Accélérations du Fan-In

Les résultats obtenus et illustrés par le tableau 1 font référence à un temps séquentiel. Ce temps est mesuré sur un algorithme Fan-Out séquentiel que nous avons développé et que nous avons exécuté sur un processeur de l'*iPSC/2*. Seule la factorisation numérique est prise en considération dans les mesures. La dernière colonne indique le nombre de messages émis par l'ensemble des processeur lors d'une exécution sur 32 processeurs. La figure 20 montre les accélérations atteintes par rapport à un code séquentiel.

3.2 Approche par variables partagées

Dans cette approche, l'utilisateur ne s'occupe plus de la répartition des données. En effet, la mémoire virtuelle partagée *KOAN* lui offre la vision d'un espace d'adressage unique global. Il suffit pour cela de déclarer une région partagée, et de l'initialiser avec les valeurs de la matrice A. Ceci se fait comme suit :

- sur la machine hôte :

  `init_region(adr, size, MAPPING)`

  Cette instruction transmet aux nœuds un vecteur de taille `size` dont les valeurs sont stockées à l'adresse `adr`. Le paramètre `MAPPING` spécifie la manière dont les pages sont au départ réparties sur les différentes mémoires locales des processeurs (`BLOCK, MODULO`).

- sur chaque nœud de la machine :

  ```
  id = create_region(size)
  adr = map_region(id, MAPPING, COHERENCY)
  load_region(id)
  ```

  ce qui a pour effet de créer une région de taille `size` et de la répartir sur les différent processeurs selon une fonction de distribution (`MAPPING`). Le paramètre `COHERENCY` indique quel type de cohérence associer à cette zone partagée. `load_region` lance l'initialisation de la région partagée à partir du frontal de la machine.

Pour plus de détails, le lecteur est invité à consulter [13]. Dans les mises en œuvre qui nous intéressent, seule une cohérence forte a été exploitée. Nous

verrons par la suite qu'un protocole de cohérence faible est plus adapté aux codes creux.

Dans ce qui suit, nous ne distribuons que le contrôle, sans nous soucier de la localisation des données qu'il fait intervenir. C'est là l'un des apports majeurs de la mémoire virtuelle partagée.

3.2.1 Fan-Out supernodal

Etant donné que nous disposons maintenant d'un modèle de programmation par variables partagées, une première expérience a consisté à mettre en œuvre un algorithme pour machines à mémoire partagée. Notre choix s'est porté sur le Fan-Out partagé, dont le code est proposé dans la série des benchmarks *SPLASH* [16], et que nous avons introduit dans le paragraphe 2.2.2. D'une part, c'est un algorithme performant et d'autre part, il est plus facile d'adapter un code existant. L'algorithme est illustré par la figure 21.

Rappelons en quelques mots son comportement : une file d'attente est initialisée par l'ensemble des colonnes ne nécessitant aucune modification par d'autres colonnes. Chaque processeur vient prendre une tâche dans cette file (un tâche correspond à un supernœud). Il accomplit les modifications nécessaires à l'intérieur même de ce supernœud (ensemble de cmod() et de cdiv()), puis il calcule sa contribution aux autres colonnes de la matrice. Si ce supernœud se trouve être le dernier à affecter une colonne ou un supernœud, celui-ci est mis dans la file. Donc, une colonne est accédée en écriture par plusieurs processeurs avant d'être mise dans la file, ensuite, elle est accédée en lecture par un seul processeur et est utilisée pour les modifications qu'elle engendre.

Les points de synchronisation dans cet algorithme se trouvent à deux niveaux :

1. l'accès à la file d'attente, que ce soit pour en retirer ou y déposer une tâche, doit être effectué en exclusion mutuelle ;

2. l'accès à une colonne pour écriture doit se faire en exclusion mutuelle également. Il ne faut pas que deux processeurs modifient la même colonne au même moment.

Par conséquent, un sémaphore d'exclusion mutuelle est utilisé pour synchroniser les accès à la file d'attente, et n sémaphores sont utilisés pour

```
Q := J(1), J(2), ..., J(n)
For j = 1 to n
    pop J(j) from Q
    let X(j) be the supernode containing column j
    While column j requires further cmod's do
        if S_j ≠ ∅
            wait until S_j ≠ ∅
        obtain K from S_j
        i = next(j, K)
        lock
        S_i := S_i ∪ K
        unlock
        cmod(j, K)
    Endwhile
    cmod(j,X(j))
    cdiv(j)
    if j is the last column of supernode X(j)
        i :=next(j, X(j))
        lock
        S_i := S_i ∪ j
        unlock
    Endif
Endfor
```

Figure 21 : Fan-Out supernodal pour machines à mémoire partagée

l'accès en écriture aux colonnes (n étant le nombre de colonnes de la matrice en entrée).

Les résultats que nous avons obtenus, sur un jeu de matrices de test, tendent à prouver que cette approche ne convient pas à l'architecture visée. Ces algorithmes ne peuvent pas être performants si les outils de synchronisation (utilisés intensivement) ne sont pas efficacement mis en œuvre. C'est notamment le cas sur des machines à mémoire distribuée où la synchronisation se fait par envoi de messages. Davantage de détails sur la mise en œuvre de la synchronisation et de l'exclusion mutuelle dans *KOAN* sont décrits dans [11].

En résumé, les problèmes que nous avons rencontrés dans cette expérience sont dus à :

- la présence d'une file d'attente gérée en mémoire virtuelle partagée ;
- le nombre élevé de sections critiques ;
- le taux élevé de faux partage de données : le grain de cohérence dans *KOAN* est la page de mémoire. Lorsque deux processeurs accèdent en écriture à deux colonnes différentes mais qui se trouvent rangées dans la même page de mémoire, le système considère que c'est la même entité qui est accédée. Par conséquent, et en vue de maintenir une cohérence forte sur les données, chaque accès en écriture sur une colonne chez un processeur invalide la copie de la page en question se trouvant chez l'autre. Ainsi, la page va et vient d'un processeur à l'autre (effet *ping-pong*) engendrant une perte de temps. Dans l'approche Fan-Out, ce phénomène est important vu l'absence totale de localité spatiale et temporelle dans les écritures (un processeur lit une colonne k puis écrit dans toutes les colonnes de L_{*k}).

3.2.2 Fan-In à grain moyen

Vu les résultats de l'expérience précédente, en particulier la prédominance de la localité spatiale ou à défaut temporelle dans les écritures, il est clair que les approches Fan-In sont plus adéquates. Nous avons donc développé puis mis en œuvre un certain nombre de variantes du Fan-In à grain moyen. Ce choix vise dans un premier temps les objectifs suivants :

- supprimer la file d'attente qui provoque un goulôt d'étranglement. La solution adoptée est la synchronisation globale des processeurs au calcul de chaque colonne. Ainsi tous les processeurs contribuent au calcul d'une même colonne. Ce parallélisme exploite l'indépendance des différents $Cmod(j,*)$ entre eux.

- exploiter l'architecture de la machine : les différents $Cmod(j,*)$ au niveau d'un processeur effectuent des écritures locales dans un vecteur temporaire. Puis ces vecteurs sont accumulés par le biais d'une fonction globale (offerte par le système *NX/2*) qui synchronise les processeurs. Celle-ci effectue les sommes selon un arbre binaire de recouvrement. Un seul processeur est chargé, à l'issue de cette opération, de rendre effective la mise à jour sur la colonne j en recopiant le résultat en mémoire partagée.

Le point de départ est donc l'algorithme 2 de la figure 5. La différence majeure entre les différentes variantes de la parallélisation de cet algorithme réside dans la manière de distribuer la boucle interne sur k. Nous adoptons la notation suivante :

DOPAR signifie que la boucle est parallèle ;

DISTRIBUTED signifie que le domaine d'itération est distribué cycliquement sur les processeurs ;

Local signifie que les *Cmod( )* effectuent toutes les écritures dans un vecteur temporaire local ;

GdSum est une opération de sommation (réduction) à partir de vecteurs locaux, offerte par le système *NX/2* ;

Gsync est une barrière de synchronisation.

A partir de là, la figure 22 montre quatre variantes du même algorithme :

1. les $Cmod(j,*)$ sont distribués de manière à répartir la charge de calcul sur les processeurs. Ceci se fait en évaluant à l'exécution le nombre d'opérations nécessaires pour la colonne j, ce qui coûte un temps non négligeable ;

2. dans le même esprit que la version précédente avec une distribution plus fine. Un processeur peut n'avoir à calculer qu'une partie d'un $Cmod(j,k)$;

3. les $Cmod(j,*)$ sont distribués de manière statique à la compilation. Si une opération $Cmod(j,k)$ n'a pas d'effet sur la colonne j ($L_{jk} = 0$), elle n'est pas effectuée. Mais cela ne rentre pas en ligne de compte pour l'équilibrage de la charge de travail. En fait, une répartition assez quelconque des éléments non nuls de la matrice garantit un minimum d'équilibrage de la charge. Cette façon de faire nous permet d'économiser le temps nécessaire à l'évaluation du nombre de calculs pour chaque colonne ;

4. la version précédente est reprise en supprimant la barrière de synchronisation en fin de boucle externe. Celle-ci qui était obligatoire pour gérer la cohérence des données au niveau algorithmique n'est plus indispensable puisque chaque colonne n'est utilisée que par un seul et même processeur durant tout le calcul.

Les résultats obtenus sur la MVP *KOAN* sont reportés sur la figure 23. Sur cette figure, le temps donné pour un processeur est le temps du code parallèle exécuté sur un nœud de la machine. Globalement, les performances sont comparables à la version par messages (pour la version 4) jusqu'à 4 processeurs. Au-delà, elles sont moins satisfaisantes. Ces résultats montrent que :

- le coût de la gestion d'un grain fin n'est pas compensé par le gain apporté par un meilleur équilibrage de la charge de travail (la version 2 est moins performante que la version 1) ;

- la localité des données est prépondérente par rapport à l'équilibrage de la charge (les versions 3 et 4 donnent de meilleures performances que les versions 1 et 2) ;

- l'implémentation logicielle (par envoi de messages) de la barrière de synchronisation est très coûteuse. En outre, elle ne permet pas qu'une itération compense le déséquilibrage de la charge survenu dans les itérations précédentes (La version 3 est moins performante que la version 4).

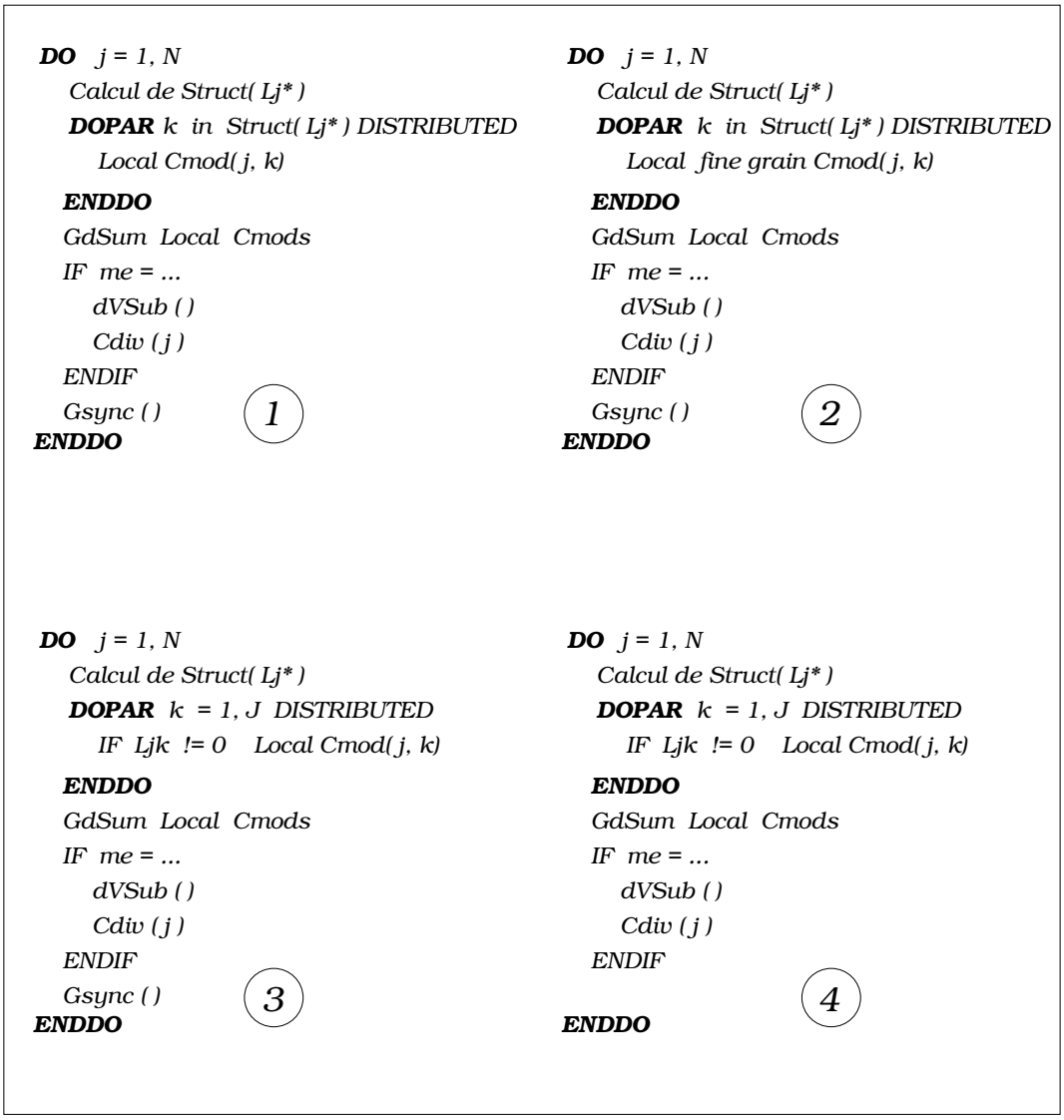


Figure 22 : Fan-In à grain moyen.

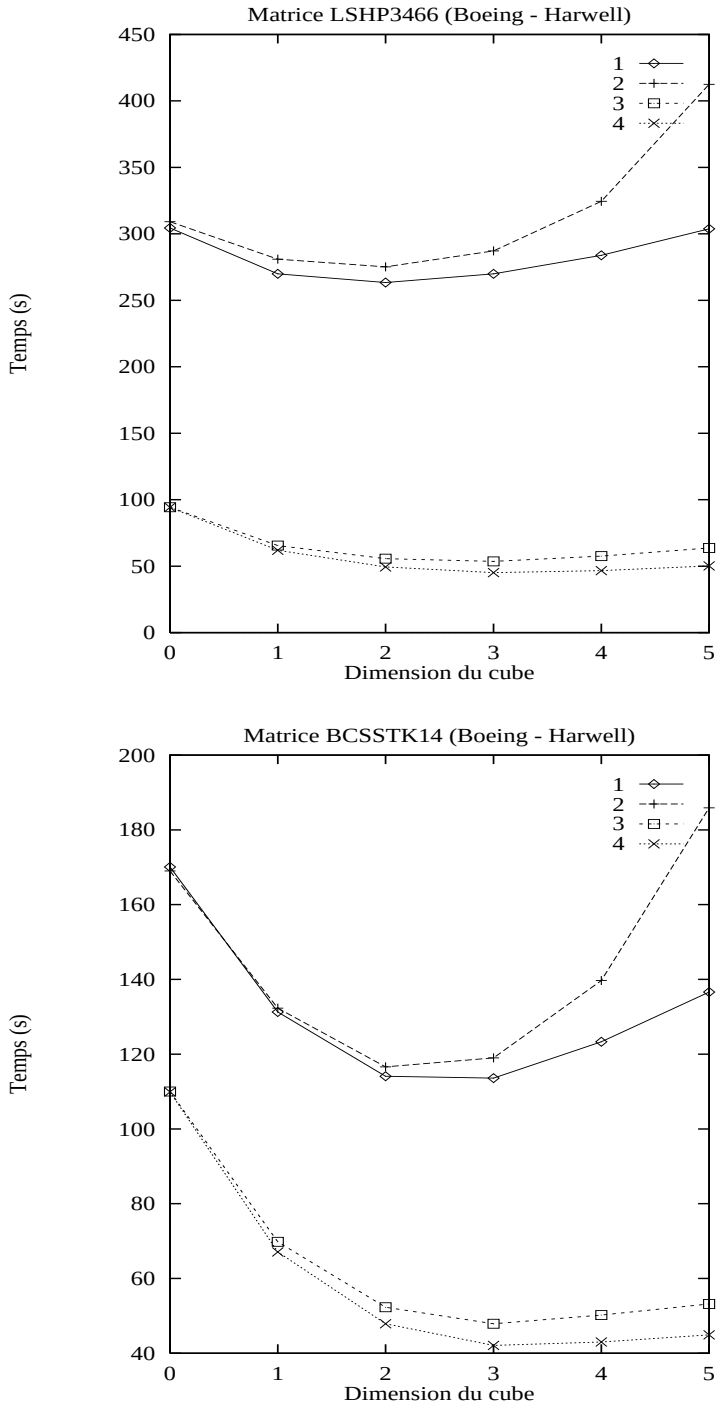


Figure 23 : Temps du Fan-In à grain moyen.

3.2.3 Fan-In à grain large

Afin d'améliorer les résultats que nous avons obtenus jusqu'à présent, nous visons les objectifs suivants :

- un grain de calcul plus large ; ceci est devenu possible grâce à la mémoire virtuelle partagée. Il est difficile, voire impossible, d'envisager cette approche sur une machine à mémoire distribuée sans mécanisme de MVP. En effet, les tâches de grain large font intervenir des données dont le processeur ne dispose peut-être pas.
- une utilisation limitée des barrières de synchronisation.

Etant donné que la MVP met à la disposition d'un processeur toute donnée dont il a besoin (quelle que soit sa localisation), l'utilisateur peut se consacrer à la distribution du contrôle uniquement. Toutefois, l'utilisateur averti doit toujours favoriser la réutilisation des données (localité temporelle) afin de minimiser les transferts au niveau de la MVP. La recherche d'algorithmes favorisant un bon équilibrage de la charge et exhibant une bonne localité reste également à la charge de l'utilisateur.

Grossissement du grain. Afin d'exploiter un grain plus large, la boucle externe de l'algorithme séquentiel est distribuée sur les processeurs, ce qui correspond à la distribution des tâches $T(j)$.

Respect des dépendances. Un processeur doit vérifier avant d'effectuer un $Cmod(j,k)$ que la colonne k a été produite ($Cdiv(k)$ effectué). Pour cela, il doit vérifier qu'un signal lui est parvenu. En contrepartie, tout processeur, après avoir produit une colonne j, doit le signaler aux autres processeurs. Nous avons mis en œuvre un système de gestion d'événnements asynchrones sur l'hypercube offrant les fonctionnalités suivantes :

- signaler qu'un événement particulier s'est produit ;
- vérifier si un événement particulier s'est produit, et le cas échéant, se bloquer en attendant qu'il se produise.

Ceci est réalisé grâce aux deux primitives `Post-event(e)` et `Wait-event(e)`, `e` étant le type d'événement considéré.

Les événements auxquels nous nous intéressons, à savoir la réalisation d'un *Cdiv(j)*, sont des évènemets stables dans le temps (l'occurence d'un

événement n'est jamais remise en cause). Aussi, l'opération `Wait-event(e)` est consultative et non consommatrice.

Dans une première mise en œuvre des synchronisations par événements, chaque nœud dispose dans sa mémoire locale d'une vision globale sur l'ensemble des événements qui se sont produits. L'occurence d'un événement se traduit par une diffusion de l'information à l'ensemble des processeurs. Ultérieurement, la mise en place d'un gestionnaire unique distibué d'événements pourra être envisagée.

Dans ce nouveau contexte, l'algorithme de factorisation numérique est décrit par la figure 24. Afin de minimiser l'attente des processeurs sur un `wait-event(e)`, les colonnes de la matrice ne sont pas traitées dans l'ordre de 1 à n, mais selon une numérotation topologique de l'arbre d'élimination. Concrètement, supposons que le vecteur `TOPO[`t`]` indique la colonne dont le numéro d'ordre topologique est t. La boucle externe de l'algorithme s'écrira :

```
DOPAR t = 1 to n DISTRIBUTE [CYCLIC]
   j = TOPO[t]
   ⋮
ENDDO
```

Nous avons testé deux heuristiques de numérotation topologique de l'arbre d'élimination. Elles correspondent à un marquage au plus tôt et au plus tard des colonnes suivant leur appartenance à un niveau de l'arbre d'élimination [6]. En effet, la seule contrainte d'apartenance à un niveau pour un nœud (une colonne) est que son père soit dans un niveau supérieur. Sur les tests que nous avons effectués, il n'est apparu d'avantage pour aucune des deux méthodes. Nous pensons qu'un compromis entre les deux marquages mènera vers un meilleur équilibrage de la charge et un minimum d'attente dans le cas général, en tenant compte du :

- nombre total de processeurs ;
- nombre de tâches (colonnes) disponibles à chaque niveau de l'arbre d'élimination ;
- nombre d'opérations arithmétiques nécessaires pour chaque tâche.

```
DOPAR j = 1 to n DISTRIBUTE [MODULO]

   DO k ∈ Struct(L_j*)
      Wait-event(k)
      Cmod(j, k)
   ENDDO

   Cdiv(j)
   Post-event(j)
ENDDO
```

Figure 24 : Algorithme synchronisé par événements

Les mesures que nous avons effectuées sur les temps d'exécution sont reportées dans la figure 25. Il en ressort que l'algorithme se comporte mieux en termes de performances avec la matrice GH1-01 qu'avec la matrice Bcsstk14, et mieux encore qu'avec la matrice Lshp3466. En effet, plus la matrice est pleine moins il y a de colonnes dans une même page de mémoire, et donc moins le taux de faux partage est élevé. Sur l'*iPSC/2*, une page de mémoire contient en moyenne des éléments de cinq colonnes différentes de la matrice GH1-01. Dans le cas des matrices de la collection Boeing-Harwell, en l'occurence la Bcsstk14 et la Lshp3466, le nombre moyen d'éléments par colonne est de 61 et 23 respectivement. Donc, une page de mémoire contient 9 colonnes de la matrices Bcsstk14 et 23 colonnes de la matrice Lshp3466. Ceci induit un faux partage considérable, occasionnant des invalidations de pages. La figure 26 donne un aperçu sur le temps d'attente maximal sur des défauts de page, ainsi que le nombre total de défauts de page ayant survenu durant l'exécution. Ces chiffres sont à l'origine des explications que nous avançons. La figure 27 donne les taux de défauts de page en lecture et en écriture. Ils représentent une vision temporelle du taux de partage des pages de mémoire. Nous constatons que le taux de partage en écriture tend à se stabiliser tandis que le taux de partage en lecture ne cesse de croître. Cette information, conjuguée au fait que le rapport écriture/lectures est de 2.27% pour la matrice BCSSTK14 et de 4.31% pour la matrice LSHP3466, nous montre que

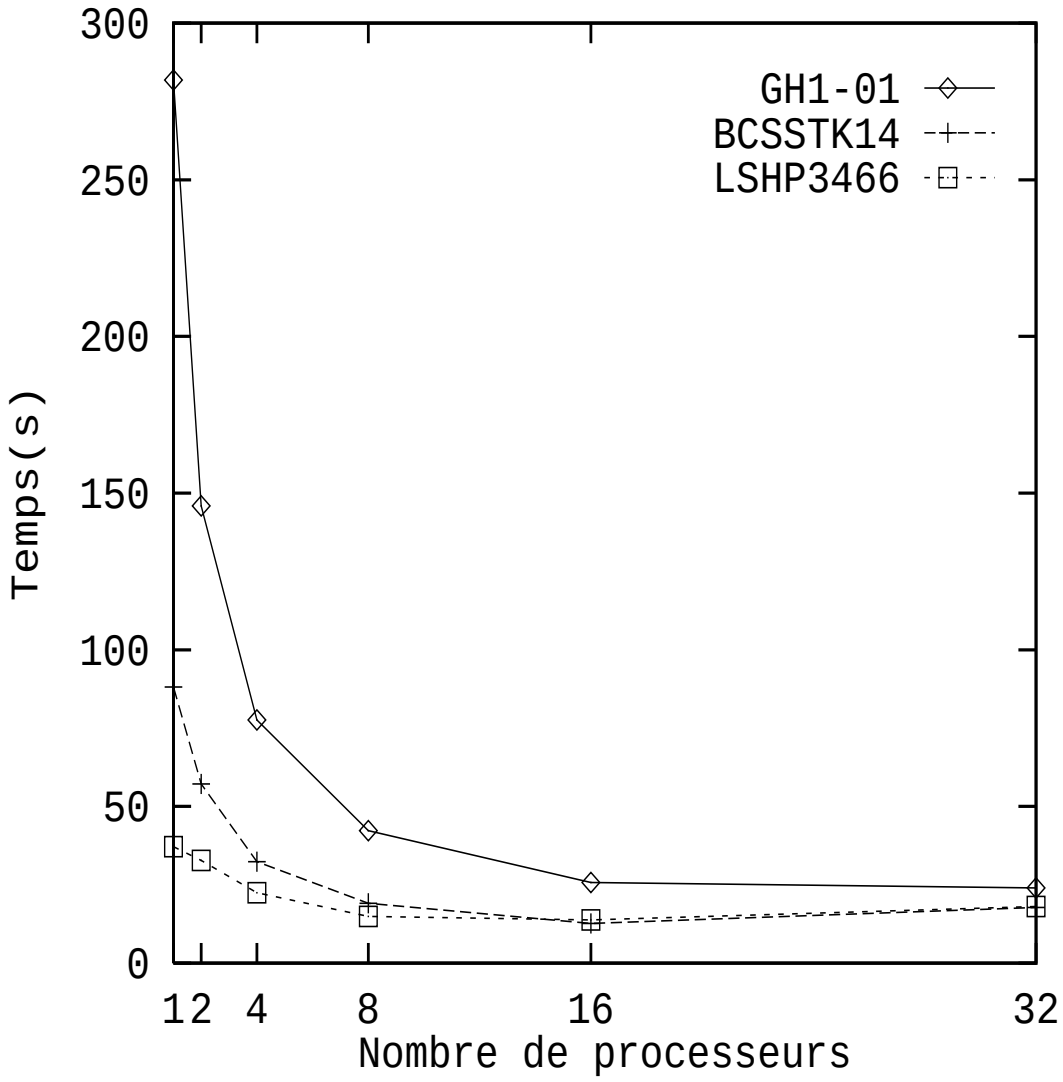


Figure 25 : Temps du Fan-In à grain large.

les conflits de pages sont essentiellement de type 1 rédacteur / n lecteurs. Et non seulement leur nombre augmente rapidement, mais le temps de service d'un défaut de page s'en trouve lui-même augmenté (comme nous pouvons le constater sur la figure 28), détériorant les performances du système.

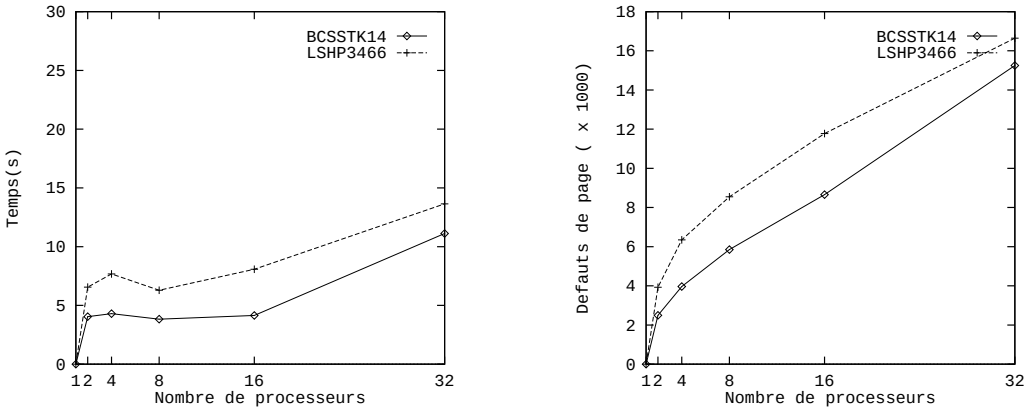


Figure 26 : Défauts de page du Fan-In à grain large.

Pour une taille de page mémoire fixée à 4096 octets, un nombre minimal de 512 éléments par colonne, stockés sur 64 bits, est nécessaire pour garantir qu'une page ne contienne pas plus de deux colonnes distinctes (au maximum une frontière puisque les colonnes ne sont pas alignées sur un début de page). Or, avec 512 éléments par colonnes et pour rester dans le cas creux, il faut traiter des matrices de l'ordre de 50000 colonnes. Des matrices de cet ordre occuperaient alors environ 300 Mo de mémoire ! Nous ne pouvons pas traiter des problèmes de cette taille bien qu'ils soient représentatifs des problèmes réels. Toutefois, ces tailles de problèmes permettraient à l'utilisateur de se consacrer totalement sur l'équilibrage de la charge, ultime obstacle pour l'obtention des performances maximales.

Nous comparons également les techniques d'ordonnancement choisies, à savoir :

1. une distribution cyclique des itérations de la boucle de 1 à n ;
2. une distribution par entrelacement simple après renumérotation des itérations sur la base d'un parcours par niveaux de l'arbre d'élimination.

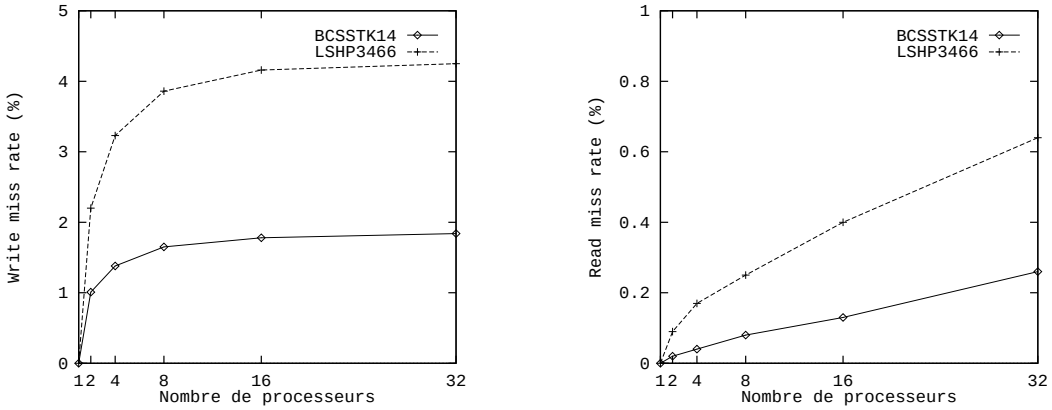


Figure 27 : Taux de défauts de page.

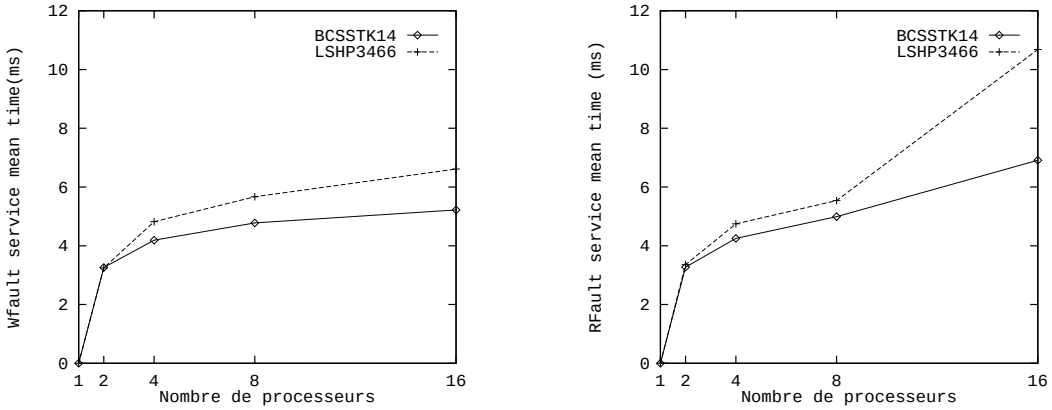


Figure 28 : Temps moyen de service d'un défaut de page.

la figure 29 donne les résultats de cette comparaison. Elle montre un avantage net au profit de l'exploitation de l'arbre d'élimination pour l'ordonnancement. En ce qui concerne les matrice Boeing-Harwell utilisées, le parallélisme potentiel est limité avec une distribution cyclique de la boucle externe de l'algorithme. Nous constatons également que l'exploitation de l'arbre d'élimination pour l'ordonnancement a un effet de bord appréciable : il y a moins de conflits d'accès aux données. Ce qui est traduit par une diminution notable du temps passé dans *KOAN*. Ceci s'explique par le fait que l'ordonnancement choisi fait travailler les processeurs sur des colonnes non voisines, et qui ont donc moins de chance de se trouver stockées dans la même page de mémoire. Par conséquent, le problème du faux partage de données s'est trouvé exacerbé avec la première technique d'ordonnancement.

Plusieurs optimisations restent possibles, notamment l'utilisation d'un protocole de cohérence faible. Ainsi, le problème du faux partage de données n'engendrera plus d'effet "ping-pong". Sa mise en œuvre pourra être basée sur l'outil de synchronisation proposé. Un `Post_event(j)` mettra dans un état cohérent tous les morceaux de page contenant une copie de la colonne j, et ce sur tous les processeurs qui en disposent. Reste à faire la part de l'utilisateur, du compilateur et du système pour exploiter correctement et efficacement ce protocole de cohérence.

Une autre optimisation, relative cette fois au stockage en mémoire de la matrice à traiter (voir paragraphe A pour la représentation des matrices creuses), a été envisagée. Elle consiste à regrouper les colonnes correspondant à des taches attribuées à un même processeur. Ceci est possible puisque l'attribution des tâches est statique. Cette technique permet de réduire au minimum le nombre de frontières entre zones de données accédées en écriture par des processeurs différents. Dans notre cas, nous n'avons pas constaté un gain substantiel de performance. Dans le cas de la matrice BCSSTK14, le nombre de défauts de page en écriture a dimnué de 13% pour 32 processeurs, jusqu'à 63% pour 2 processeurs. Mais pour 32 processeurs, le nombre d'invalidations de page a augmenté de près de 188%, occasionnant plus de défauts de page en lecture. Le même phénomène a été observé pour la matrice LSHP : diminution des défauts de page en écriture variant de 35% à 82% et augmentation des invalidations jusqu'à 138%. Une explication possible est qu'un processeur passant plus de temps sur une même page, puisqu'il doit en traiter tous les éléments, engendre plus d'invalidations sur les processeurs ayant besoin de cette page en lecture.

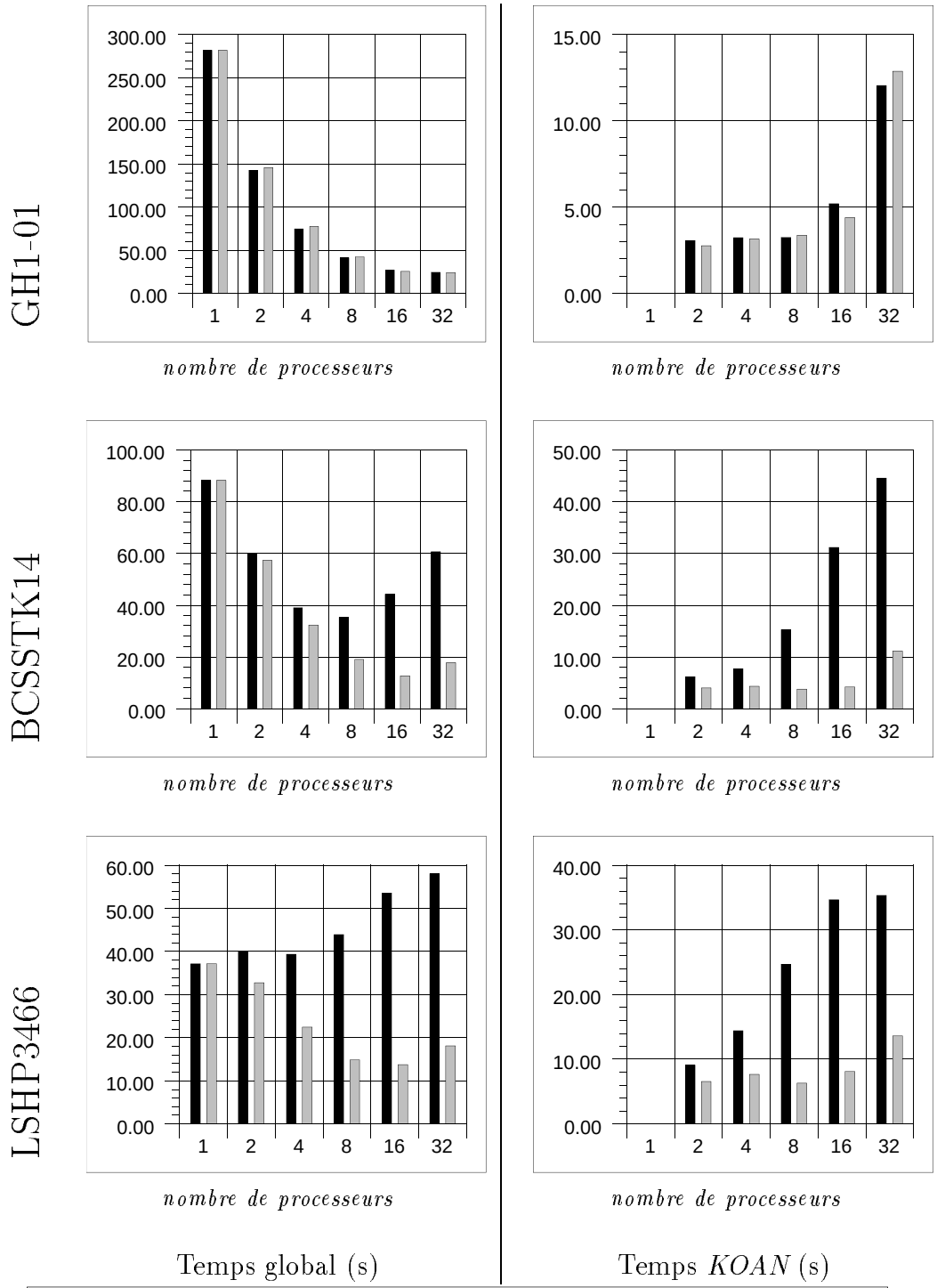


Figure 29 : Stratégies d'ordonnancement - Résultats

Conclusion

Nous avons étudié deux approches distinctes pour la mise en œuvre d'un algorithme de factorisation de matrices creuses, sur une machine à mémoire distribuée. D'une part, l'approche par distribution des données, et communication explicite par messages, et d'autre part, en exploitant une mémoire virtuelle partagée et en distribuant le contrôle.

Chaque approche a ses avantages et ses limites. La programmation par distribution des données permet au code généré d'être efficace. Ceci s'explique par le fait que l'utilisateur, *expert*, spécifiant la distribution des données sur les différentes mémoires locales, peut optimiser les accès en conséquence, et minimiser les communications au strict nécessaire. Ce travail peut se révéler coûteux en temps par manque d'outils adéquats.

Par contre, une MVP opère une répartition initiale "aveugle" de l'ensemble des données, qui n'est pas forcément adaptée à la distribution du contrôle qui sera spécifiée. Mais, vu que les données migrent à la demande des processeurs, cette répartition n'engendre qu'un coût relativement faible au lancement de l'application.

La subdivision de l'espace d'adressage en pages de taille fixe fait apparaître le problème du faux partage de données. Par ailleurs, cette même subdivision permet une vectorisation implicite des accès aux données puisque chaque transfert de page véhicule plus d'information que ce qui est nécessaire à l'exécution d'une instruction. En d'autres termes, la latence due à l'initialisation d'un transfert est rentabilisée puisqu'elle n'est payée qu'une seule fois au lieu de l'être pour chaque donnée à transmettre. En outre, cela permet une exploitation de la localité spatiale et/ou temporelle, même si l'utilisateur n'a pas fait d'efforts dans ce sens. Dans le cas de la factorisation de Cholesky, cette localité est faible vu que le calcul sur un processeur ne fait pas systématiquement intervenir des colonnes successives.

A Matrices de test

Les performances des codes dits "creux" (opérant sur des matrices creuses) sont étroitement liées à la nature des données à traiter. Aussi, nous avons évalué les performances des algorithmes développés ici sur plusieurs matrices. Certaines d'entre elles proviennent de la collection Boeing Harwell, telles que les BCSSTK14 (toîture de l'*Omni Coliseum, Atlanta*) et LSHP3466 (discrétisation d'une région en L). Les principales caractéristiques de ces matrices sont résumées dans le tableau 2. Pour des contraintes de configuration matérielle de notre machine parallèle, il n'a pas été possible de traiter des matrices de plus grande taille.

La matrice GH1-01 a été obtenue à partir d'un générateur de matrices creuses aléatoires. Des informations complémentaires sur cette matrice sont illustrés par la figure 30.

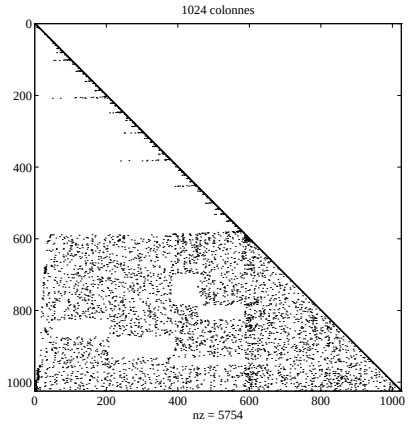


Figure 30 : Triangle inférieur de la matrice GH1-01

Toutes ces matrices ont été renumérotées par l'algorithme du degré minimal, qui assure un faible remplissage du facteur de Cholesky au cours du processus de factorisation. Ces matrices sont rangées en mémoire dans un format compacté faisant intervenir les trois vecteurs suivants :

`Nz:` valeurs des éléments non nuls stockés par colonnes ;

`Row:` indices de ligne des valeurs dans `Nz` ;

matrice	*ordre*	*nz(A)*	*nz(L)*	*flops*
Lshp3466	3 466	23 896	86 582	7 969 619
Bcsstk14	1 806	32 630	110 461	19 476 401
GH1-01	1 024	5 754	109 701	29 889 747

Tableau 2 : Matrices de test

`FirstNz:` début de chaque colonne dans `Row`.

Le nombre d'opérations flottantes indiqué dans le tableau 2 compte indifféremment les additions, multiplications, divisions et racines carrées. Sur l'*iPSC/2*, le temps d'exécution de ces opérations sont équivalents.

References

[1] Cleve ASHCRAFT, Stanley EISENSTAT, and Joseph LIU. A fan-in algorithm for distributed sparse numerical factorisation. *Siam journal of scientific and statistical computations*, 1990.

[2] Alain GEORGE and Joseph LIU. An automatic nested dissection algorithm for irregular finite element problems. *Siam journal of numerical analysis*, 1978.

[3] Alain GEORGE and Joseph LIU. The evolution of the minimum degree ordering algorithm. *Siam review*, 1989.

[4] Alan GEORGE, Michael HEATH, Joseph LIU, and Esmond NG. Sparse cholesky factorisation on a local-memory multiprocesor. *Siam journal of scientific and statistical computations*, 1988.

[5] Alan GEORGE, Michael HEATH, Joseph LIU, and Esmond NG. Solution of sparse positive definite systems on a shared memory multiprocesor. *International journal of parallel programming*, 1986.

[6] Hjalmtyr HAFSTEINSSON. *Parallel sparse Cholesky factorization.* PhD thesis, Cornell university, 1988.

[7] Michael HEATH, Esmond NG, and Barry PEYTON. *Parallel algorithms for matrix computations*, chapter Parallel algorithms for sparse linear systems. Siam, 1990.

[8] I.DUFF, A.ERISMAN, and J.REID. On george's nested dissection method. *Siam journal on numerical analysis*, 1976.

[9] P.S. KUMAR, M.K. KUMAR, and A. BASU. Parallel algorithms for sparse triangular system solution. *Parallel computing*, 1993.

[10] Zakaria LAHJOMRI and Thierry PRIOL. Koan : a shared virtual memory for an ipsc/2 hypercube. CONPAR/VAPP, september 1992.

[11] Zakaria LAHJOMRI and Thierry PRIOL. *Koan : A shared virtual memory for an iPSC/2 hypercube.* Technical Report, Irisa/Inria, 1991.

[12] Joseph LIU. A compact row storage scheme for cholesky factors using elimination trees. *ACM transactions on mathematical software*, 1986.

[13] Thierry PRIOL and Zakaria LAHJOMRI. Experiments with shared virtual memory and message-passing on an ipsc/2 hypercube. International Conference on Parallel Processing, August 1992.

[14] Edward ROTHBERG and Anoop GUPTA. *A comparative evaluation of nodal and supernodal parallel sparse matrix factorisation: detailed simulation results.* Technical Report, Department of computer science-Stanford university, 1990.

[15] Edward ROTHBERG and Anoop GUPTA. Techniques for improving the performance of sparse matrix factorisation on multiprocessor workstations. Supercomputing 90, November 1990.

[16] Jaswinder SINGH, Wolf-Dietrich WEBER, and Anoop GUPTA. Splash: stanford parallel applications for shared-memory. Computer systems laboratory, Stanford. 1991.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399