# Programmation distribuée avec partage de tableaux : la bibliothèque CIDRE

F. André et Y. Mahéo

IRISA, Campus de Beaulieu, 35042 Rennes cedex
Email : (fandre | maheo)@irisa.fr

## 1 Introduction

La modularité et l'extensibilité constituent deux points forts incontestables des architectures à mémoire distribuée.

Dans ces architectures, bâties à partir de nœuds processeur–mémoire interconnectés, on peut facilement faire varier le nombre de nœuds pour adapter la puissance de la machine à la taille du problème à traiter et aux performances souhaitées.

Parmi ces architectures, on trouve d'une part des calculateurs spécialement conçus avec cette orientation comme l'Intel Paragon ou l'IBM SP2, d'autre part des réseaux de stations de travail haute performance, par exemple un réseau ATM reliant des PC. Ceci met ces architectures à la portée d'un grand nombre d'usagers.

En dépit d'importants efforts de recherche et de progrès remarquables depuis quelques années, leur programmation demeure encore assez complexe en raison de leurs mémoires distribuées qui contraignent à utiliser des opérations de communication.

Le modèle de programmation qui à ce jour paraît encore plus attrayant dans le cadre d'applications importantes est celui de la programmation parallèle par variables partagées. Ce modèle permet de mettre en évidence le parallélisme qui sera source de performances tout en gardant une vue globale des structures de données manipulées.

L'amélioration de l'utilisation des architectures à mémoire distribuée passe donc par la mise en œuvre du modèle de processus communiquant par variables partagées sur architecture à mémoire distribuée. Les systèmes de mémoire virtuelle partagée constituent une solution à ce problème [9, 8, 6]. Les variables partagées sont placées dans la mémoire virtuelle et sont donc adressées de manière uniforme. Cependant plusieurs inconvénients viennent tempérer cette apparente facilité d'emploi. Le plus important est lié au coût des communications des pages virtuelles. La taille des pages est indépendante de celle des variables accédées : il s'ensuit un volume de communication plus important que celui nécessairement requis. Mais surtout, lorsqu'une page contient plusieurs variables accédées en parallèle par différents processeurs, des scénarii de communications «ping-pong» se produisent.

Notre objectif est de proposer une alternative à ce système de mémoire virtuelle partagée. Nous avons conçu une mise en œuvre des variables partagées originale et efficace sur les deux points essentiels de leur gestion :

- Sur le plan de l'adressage : elle offre à l'utilisateur un adressage uniforme de ses variables mais l'interprétation en terme d'adresses locales dans les mémoires distribuées est optimisée grâce à une technique de pagination logique des adresses.

- Sur le plan des communications : nous utilisons des communications directes des éléments requis mais nous profitons au maximum des techniques de vectorisation, en liaison avec notre mécanisme de pagination logique pour limiter le nombre et le volume des communications.

La suite de cet article précise les mécanismes ci-dessus. La section 2 présente globalement l'environnement de programmation et d'exécution et explicite les différents niveaux d'abstraction. La section 3 décrit le modèle de programmation offert à l'utilisateur en l'illustrant par un exemple. La section 4 est consacrée à la mise en œuvre de la bibliothèque de gestion des tableaux distribués et montre donc les techniques d'adressage et de communication développées. La section 5 conclut l'article.

# 2 Structuration de l'environnement de programmation et d'exécution

Rappelons tout d'abord que nous nous concentrons sur la gestion des structures de données. Les autres aspects de la programmation, notamment la gestion des processus parallèles, sont en dehors de la portée de cet article.

Au niveau le plus externe, nous allons offrir à l'utilisateur un environnement de programmation où il peut déclarer des structures de données quelconques et les rendre accessibles à plusieurs processus, ceci en utilisant la notation usuelle du langage de programmation choisi. Les structures de données étant a priori accessibles en lecture et en écriture par les différents processus, divers mécanismes de protection et protocoles de contrôle de cohérence seront proposés aux concepteurs d'applications. Ces outils seront mis en œuvre grâce à une (ou des) bibliothèque(s). La bibliothèque d'objets partagés Phosporus [5] offre un bon exemple de protocoles envisageables dans ce cadre. Cet aspect de programmation «haut niveau», qui correspond aux niveaux 5 et 6 de la figure 1, n'est pas figé dans la réalisation actuelle de notre environnement : on peut envisager plusieurs versions selon les langages source choisis.

Les couches de programmation haut niveau reposent sur le système de gestion de structures de données distribuées décrit dans cet article. La bibliothèque de tableaux distribués CIDRE présente au niveau 4, permet de déclarer une structure de données de type tableau multi-dimensionnel et d'en préciser la décomposition logique en fonction d'une architecture répartie. La décomposition du tableau s'exprime de manière similaire à ce que l'on trouve dans un langage comme HPF [7].

Les accès à une telle structure de données se font en utilisant la notation usuelle indicée. Les indices sont calculés globalement en fonction des bornes du tableau, indépendamment de la répartition choisie. C'est là le point fort de la méthode de programmation proposée : les références aux tableaux sont strictement les mêmes qu'avec une programmation pour mémoire partagée. L'exemple de la figure 2 témoigne de cette notation.

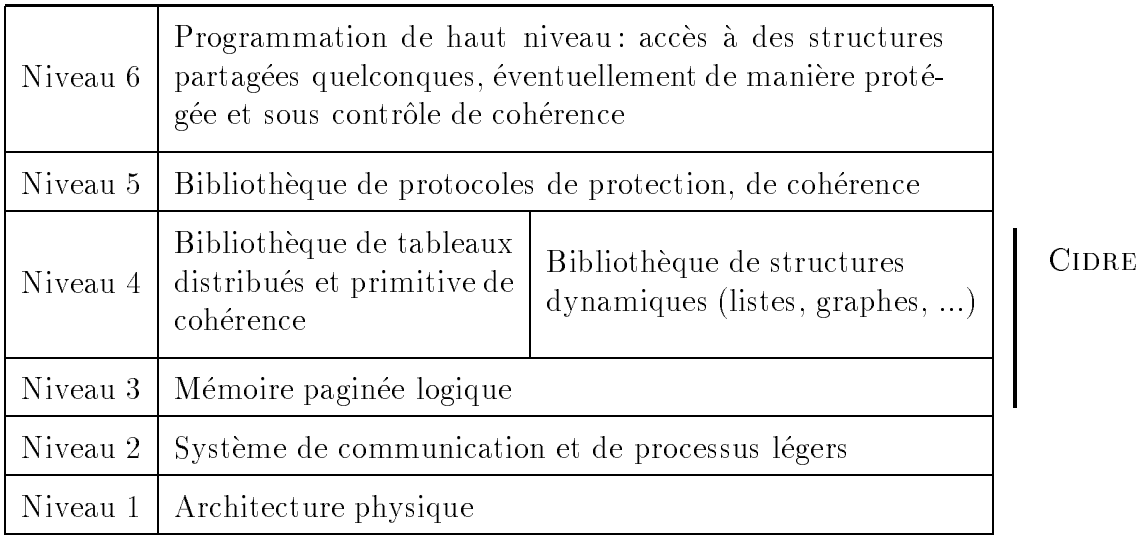


FIG. 1 – *Structuration de l'environnement*

La bibliothèque CIDRE met à disposition une primitive de synchronisation nommée `coherence` qui permet à des groupes de processus parallèles qui partagent en lecture et écriture une même structure de données de se synchroniser pour obtenir une vision unique cohérente de la structure. Cette primitive, détaillée au paragraphe 3, sert de base à la construction de protocoles plus évolués du niveau 5.

Au niveau 4, d'autres bibliothèques pourront être construites pour traiter des structures de données de type différent du tableau.

La mise en œuvre de la bibliothèque CIDRE repose sur un mécanisme que nous avons appelé *mémoire paginée logique* (niveau 3). Ce niveau 3 gère la distribution des tableaux dans les différentes mémoires locales. Un tableau distribué est décrit comme un ensemble de *pages logiques* dont la taille est fonction de celle du tableau. Les processeurs possédant une partie du tableau dans leur mémoire locale disposent d'un descripteur du tableau sous forme d'une table des pages logiques. On y trouve l'implantation du tableau. Les accès réalisés au niveau 4 sous forme d'indices globaux sont interprétés au niveau 3 en fonction de la structuration en pages. Nous verrons au paragraphe 4 que cette conversion est très rapide, ce qui fait l'intérêt de la méthode.

Enfin, en fonction de la localisation de la page adressée, l'accès peut être local au processeur ou distant. Dans ce cas, on fait appel au niveau 2 qui est celui du système de base de la machine cible et que l'on suppose offrir un mécanisme de communication entre processus.

La notion de processus légers bien que non indispensable, facilite la mise en œuvre de la bibliothèque de tableaux distribués car elle permet le recouvrement entre différentes activités. Enfin, le niveau 1 représente l'architecture physique.

Quelques bibliothèques d'objets partagés ont été conçues récemment [1, 5, 3, 12, 4]. Elles se situent à différents niveaux d'interface avec le programmeur d'applications et proposent différents protocoles de gestion de cohérence. Par rapport à ces bibliothèques, la bibliothèque CIDRE apporte des fonctionnalités au niveau des aspects d'implantation des structures de données et de gestion des communications, domaines peu explorés dans les bibliothèques citées ci-dessus.

# 3 Le modèle de programmation

Nous précisons ici l'interface de la bibliothèque offerte au programmeur et le style de programmation qui en découle.

Le modèle de programmation repose sur l'existence de processus parallèles. La décomposition en processus est du ressort du concepteur de l'application. Les processus peuvent être, comme c'est souvent le cas lors de l'utilisation d'architectures fortement parallèles, bâtis sur le même modèle. Typiquement nous avons un code SPMD (Single Program Multiple Data) dans lequel chaque processus, selon son identité, détermine le travail qu'il doit faire.

L'exemple illustre ce cas. $P$ processus exécutent le code décrit (qui représente un algorithme de type Jacobi). Les variables partagées sont déclarées en appelant la fonction de bibliothèque `create`. Cette fonction prend en paramètre les dimensions des tableaux puis leur répartition, qui peut être cyclique ou par blocs, sur les $P$ processeurs de l'architecture. Dans notre exemple, les matrices $A$ et $B$ de taille $N \times N$ sont créées. Elles sont toutes deux décomposées en blocs de taille $N/P \times N$.

L'adressage des variables partagées est global, sans mention explicite de la localisation de l'élément de tableau référencé, comme le montre l'affectation

$\texttt{write}(B, i, j, f(\texttt{read}(A, i+1, j), \texttt{read}(A, i-1, j), \texttt{read}(A, i, j+1), \texttt{read}(A, i, j-1)))$

où $f$ est une fonction prédéfinie.

Dans cet exemple le processeur $P_i$ possède le bloc de lignes $(N/P \times i)$ à $(N/P \times (i+1)-1)$. Pour effectuer ses calculs à l'étape $k$, il a besoin de la ligne $(N/P \times i - 1)$, après qu'elle ait été calculée par le processeur $P_{i-1}$ à l'étape $k-1$, et de la ligne $(N/P \times (i+1))$ calculée par $P_{i+1}$ à l'étape $k-1$. Réciproquement : $P_{i-1}$ a besoin de la ligne $(N/P \times i)$ et $P_{i+1}$ a besoin de la ligne $(N/P \times (i+1) - 1)$.

Pour assurer que chaque processeur travaille sur des données à jour à l'étape $k$ (c'est-à-dire résultant de l'étape $k-1$), nous utilisons deux opérations de synchronisation–cohérence par processus. La première coordonne les processeurs $P_i$ et $P_{i-1}$ pour la mise à jour des lignes $(N/P \times i - 1)$ et $(N/P \times i)$ et la fourniture à chacun d'eux de la même vision (à jour) de ces lignes ; la seconde coordonne les processeurs $P_i$ et $P_{i+1}$ pour l'obtention d'une vision cohérente et à jour des lignes $(N/P \times (i+1) - 1)$ et $(N/P \times (i+1))$.

D'une manière générale, la fonction `coherence` précise les éléments d'un tableau partagé (section de tableau spécifiée par un triplet borne inférieure–borne supérieure–pas dans chaque dimension) qui sont à rendre cohérents pour le groupe de processus cités en dernier paramètre de cette fonction. La sémantique de la fonction `coherence`, appliquée à une zone `Z` d'une variable partagée par un groupe `G` de processus est la suivante : Synchronisation des processus de `G` afin de prendre en compte toutes les opérations d'écriture qui ont été faites par chacun d'eux sur des éléments de `Z` depuis la dernière opération de `coherence` (ou depuis le début de l' exécution) ; diffusion de la version à jour à tous les processus de `G`.

---

**process** `myself`

$A$ = `create`('A', $N$, $N$, $N/P$, $N$)
$B$ = `create`('B', $N$, $N$, $N/P$, $N$)

*prev_set* = {`myself`, `myself`−1}
*next_set* = {`myself`, `myself`+1}

*my_first_line* = $N/P$*`myself`
*my_last_line* = $N/P$*(`myself`+1) −1

**for** $k$=1 **to** *nloop*

**if** (`myself` ≠ 0)
`coherence`($A$, *my_first_line*-1,*my_first_line*,1,0,N-1,1, *prev_set*)

**if** (`myself` ≠ $P-1$)
`coherence`($A$, *my_last_line*,*my_last_line*+1,1,0,N-1,1, *next_set*)

**for** $i$ = *my_first_line* **to** *my_last_line*
**for** $j = 0$ **to** $N-1$
`write`($B, i, j$, $f$(`read`($A, i+1, j$), `read`($A, i-1, j$), `read`($A, i, j+1$), `read`($A, i, j-1$)))

**for** $i$ = *my_first_line* **to** *my_last_line*
**for** $j = 0$ **to** $N-1$
`write`($A, i, j$, `read`($B, i, j$))

---

FIG. 2 – *Exemple de code : algorithme de Jacobi*

# 4 Mémoire paginée logique

Le niveau 3 de la bibliothèque offre une gestion des accès aux tableaux distribués. Les mécanismes mis en œuvre dans cette gestion sont issus de ceux utilisés dans le compilateur HPF Pandore [2]. Ils sont fondés sur la pagination logique des tableaux dirigée par la distribution en blocs rectangulaires qui a été spécifiée par l'utilisateur lors de la création du tableau. L'objectif est de fournir un accès élémentaire rapide tout en maintenant à un niveau raisonnable le surcoût mémoire induit par la représentation [11].

L'espace d'adresses multi-dimensionnel défini par chaque tableau est linéarisé et découpé en pages. Ces pages sont utilisées pour stocker les données locales et les copies temporaires des données distantes.

Les éléments sont accédés uniformément : à partir du vecteur d'indices globaux sont calculés un numéro de page et un offset dans cette page. Ce couple $(PG, OF)$ permet d'accéder, via la table des pages stockée sur chaque processeur, à l'élément mémoire correspondant.

Pour chaque tableau, on définit une direction des pages — *i.e.* une fonction de linéarisation — et une taille des pages en fonction des paramètres de distribution du tableau. La direction des pages correspond à la dimension selon laquelle la taille des blocs est la plus grande. On choisit comme taille de page une puissance de deux afin d'accélérer les calculs d'adresse : le calcul du couple $(PG, OF)$ ne fait intervenir que des opérations logiques peu coûteuses (décalages, masques). La figure 3 illustre les deux cas qui peuvent se présenter :

- S'il existe une dimension non distribuée, la taille des pages est égale à la première puissance de deux supérieure à la taille du tableau dans cette dimension. Le calcul du couple $(PG, OF)$ est alors très efficace (identité dans le cas 2D).

- Si toutes les dimensions sont distribuées, on prend comme taille des pages la puissance de deux inférieure à la taille des blocs dans leur plus grande dimension. Il se peut alors qu'une page couvre une frontière de bloc. Dans ce cas, chacun des deux processus impliqués n'est responsable que d'une partie de la page.

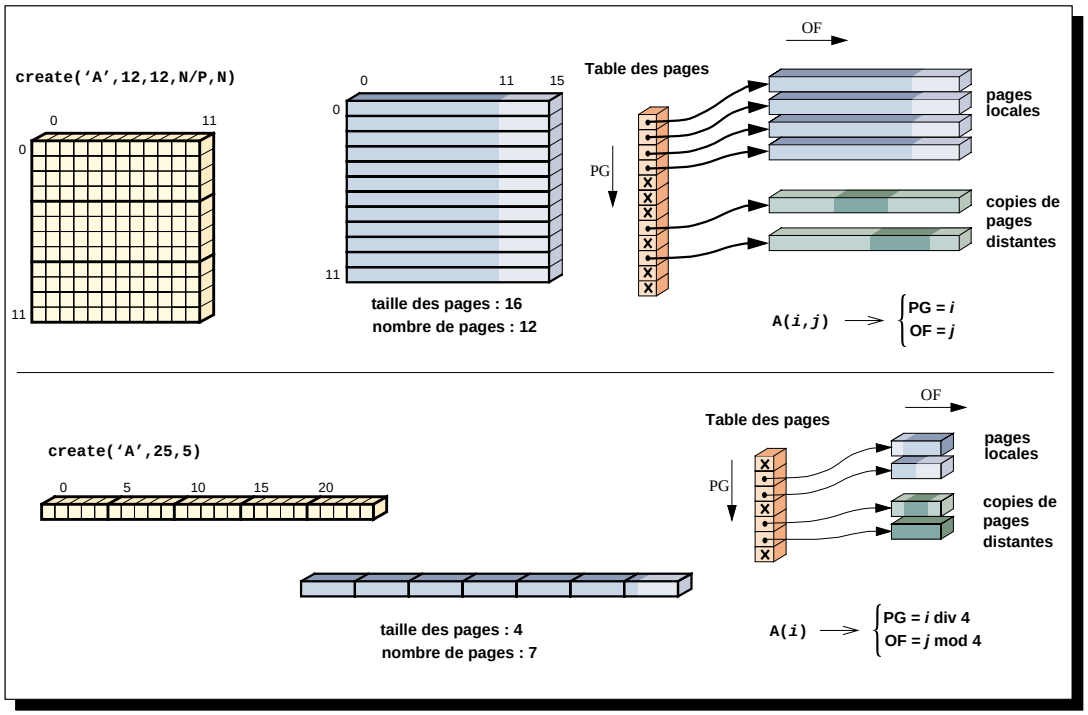


FIG. 3 – *Pagination logique des tableaux*

Outre les accès élémentaires efficaces, la gestion paginée des tableaux permet l'optimisation des communications qui peuvent être mises en jeu lors de l'opération de synchronisation-cohérence.

Les communications sont organisées en *segments* (un segment est une portion contiguë au sein d'une page). Pour les gros segments, on utilise des communications directes dans lesquelles on transfère une zone mémoire contiguë à la fois chez l'émetteur et chez le récepteur sans qu'il soit nécessaire de passer par un tampon de communication. Quant aux petits segments, ils peuvent être agrégés dans un tampon plus grand afin de minimiser le nombre de messages transmis s'il se révèle plus efficace de transférer de gros messages plutôt que plusieurs petits. Par ailleurs lors de la préparation des communications, on élimine les transferts multiples de la même variable élémentaire. Une description détaillée de ces mécanismes peut être trouvée dans [10].

# 5 Conclusion

La bibliothèque de gestion de tableaux distribués que nous venons de décrire est en cours de réalisation. Certains éléments n'ont pas encore pris leur forme définitive : par exemple, le langage d'interface (syntaxe des primitives telle que `coherence...`) est susceptible d'évoluer. Par ailleurs, quelques points plus fondamentaux sont à l'étude, pour lesquels plusieurs solutions sont explorées : il s'agit notamment du test qui permet de savoir si une référence correspond à une donnée présente localement ou non. Il est nécessaire de trouver une implantation efficace de ce test pour obtenir de bonnes performances d'accès, ou bien d'en minimiser le nombre.

Par contre, d'autres aspects comme la gestion par pages logiques et la mise en œuvre de la cohérence sur un modèle inspiré de la «règle des écritures locales» bénéficient de notre expérience acquise lors du développement de l'environnement de compilation Pandore et ont prouvé leur efficacité. Nous espérons donc pouvoir mettre à la disposition d'une plus vaste communauté d'utilisateurs ces outils de gestion de tableaux à la fois efficaces et pratiques à utiliser car ne nécessitant pas l'introduction de communications explicites pour le programmeur.

# Références


[1] J.-M. Adamo. Arch, an object-oriented library for asynchronous and loosely synchronous system programming. Technical Report 228, Cornell Theory Center – Ithica, NY 14853-3801, December 1995.

[2] F. André, M. Le Fur, Y. Mahéo, and J.-L. Pazat. The Pandore Data Parallel Compiler and its Portable Runtime. In *International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe '95*, number 919 in LNCS, Milan, Italie, mai 1995. Springer Verlag.

[3] Thomas Brandes. Adaptor : The distributed array library. Technical report, German National Center for Computer Science (GMD) – St-Augustin, Germany, April 1993.

[4] W.W. Carlson and J.M. Draper. Distributed data access in AC. In *ACM PPOPP Santa Clara, CA, USA*, pages 39–47, 1995.

[5] R. C. Dantart, I. Demeure, P. Meunier, and V. Bartro. Phosphorus : a tool for shared-memory management in a distributed environnment. Technical Report 95D003, E.N.S.T Paris, Departement Informatique, March 1995.

[6] M.R. Eskicioglu. A Comprehensive Bibliography of Shared Distributed Memory. *ACM Operating Systems Review*, 30(1):71–96, January 1996.

[7] High Performance Fortran Forum. High Performance Fortran Language Specification, Version 1.0. Technical report, Rice University, Houston, Texas, 1993.

[8] Z. Lahjomri and T. Priol. Koan: a Shared Virtual Memory for the iPSC/2 Hypercube. In *2nd Joint International Conference on Vector and Parallel Processing, CONPAR 92 - VAPP V*, number 634 in LNCS, Lyon, septembre 1992. Springer Verlag.

[9] K. Li and R. Shaefer. A Hypercube Shared Virtual Memory System. In *International Conference on Parallel Processing*, University Park, Pennsylvanie, 1989.

[10] Y. Mahéo. *Environnements pour la compilation dirigée par les données : supports d'exécution et expérimentations.* PhD thesis, IFSIC / Université de Rennes I, juillet 1995.

[11] Y. Mahéo and J.-L. Pazat. Distributed Array Management for HPF Compilers. In *High Performance Computing Symposium*, Montréal, Canada, juillet 1995.

[12] J. Nieplocha, R.J. Harrisson, and R.J. Littlefield. Global arrays : A portable "shared-memory" programming model for distributed memory computers. In *Supercomputing*, pages 1–9, November 1994.