**RENPAR/CFSE/SYMPA**
**Paris, cité des sciences et de l'industrie, 24 – 27 avril 2001**

# Nouvelles fonctionnalités pour le parallélisme de données. *Partie 1.*

Bertrand Ducourthial*, Nicolas Sicard**

*Laboratoire Heudyasic
Université de Technologie de Compiègne,
BP 20529, 60205 Compiègne, France,
Bertrand.Ducourthial@utc.fr

**Institut d'Électronique Fondamentale
Université de Paris-Sud,
bât. 220, 91405 Orsay, France
sicard@ief.u-psud.fr

---

**Résumé**
Nous présentons ici la première partie d'une étude qui porte sur l'introduction de nouvelles fonctionnalités en parallélisme de données. Nous proposons un codage à base de graphes pour les données irrégulières, et nous expliquons comment simplifier les calculs en agrégeant ces graphes. Nous introduisons aussi une nouvelle primitive, qui effectue un calcul asynchrone sur ces structures.

**Mots-clés :** parallélisme de données, structures de données, asynchronisme.

---

## 1. Introduction

### 1.1. Programmation à parallélisme de données

Le parallélisme de données tente de tirer avantage de l'exécution simultanée du même traitement sur un ensemble de données. Si l'environnement séquentiel travaille sur des scalaires (*e.g.*, un pixel), l'environnement à parallélisme de données travaille, lui, sur des ensembles de scalaires, organisés, par exemple, en tableaux (*e.g.*, une image). Ainsi, pour seuiller une image avec un seuil $s$, nous programmons une double boucle imbriquée sur $x$ et $y$ pour calculer $I[x, y] = \min(I[x, y], s)$. En parallélisme de données, nous écrirons $I = \min(I, s)$.

L'origine du travail sur tableaux vient du calcul scientifique et des architectures vectorielles (*e.g.*, Cray 1), qui furent optimisées pour ces structures et les calculs associés [9]. Le modèle à parallélisme de données a ensuite été intensivement étudié [13, 18, 1, 15], et des machines parallèles SIMD ont été proposées (*e.g.*, CM 200 en 1990). Mais l'augmentation de la puissance des processeurs standard a rendu caduque l'idée d'assembler de nombreux processeurs simples. Les architectures SIMD sont maintenant réservées à des domaines bien précis, tel que, par exemple, le traitement d'images. Ainsi les processeurs standard actuels confient certains traitements à des circuits de type SIMD : technologie MMX/SSE/KNI dans les Pentium d'Intel et 3DNow! dans les Athlon d'AMD, processeurs AltiVec de Motorola et MAJC de Sun, Playstation2 de Sony.

### 1.2. Intérêt du parallélisme de données

Même si le parallélisme de données n'est plus considéré comme un modèle de programmation généraliste, il reste utilisé et apprécié. Outre des facteurs historiques (existence de codes scientifiques basés sur des matrices), on peut avancer plusieurs explications pour expliquer l'intérêt que suscite ce type de parallélisation.

Le premier facteur déterminant est sa simplicité. Le nombre de constructions est faible [4]. Et l'on passe aisément d'un raisonnement sur des scalaires à un raisonnement sur des tableaux. Comme le montre notre exemple introductif, et contrairement au parallélisme de contrôle, le parallélisme de données simplifie généralement l'algorithmique : on dit qu'il offre plus d'*expressivité*.

Un autre facteur est celui de la preuve de programme. La simplicité des constructions data-parallèles autorise des schémas de preuves semblables à ceux qu'il est possible de faire sur des programmes séquentiels [4, 3].

Ceci est à opposer au modèle à passage de messages, considéré comme l'«assembleur du parallélisme» dans [6] : sa complexité conduit à multiplier le nombre d'erreurs, qui ne sont pas toujours simples à détecter.

Enfin, suivant [2], on peut conclure sur le fait que le parallélisme de données apporte au modèle à parallélisme de contrôle tel que CSP ce que le `while` du Pascal a apporté aux langages abusant des `goto`. Ainsi, les programmes Occam sont des suites parallèles de phases séquentielles (`PAR` de `SEQ`), tandis que les programmes à parallélisme de données sont des listes séquentielles de phases parallèles (`SEQ` de `PAR`). Le parallélisme de données apporte ainsi une structuration au parallélisme, qui fait défaut au modèle à passage de messages.

### 1.3. Difficultés

Sauf à utiliser une architecture SIMD adéquate, les données manipulées par le programmeur sont généralement de plus grande taille que le nombre de processeurs dans la machine utilisée. Les langages offrent donc au programmeur une machine virtuelle idéale, correspondant à la taille de ses données. Mais il faut ensuite plaquer cette machine virtuelle sur les processeurs physiques. On trouve alors presque autant de solutions que de langages data-parallèles, qui se répartissent en deux grandes classes : ceux dont la machine virtuelle est d'*instanciation* et ceux pour lesquels elle est d'*alignement*. Les premiers n'admettent que des objets ayant la topologie de la machine virtuelle (langages C$\star$, DPCE, Hyper C), tandis que les seconds autorisent des objets de formes diverses, et pour lesquels il faudra préciser la taille et le placement dans la topologie de la machine virtuelle (Fortran 90, HPF [16], Idole [14]).

Idéalement, le parallélisme de données n'aimerait s'occuper que de données très régulières, toutes de la même taille, et qui se répartiraient facilement sur les processeurs de la machine. Pour prendre en compte cette irrégularité, les langages data-parallèles à machine virtuelle d'instanciation ont proposé des *collections* sur des machines virtuelles régulières (C$\star$), ou des machines virtuelles irrégulières (Parallaxis, Hyper C). Les langages à machine virtuelle d'alignement ont proposé des objets parallèles dynamiques [8] ou des tableaux dynamiques, comme dans Fortran 90, qui sont placés dynamiquement sur les processeurs par des directives dans HPF. Enfin, d'autres solutions plus générales ont été proposées, telles que les langages HPF 2 et Vienna Fortran [16], les extensions pour matrices creuses [20], l'utilisation d'arbres [5], les `shape` dans Idole [7], les objets fragmentés dans ACADA [22], *etc.*

Lorsqu'il ne s'agit pas d'une architecture SIMD, les mises en œuvre sont confrontées au grain très fin des calculs, ainsi qu'à leur forte synchronisation. Répartir des calculs trop élémentaires n'est pas très efficace. En outre, lorsque ces calculs nécessitent de nombreuses synchronisations, le contrôle de la parallélisation devient trop important par rapport au gain qu'elle pourrait offrir. La parallélisation efficace d'applications irrégulières (*e.g.*, segmentation d'images) n'est donc pas simple.

### 1.4. Notre contribution

Afin d'étendre son champ d'applications, nous introduisons de nouvelles fonctionnalités en parallélisme de données. Cet article (1ère partie) est consacré à leur définition.

Nous montrons comment des objets irréguliers telles que les régions d'une image peuvent être efficacement codés par des graphes (`snet`) dans une machine virtuelle d'instanciation. En regroupant des ensembles de nœuds du `snet`, on forme des graphes plus petits (`vnet`), qui simplifient la programmation. Par exemple, ils évitent de reporter certains traitements concernant les régions d'une image au niveau des pixels. Ils permettent aussi d'optimiser les calculs lorsque la machine virtuelle possède plus de processeurs que la machine physique.

Les circuits spécialisés asynchrones sont plus rapides et consomment moins que les circuits électroniques synchrones. De même, la plupart des programmes s'exécutant sur machines parallèles sont plus rapides lorsqu'ils comportent moins de synchronisation [21]. Or, le parallélisme de données sous-entend une forte synchronisation des calculs élémentaires. Nous montrons qu'un certain nombre de synchronisations peuvent être évitées en data-parallélisme. Par exemple, le calcul de la distance à un pixel dans une image peut être réalisé sans synchronisation, et le calcul de la distance à plusieurs points tire avantage de la dé-synchronisation des traitements élémentaires. Nous introduisons alors la primitive de calcul asynchrone `scour`.

## 2. Les structures de données `snet` et `vnet`

### 2.1. Structures de données

Nous considérons une machine virtuelle d'instanciation codée par un graphe orienté, dit *graphe de base* `anet`, dont les nœuds correspondent aux processeurs virtuels, et les arcs aux liens de communication unidirectionnels

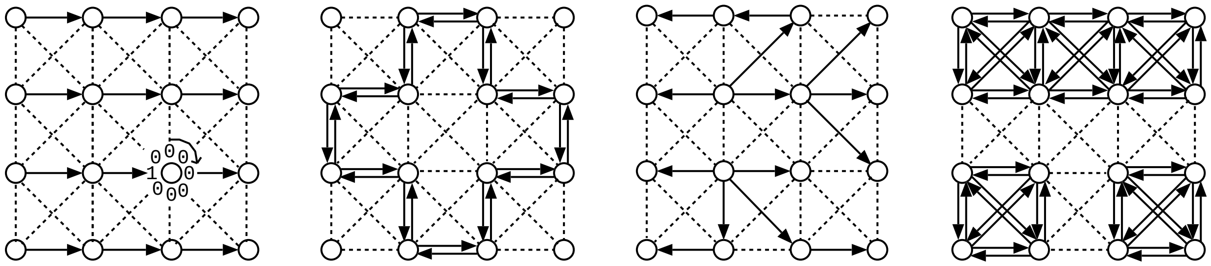



FIG. 1 – *Représentation de structures irrégulières par un* `snet`*, codé par une* `pvar<mask>` .

entre processeurs voisins. Tout type de topologie peut être employée, mais nous considérerons dans cet article une grille orientée symétrique (*i.e.*, à tout arc $(u, v)$ correspond l'arc $(v, u)$) avec diagonales (*cf.* figure 1).

Une variable parallèle `pvar<T>` est un ensemble de données de type `T` réparties à raison d'une par nœud du graphe de base. Ainsi, une image `I` de pixels codés sur des octets est déclarée par : `pvar<ushort> I`. Un objet irrégulier, tel qu'un contour ou une région d'une image, correspond à un sous-graphe du graphe de base (*cf.* figure 1). Un tel sous-graphe, appelé `snet`, est obtenu en masquant certains liens entrants au niveau de chaque nœud dans le graphe de base. Il peut donc être codé, sur chaque nœud de la grille, par un mot de 8 bits appelé `mask` : un bit à 0 (resp. 1) indique que le lien est (resp. n'est pas) masqué ; les liens sont numérotés dans le sens des aiguilles d'une montre en partant de 12h (figure 1-a). L'ensemble de ces `mask` définit une `pvar<mask>`. Ainsi, les structures irrégulières se manipulent comme les autres variables parallèles. Des primitives spécifiques, telles que `invert`, `link-with-equals`, `symmetrize`, `spanning-tree` sont aisément définies sur ces `pvar`.

### 2.2. Structures virtuelles

Les algorithmes data-parallèles travaillent avec des `pvar` de données élémentaires. Or, il s'avère qu'ils n'ont pas toujours besoin de se référer à l'ensemble des données d'une `pvar`. Par exemple, une fois que les régions d'une image `I` sont identifiées et codées par un `snet`, il serait souhaitable — lorsque c'est possible — d'appliquer une succession de traitements directement à des «objets régions» plutôt qu'à l'ensemble des données élémentaires qu'elles représentent. L'algorithme gagnerait en clarté, et la programmation en expressivité.

L'environnement pourrait aussi y gagner en efficacité. En effet, la programmation à parallélisme de données sous-entend qu'un même traitement est appliqué à un ensemble de données défini par un contexte (`where(condition)...`), voire à la totalité des données de la pvar. Lorsqu'il y a moins de processeurs physiques que de processeurs virtuels dans la machine virtuelle d'instanciation, chaque processeur physique doit traiter les uns après les autres les processeurs virtuels qui lui échoient. En travaillant sur des données moins nombreuses (régions), et non sur l'ensemble des données élémentaires (pixels), un grand nombre de calculs seront évités.

Partant d'un `snet` quelconque, nous définissons un *graphe virtuel* `vnet` en regroupant en un seul sommet les composantes fortement connexes du `snet` (graphe réduit) par la primitive `virtualize`. Une fois défini, un `vnet` joue le rôle de machine virtuelle d'instanciation, à la manière du graphe de base. Il permet de définir des `vpvar`, qui sont des variables parallèles composées d'une donnée élémentaire par nœud du `vnet`. Une telle `vpvar` peut être initialisée grâce à une $\beta$-réduction composante par composante d'une `pvar` (*e.g.*, `MaxReduction`, *cf.* figure 2). De même, nous définissons des `vsnet` qui sont des sous-graphes du `vnet`, afin de coder des formes quelconques dans le `vnet`, et de représenter, par exemple, des régions fusionnées (*cf.* figure 2). Les primitives spécifiques aux `snet` (*e.g.*, `link-with-equals`) sont étendues aux `vsnet`.

Contrairement au graphe de base, il peut y avoir plusieurs `vnet` différents dans un même programme. On précise donc avec `with(vnet)...` le `vnet` auquel se réfère la `vpvar` ou le `vsnet` manipulé.

## 3. La primitive `scour`

### 3.1. Calcul sur les `snet`

Dans un programme data-parallèle, certains calculs sur les `pvar` ne nécessiteront pas de «communication» : il s'agit des opérations unaires (*e.g.*, `I = max(s, I)`), ou des opérations $n$-aires portant sur des `pvar` «alignées» (*e.g.*, `I = J + K`). Mais d'autres calculs impliqueront des communications. Par exemple, dans C$\star$, le moyen-

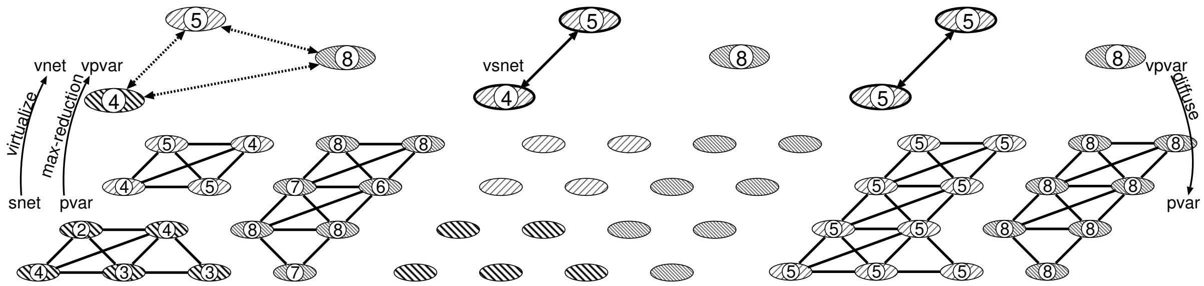



FIG. 2 – *Exemple d'utilisation des* **vnet** *: construction d'un vnet de 3 nœuds à partir d'un* **snet** *de 16 nœuds (*`virtualize`*); construction d'une* **vpvar** *sur ce* **vnet** *avec une* **MaxReduction***; calcul dans le* **vnet** *: création d'un* **vsnet** *regroupant des régions de valeurs proches ; report des résultats dans le* **snet** *et la* **pvar** *(*`diffuse`*).*

nage d'un pixel avec ses quatre plus proches voisins dans une `cell` en forme de grille est codé par : `I = ([.-1][.]J + [.+1][.]J + [.][-1]J + [.][+1]J + J)/5`. Lorsqu'un tel calcul applique un opérateur associatif et commutatif (ici l'addition) sur un voisinage (ici les 4 plus proches voisins dans la grille), il peut être exprimé via un `snet` (défini ici avec le `mask 10101010`, *cf.* figure 3-a). Nous introduisons donc la primitive `step-scour`, qui applique un opérateur sur une `pvar` selon un voisinage codé par un `snet`. Le calcul de la moyenne devient : `snet vois = 10101010; I = +-step-scour(J, vois); I = I / 5`.

L'utilisation des `snet` pour structurer les calculs ne se limite pas au voisinage direct des nœuds de la machine virtuelle `anet`. En effet, un `snet` peut, par exemple, coder des lignes : `snet lignes = 00000100` (lignes est → ouest dans la grille avec diagonales, *cf.* figure 1-a). Nous introduisons la primitive `scour`, qui correspond à une cascade de `step-scour` débutant sur les nœuds n'ayant pas d'antécédent. Ainsi, la recherche du plus grand pixel d'une ligne de l'image `J` s'exprime par `I = max-scour(J, lignes)` (le résultat est disponible dans la dernière colonne de `I`, *cf.* figure 3-b). On retrouve ici la primitive `scan` [1].

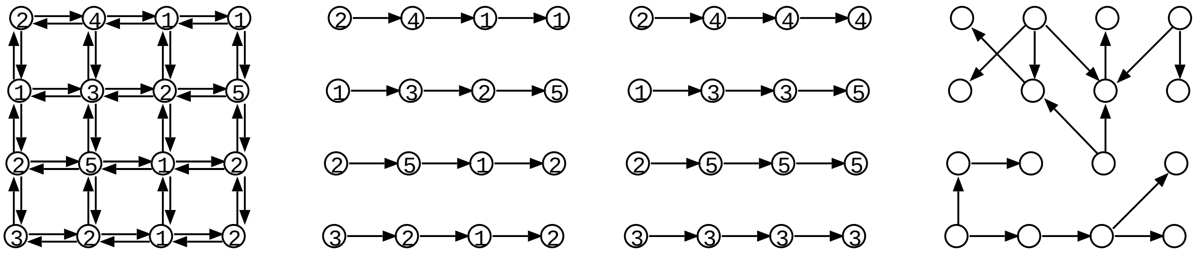
Les `snet` permettent donc d'exprimer, dans la machine virtuelle `anet`, les calculs sur les voisinages, mais aussi les `scan`, et ainsi de retrouver la richesse de l'algorithmique basée sur les calculs préfixes [1, 15]. Mais les `snet` peuvent coder des structures plus complexes que les lignes, et l'on peut se demander sur quels graphes les `scan` peuvent être généralisés de sorte que le résultat sur un nœud soit égal au résultat de l'application de l'opérateur utilisé sur ses antécédents (comme c'est le cas pour les `scan`). Nous avons montré dans [11] que les `scan` pouvaient être étendus pour tout type d'opérateur associatif et commutatif sur des graphes ne comportant pas deux chemins différents entre deux nœuds. Ces graphes, dits *préfixes*, correspondent à certaines connexions d'arborescences (*cf.* figure 3-c).

Par définition, les graphes préfixes n'ont pas de circuits. Lorsque le graphe codé par le `snet` présente un circuit, nous imposons à l'opérateur utilisé d'être idempotent, pour que le sens du calcul soit évident (il n'est pas simple de donner un sens au calcul d'une addition sur un circuit par une `+-scour`). Dans le cas d'un opérateur associatif, commutatif et idempotent (*s-opérateur*), tel que $\max$, $\min$, $\mathrm{pgcd}$, $\cup \ldots$, la même valeur est obtenue sur tous les nœuds d'une même composante fortement connexe. Cela correspond à une $\beta$-réduction suivie d'une $\alpha$-extension [13]. Ainsi, pour diffuser la valeur du plus grand pixel dans chaque région (codées par le `snet régions`) d'une image `J`, on écrira `I = max-scour(J, régions)`. Pour calculer la somme (non idempotente) des pixels de chaque région, on crée préalablement une arborescence couvrante (graphe préfixe) dans chaque région : `snet forêt = spanning-tree(régions); I = +-scour(J, forêt)`.

### 3.2. Calcul asynchrone

Comme nous l'indiquions au § 1, les circuits électroniques asynchrones présentent des avantages par rapport aux circuits synchrones. En effet, les communications d'un circuit synchrone sont dépendantes de l'horloge, dont la période est calculée dans la pire des configurations (circuit le plus lent dans les conditions d'utilisation les plus pénalisantes). Au contraire, les communications dans un circuit asynchrone sont effectuées au plus tôt. En outre, les circuits asynchrones consomment moins, et ne nécessitent pas de câbler l'horloge. Ces remarques motivent les travaux sur les circuits SIMD asynchrones [12, 17].

Si l'utilisation d'une horloge permet d'appréhender plus facilement le fonctionnement d'un circuit, elle le ralentit fortement. De même, les supports d'exécution disponibles sur les machines parallèles telles que les




**a**.snet pour calculer la moyenne des 4 plus proches voisins avec step-scour.

**b**.la primitive max-scour sur un snet en lignes correspond à un max-scan (état initial à g., résultat à d.).

**c**.exemple de snet préfixe (sur lequel la primitive scour généralise les scans).

FIG. 3 – *Calculs sur les* `snet` *avec la primitive* `scour`*.*

PC multi-processeurs ou les grappes de PC sont intrinsèquement asynchrones (les processeurs ne calculent pas tous au même rythme). Pour faciliter leur utilisation, on utilise des barrières de synchronisation. Ces synchronisations simplifient la programmation, mais elles freinent considérablement l'exécution. Par exemple, les calculs matriciels parallèles ont de meilleures performances en limitant les synchronisations [21].

Ces constatations nous ont conduits à étudier l'intérêt d'une primitive de calcul asynchrone au sein même de la programmation à parallélisme de données. La primitive `scour` sous-entend un grand nombre de calculs élémentaires sur les nœuds de la machine virtuelle `anet`. Il apparaît en fait que ces calculs n'ont pas besoin d'être synchronisés. En effet, dans le cas où le graphe sous-jacent (codé par le `snet`) est préfixe, l'absence de synchronisation est transparente : les résultats progresseront depuis les nœuds racines vers les feuilles. Dans le cas où le graphe présente des circuits, la dé-synchronisation des calculs élémentaires est également transparente lorsque l'opérateur est associatif, commutatif et idempotent ($s$-opérateur) ; on retrouve là un résultat de [19] (*cf.* [11]). Nous avons étendus ces résultats aux $r$-*opérateurs* afin d'accroître le nombre d'opérateurs admissibles avec une primitive `scour` asynchrone [10]. Les $r$-opérateurs sont obtenus en di-symétrisant les $s$-opérateurs grâce à une fonction. Par exemple, l'opérateur $\mathrm{minc}$, défini par $\mathrm{minc}(x,y) = \min(x, y+1)$ est obtenu en déformant le $s$-opérateur $\min$ par la fonction $x \mapsto x+1$. Un $s$-opérateur est en fait un $r$-opérateur dont la fonction de déformation est l'identité. Nous avons montré que les calculs élémentaires d'une primitive `scour` pouvaient être désynchronisés dans le cas des $r$-opérateurs idempotents [10, 11].

Fort de ces résultats, nous définissons la primitive `scour` comme étant une succession de `step-scour` jusqu'à stabilisation des résultats. La terminaison doit maintenant être détectée. Sur une machine SIMD asynchrone, on dispose généralement d'un détecteur de terminaison [12]. Lorsqu'il y a moins de processeurs physiques que de nœuds dans la machine virtuelle (mise en œuvre sur PC multi-processeurs ou grappes de PC), il est nécessaire d'organiser l'exécution des `step-scour` afin d'éviter des calculs inutiles. Cette organisation inclut alors la détection de terminaison.

Si l'asynchronisme de la primitive `scour` présente des avantages en terme de mise en œuvre, elle en présente également sur le plan de l'algorithmique data-parallèle. Ainsi, le calcul de la distance à un point (*germe*) dans un plan discrétisé par une grille nécessite une succession de calculs élémentaires, afin que chaque point de la grille obtienne sa distance au germe. En programmation data-parallèle, la synchronisation de ces calculs élémentaires ralentirait fortement le calcul de la distance. Or, ce calcul peut être implémenté *via* une $\mathrm{minc}$-`scour` asynchrone [10] (l'opérateur $\mathrm{minc}$ est idempotent : $\mathrm{minc}(x,x) = x$). L'itération des calculs élémentaires se trouve donc remplacée par une seule primitive ; la programmation gagne en expressivité.

La primitive `scour` permet d'exploiter la forte connexité du `snet` sous-jacent lorsqu'elle est utilisée avec un $s$-opérateur (*e.g.*, $\min$), puisqu'un résultat unique est obtenu par composante fortement connexe. Elle permet d'exploiter la distance avec un $r$-opérateur binaire (*e.g.*, $\mathrm{minc}$), où la déformation du $s$-opérateur sera appliquée sur toutes les données entrantes de chaque nœud. Avec des $r$-opérateurs non binaires, on peut différencier le traitement appliqué à chaque donnée entrante ; on exploite alors les chemins du `snet` sous-jacent [11].

Outre une plus grande expressivité, l'asynchronisme de la primitive `scour` peut également conduire à d'intéressantes optimisations. Ainsi, certains problèmes s'apparentant à une optimisation globale impliquent que chaque nœud choisisse un voisin, sur la base d'un critère calculé sur plusieurs nœuds (*e.g.*, triangulation de Voronoï, segmentation d'images, forêt des meilleurs chemins). Le choix d'un nœud pour l'un de ses voisins peut être remis en cause par le changement d'un autre choix local. Ce type de problème peut conduire, dans un modèle très synchrone tel que le data-parallélisme, à de nombreuses itérations. Or, certains d'entre eux peuvent être exprimés par un r-opérateur et la primitive `scour` asynchrone. Dans ce cas, la remise en cause d'un choix local agit à la manière d'une «relaxation locale» ; les nœuds voisins se réajustent sans pour autant impliquer des itérations (souvent imbriquées) synchrones portant sur la totalité de la pvar.

## 4. Conclusion

Dans cet article, nous avons présenté la première partie d'un travail qui consiste à introduire de nouvelles fonctionnalités en parallélisme de données. Nous avons proposé un codage des données irrégulières avec des graphes (`snet`) dans une machine virtuelle d'instanciation (`anet`) de topologie quelconque. Ces graphes peuvent être codés *via* des `pvar`, ce qui rend leur manipulation efficace. Les `vnet` offrent un niveau d'abstraction supplémentaire en agrégeant les données.

Si ces structures de données élargissent encore le spectre des applications programmables en parallélisme de données, elles permettent également de structurer des calculs. Ainsi, la primitive `scour` effectue un calcul asynchrone sur ces graphes. Suivant l'opérateur ou le graphe sous-jacent, on retrouve les calculs de voisinage, les `scan`, les réductions ou les diffusions, ainsi que des calculs plus complexes (*e.g.*, distance), qui nécessitent habituellement plusieurs instructions data-parallèles.

## Bibliographie


1. G.E. Blelloch. *Vector Models for Data-Parallel Computing*. Artificial Intelligences Series. The MIT Press, 1990.
2. L. Bougé. *The Data Parallel Programming Model*, chapitre The data-parallel programming model: a semantic perspective, pages 4–26. LNCS-TS. Springer Verlag, 1996. G.-R. Perrin et A. Darte éditeurs.
3. L. Bougé, D. Cachera, Y. Le Guyadec, G. Utard, et B. Virot. Formal validation of data-parallel programs: a two-component assertional proof system for a simple language. *TCS*, 189(1-2):71–107, Décembre 1997.
4. L. Bougé et J.-L. Levaire. Control structures for data-parallel SIMD languages: semantic and implementation. *Future Generation Computer Systems*, 8:363–378, 1992.
5. F. Brégier. Proposition de gestion de l'irrégularité avec HPF2. In RenPar 10, 1998.
6. C. Calvin et L. Colombet. Introduction au modèle de programmation par processus communicants : deux exemples PVM et MPI. Rapport APACHE 12, IMAG, Grenoble, France, Juillet 1994.
7. J.-L. Dekeyser, B. Kokoszko, J.-L. Levaire, et P. Marquet. Irregular data-parallel objects in C++. In *VECPAR'96*.
8. J.-L. Dekeyser, D. Lazure, et P. Marquet. A geometrical data-parallel language. *ACM SigPlan Notices*, 29(4):31–40, Avril 1994.
9. J.-L. Dekeyser et P. Marquet. *The Data Paralel Programming Model*, chap. Supporting irregular and dynamic computations in data-parallel languages, pp. 197–219. LNCS-TS. Springer Verlag, 96. G.-R. Perrin et A. Darte éd.
10. B. Ducourthial. New operators for computing with *associative nets*. In Proceedings SIROCCO'98, pages 55–65. Carleton Scientific, Juin 1998.
11. B. Ducourthial. *Les réseaux associatifs, un modèle de programmation à parallélisme de données, pour algorithmes et donnés irréguliers, à primitives de calcul asynchrones*. Thèse de doctorat, Université Paris-Sud, 1999.
12. D. Dulac. *Contribution au parallélisme massif en analyse d'images : une architecture SIMD fondée sur la reconfigurabilité et l'asynchronisme*. Thèse de doctorat, Université Paris-Sud, 1996.
13. W. D. Hillis. *The Connection Machine*. The MIT Press, 1985.
14. B. Kokoszko. Intégration du modèle data-parallèle irrégulier IDOLE dans C++. In RenPar 8, 1996.
15. S. Lakshmivarahan et K.S. Dhall. *Parallel Computing Using the Prefix Problem*. Oxford University Press, 1994.
16. P. Marquet. Gestion de l'irrégularité dans HPF. In D. Barth, J. Chassin de Kergommeaux, J.-L. Roch, et J. Roman, editeurs, ICARE'97, chapitre 11, pages 231–250. Décembre 1997. École CNRS PRC PRS.
17. F. Robin. *Étude d'architectures VLSI numériques parallèles asynchrones pour la mise en œuvre de nouveaux algorithmes d'analyse et rendu d'images*. Thèse de doctorat, ENST Grenoble, 1997.
18. G.W. Sabot. *The Paralation Model*. Artificial Intelligences Series. The MIT Press, 1988.
19. G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.
20. M. Ujaldon, E.L. Zapata, B.M. Chapman, et H.P. Zima. Vienna-Fortran / HPF extensions for sparse and irregular problems and their compilation. *IEEE TPDS* 8(10):1068–1083, Octobre 1997.
21. A. Üresin, M. Dubois. Effects of asynchronism on the convergence rate of iterative algorithms. *JPDC*, 34:66–81, 1996.
22. L. Verbièse, N. Devesa, et P. Lecouffe. ACADA : Aide à la Conception et à l'exécution d'Applications Distribuées Adaptatives. In D. Méry et G-R. Perrin, éditeurs, RenPar 10, pages 99–102, 1998.