Architectures systoliques et parallélisme de données ; l'environment de programmation ReLaCS

Frédéric Raimbault, Patrice Quinton, Dominique Lavenier

▶ **To cite this version:**

Frédéric Raimbault, Patrice Quinton, Dominique Lavenier. Architectures systoliques et parallélisme de données ; l'environment de programmation ReLaCS. [Rapport de recherche] RR-1982, INRIA. 1993. inria-00074690

HAL Id: inria-00074690
https://inria.hal.science/inria-00074690

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

Architectures systoliques et parallélisme de données : l'environnement de programmation ReLaCS

Frédéric Raimbault, Patrice Quinton, Dominique Lavenier

N° 1982

Mai 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

INRIA
RENNES

Architectures systoliques et parallélisme de données : l'environnement de programmation ReLaCS

Frédéric Raimbault, Patrice Quinton, Dominique Lavenier *

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

Rapport de recherche n° 1982 — Mai 1993 — 16 pages

Résumé : Nous rappelons les concepts du calcul systolique et nous examinons les problèmes de mise en œuvre des solutions systoliques. Nous montrons comment adapter le modèle du parallélisme de données pour faciliter la programmation des algorithmes systoliques. Nous présentons le langage ReLaCS, en particulier ses opérateurs de communication synchrone avec lesquels les flôts de données systoliques peuvent être décrits simplement. Enfin nous décrivons l'organisation du compilateur de ReLaCS qui permet la génération de code pour des architectures SIMD, MIMD, et séquentielles.

Mots-clé : langage de programmation, langage ReLaCS, parallélisme de données, architectures SIMD, machine MicMacs, algorithmique systolique, distance de Levenshtein.

(Abstract: pto)

A paraître dans la revue TSI d'octobre 1993

*{raimbaul,quinton,lavenier}@irisa.fr

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

Data Parallelism and Systolic Architectures: the ReLaCS programming environment

Abstract: We recall the underlying concepts of systolic arrays and we study the implementation issues of systolic solutions. We show how the model of data parallelism can be adapted to simplify the task of programming systolic algorithms. We present the ReLaCS language, and we detail its synchronous communication operators. These operators allow systolic data flows to be easily expressed. Finaly, we describe the organization of the ReLaCS compiler. This compiler generates programs for SIMD and MIMD architectures.

Key-words: programming language, ReLaCS language, data-parallelism, SIMD architectures, MicMacs machine, systolic algorithms, Levenshtein algorithm.

1. Introduction

Les progrès technologiques réalisés en matière d'intégration permettent la conception de circuits spécialisés très complexes, et il est désormais fréquent de voir apparaître sur le marché des circuits dédiés de plusieurs centaines de milliers de portes.

L'exploitation de cette formidable densité d'intégration nécessite une parfaite maîtrise de la conception des circuits. Pour y parvenir, une solution, prônée par Mead and Conway [MEA 80] dès 1980, consiste à rechercher des structures répétitives obtenues par simple aboutement de cellules identiques. De cette façon des circuits très complexes peuvent être obtenus avec un effort de conception réduit.

Ce principe de conception est à la base des architectures dites systoliques [KUN 80]. Introduit en 1978 par Kung et Leiserson, ce modèle architectural est de plus en plus fréquemment utilisé pour la réalisation de processeurs intégrés spécialisés. Une architecture systolique (figure 1) est un réseau composé d'un grand nombre de cellules élémentaires identiques et interconnectées localement. Chaque cellule reçoit des données en provenance des cellules voisines, effectue un calcul, puis transmet les résultats à des cellules voisines un temps de cycle plus tard. Seules les cellules situées à la frontière du réseau communiquent avec le monde extérieur, c'est à dire une mémoire externe ou un ordinateur hôte.

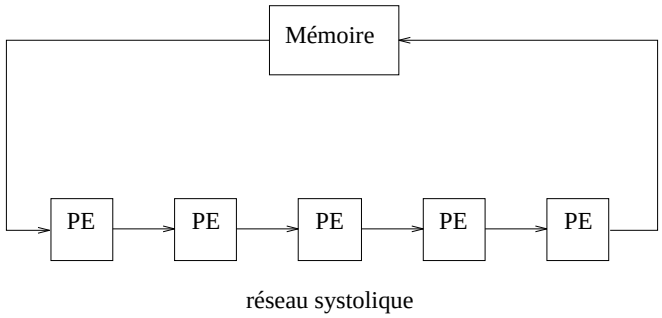

Figure 1. *Principe d'une architecture systolique*

Les cellules évoluent en parallèle, sous le contrôle d'une horloge globale : plusieurs calculs sont effectués simultanément sur le réseau, et on peut enchaîner la résolution de plusieurs instances du même problème. La régularité et la localité des algorithmes systoliques les rendent particulièrement bien adaptés à être intégrés sur silicium [QUI 89].

La conception d'un circuit systolique nécessite une simulation préalable, afin de s'assurer de sa validité au niveau fonctionnel. Mais il n'est pas rare qu'un circuit systolique atteigne une puissance de calcul de l'ordre du milliard d'opérations par seconde, ce qui impose d'avoir recours à une simulation parallèle. Les langages de description de matériel, comme VHDL [AIR 90], peuvent être employés pour décrire des architectures systoliques mais ces langages ne sont pas destinés à des simulations parallèles. Il faut donc avoir recours à des langages parallèles.

Les solutions possibles sont de trois types. On peut tout d'abord utiliser des langages dédiés. C'est le cas de APPLY [HAM 88] pour le traitement de bas niveau sur les images, ASSIGN [O'H 91] pour le traitement du signal, ou encore AL [TSE 90] pour le calcul matriciel. Ces langages ont été proposés pour programmer des calculs réguliers sur des machines parallèles. Il s'agit toutefois de langages trop particuliers pour fournir une solution acceptable.

Il est également possible de décrire un calcul systolique et ses entrées-sorties dans un langage parallèle basé sur les processus communicants comme Occam [Occ 88]. Un tel langage apporte toute la puissance d'expression du parallélisme de contrôle et des primitives de communication de base, mais est mal adapté à la programmation systolique, car il ne permet pas de définir de manière naturelle des structures régulières. Ces désavantages sont aussi ceux du langage W2 [LAM 89], le langage développé pour la machine Warp [ARN 85]. De plus W2 requiert la connaissance des détails de l'architecture, comme le nom des liens de communication.

Une troisième voie repose sur l'utilisation de langages à parallélisme de données [LAW 75, PER 85, CHR 90, PAR 92]. Ils offrent l'avantage d'être fondés sur un modèle de programmation synchrone et de voir les échanges entre processeurs d'une façon globale, comme des opérations sur des collections de données. Les langages à parallélisme de données les plus connus sont toutefois très liés au modèle d'architecture SIMD, et supposent l'existence d'un certain nombre de dispositifs matériels [TUC 90], par exemple un routeur global, un masque d'activité, des mécanismes de virtualisation, des opérateurs de réduction. Ils ne prennent pas en compte par ailleurs, les aspects entrées-sorties, qui sont comme on le verra, fondamentaux dans le calcul systolique.

Récemment, un langage fondé sur le parallélisme de données et appelé *New Systolic Language*, a été présenté [HUG 92]. Ce langage, construit à partir de C++, sépare la description des calculs de celle des communications. Cette séparation exige l'utilisation de deux langages différents, ce qui, à notre sens, ne simplifie pas la programmation.

La suite de cet article décrit l'utilisation du concept de parallélisme de données appliqué à la programmation des architectures systoliques. Dans une première partie nous présentons un exemple d'algorithme systolique et nous étudions les problèmes soulevés par sa programmation et son exécution sur une architecture parallèle. Nous exposons dans la seconde partie les éléments d'un environnement de programmation et de simulation fondé sur l'utilisation d'un langage appelé ReLaCS. Nous décrivons son modèle de programmation à parallélisme dirigé par les données, son modèle d'exécution dérivé du SIMD et sa compilation sur différentes machines.

2. Un exemple : le calcul de distances entre chaînes

Dans cette partie, nous décrivons un algorithme systolique permettant de calculer la distance de Levenshtein [WAG 76] entre chaînes de caractères. Cet algorithme est utilisé dans diverses applications de reconnaissance des formes – post-traitement des lecteurs optiques, correction de fautes de frappe, analyse de séquences biologiques, etc. – et sa simplicité permet d'illustrer les principaux problèmes liés à la programmation systolique. Nous détaillons dans le paragraphe 2.1 le principe de cet algorithme, puis nous en donnons une mise en œuvre sur une architecture bi-dimensionnelle (paragraphe 2.2). Le paragraphe 2.3 montre ensuite comment cette mise en œuvre peut être simulée sur une architecture parallèle linéaire, et s'attarde sur les particularités liées à la programmation d'une telle machine.

2.1. *La distance de Levenshtein*

Il s'agit de comparer une chaîne appelée chaîne de test avec une chaîne de référence, sachant que des erreurs d'insertion, d'omission ou de substitution de caractères sont éventuellement présentes. La distance de Levenshtein entre deux chaînes de caractères est le nombre minimun de substitutions, d'omissions et d'insertions de caractères qui permettent de passer de la chaîne test à la chaîne de référence.

Notons $R = r_1.....r_m$ et $T = t_1.....t_n$ respectivement les chaînes de référence et de test, et $D(i,j)$ la distance entre $r_1.....r_i$ et $t_1.....t_j$. La distance de Levenshtein obéit aux équations récurrentes suivantes [WAG 76] :

$$D(i,j) = Min \begin{cases} D(i-1,j-1) + d(r_i, t_j) \\ D(i-1,j) + 1 \\ D(i,j-1) + 1 \ , \end{cases} \tag{1}$$

avec les conditions initiales :

$$\begin{array}{l} D(0,0) = 0 \\ D(i,0) = D(i-1,0) + 1 \quad 1 \leq i \leq m \\ D(0,j) = D(0,j-1) + 1 \quad 1 \leq j \leq n \ . \end{array} \tag{2}$$

La distance entre R et T est la valeur $D(m,n)$ calculée en résolvant cette récurrence. La figure 2 est une illustration du calcul de la distance de Levenshtein entre la chaîne test SISTAULI et la chaîne de référence SYSTOLIC. Les valeurs des distances $D(i,j)$ y sont indiquées, ainsi que le chemin qui correspond à une mise en correspondance optimale des deux chaînes.

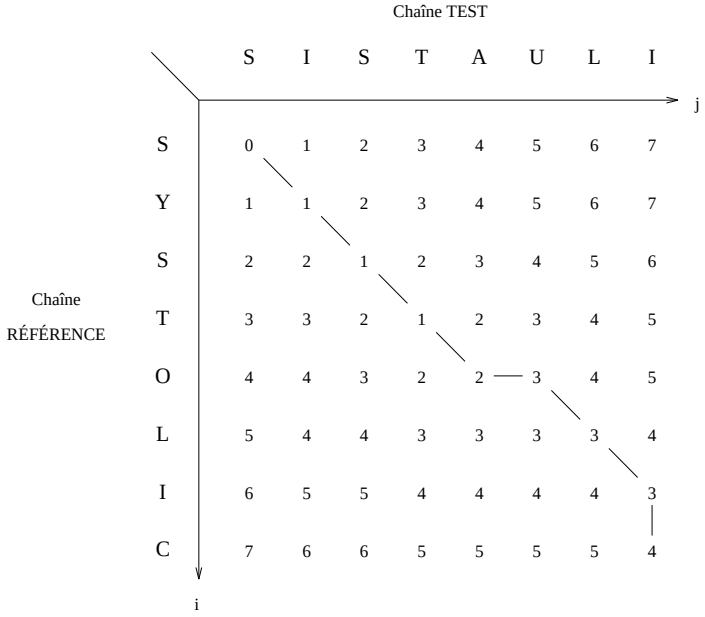

Figure 2. *Exemple du calcul de la distance de Levenshtein entre deux chaînes de caractères*

Dans une application typique, ce calcul doit être répété un grand nombre de fois puisqu'une chaîne erronée doit être comparée à toutes les références d'un dictionnaire.

2.2. *Mise en œuvre bi-dimensionnelle*

Cet algorithme possède les propriétés de régularité et de localité qui permettent de le systoliser. L'idée de base est d'associer un processeur au calcul de chaque distance $D(i,j)$, autrement dit, à chaque point de la figure 2. On obtient alors le réseau en grille de la figure 3. Ce réseau est composé de processeurs élémentaires interconnectés par l'intermédiaire de trois ports d'entrée et de trois ports de sortie.

A chaque étape du calcul, un processeur $P(i,j)$ reçoit respectivement les distances $D(i-1,j)$, $D(i-1,j-1)$ et $D(i,j-1)$ des processeurs $P(i-1,j)$, $P(i-1,j-1)$ et $P(i,j-1)$. Le caractère d'indice k appartenant à la chaîne test est diffusé verticalement aux processeurs $P(i,k)$, $(1 \leq i \leq m)$ et les caractères de la chaîne de référence sont diffusés horizontalement à travers le tableau. Les valeurs présentes sur les ports d'entrée des processeurs périphériques correspondent aux valeurs d'initialisation de l'équation [2]. Tous les processeurs effectuent simultanément le calcul correspondant à l'équation [1] avec les données qu'ils reçoivent de leurs voisins, le caractère de la chaîne de référence et le caractère de la chaîne test qu'ils possèdent. L'ensemble des opérations qui sont effectuées pour chaque pas de calcul constitue ce qu'on appelle un *cycle systolique*.

Comme une comparaison dure $n+m-1$ cycles systoliques et qu'un processeur y participe pendant un cycle seulement, chaque processeur est disponible le reste du temps. En fait, le calcul des valeurs $D(i,j)$ est effectué diagonale par diagonale. Au cours d'un cycle systolique t, seuls les processeurs $P(i,j)$ vérifiant la relation $t = i + j - 1$ sont actifs. A chaque cycle systolique, il est donc possible d'entamer la comparaison d'une nouvelle chaîne de référence avec la même chaîne test. On obtient ainsi un calcul pipeliné très efficace.

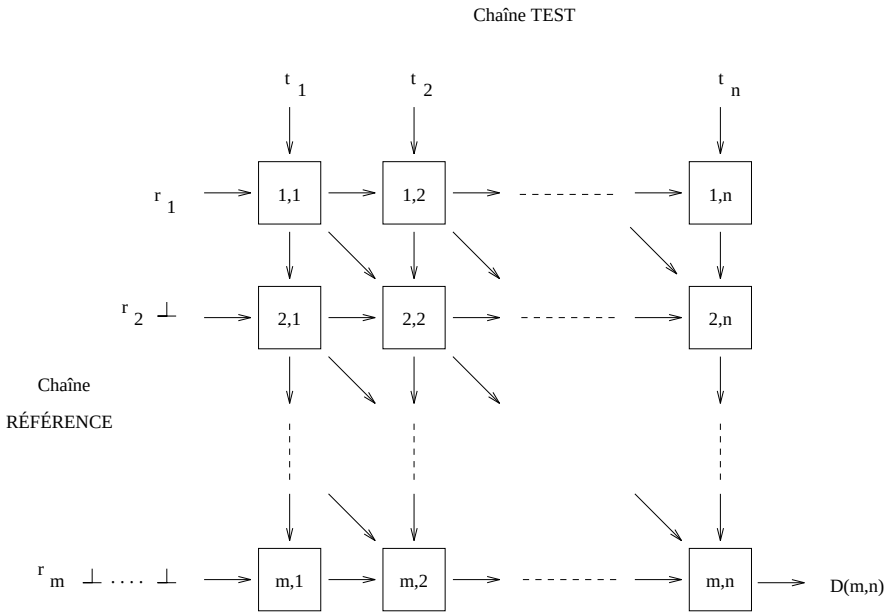

Figure 3. *Architecture en grille pour le calcul de la distance de Levenshtein*

2.3. *Programmation de cet algorithme sur un réseau linéaire*

Supposons que nous voulions simuler le fonctionnement d'une telle architecture sur une architecture parallèle. La solution architecturale bi-dimensionnelle que nous venons de décrire présente un inconvénient majeur. Pour être efficace, le réseau bi-dimensionnel doit être alimenté en données à chaque cycle de calcul : une nouvelle chaîne de référence doit être présentée aux m processeurs de la première colonne avant chaque calcul. Ceci exige de disposer d'un dispositif matériel suffisamment rapide pour permettre l'accès à m caractères en un cycle systolique.

Ce problème se pose concrètement si l'on cherche à exécuter en parallèle un tel algorithme sur des machines SIMD actuelles, telles que la MP-1 [NIC 90] ou la CM-2 [TUC 88]. Il est donc plus simple de simuler l'algorithme sur une architecture linéaire, ce qui n'est pas limitatif puisqu'il est toujours possible d'émuler une architecture bi-dimensionnelle en séquentialisant les calculs d'une colonne de processeurs sur un seul processeur du réseau linéaire.

L'architecture linéaire type pour simuler des réseaux systoliques a la structure donnée par la figure 4 qui représente la machine MicMacs [LAV 89] réalisée à l'IRISA. Elle se compose d'un réseau linéaire de processeurs élémentaires (module Mics) auquel est adjoint un mécanisme original de gestion des données (module Macs). Les propriétés de cette architecture et ses différences par rapport au modèle SIMD classique sont décrites dans[FRI 93]. C'est ce modèle architectural que l'on retrouve lorqu'on programme des machines parallèles destinées à l'accélération d'algorithmes numériques, telles que SPLASH [GOK 91], ArMen [POT 91] ou encore le réseau iWarp [BOR 88].

Sur une telle architecture, le calcul de la distance de Levenshtein est effectué comme indiqué par la figure 5. Chaque processeur émule le fonctionnement d'une colonne de l'architecture bidimensionnelle de la figure 3. De la sorte, le processeur le plus à gauche reçoit successivement les caractères r_1^k, r_2^k, ... de la chaîne de référence k, et ceux-ci circulent vers la droite à chaque cycle systolique. Les caractères de la chaîne test restent immobiles dans les processeurs du réseau. A chaque cycle systolique, un calcul de distance est effectué dans chaque processeur, et les transferts de données entre processeurs se déduisent d'une projection du fonctionnement du réseau bi-dimensionnel.

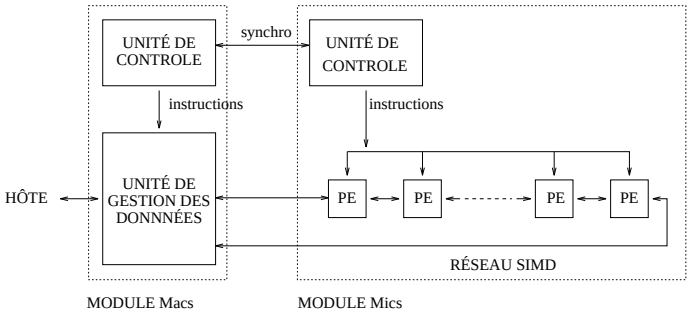

Figure 4. *La machine* MicMacs

D'une façon générale, pour programmer un algorithme systolique de façon efficace, il faut disposer d'un langage permettant de décrire et de compiler les calculs, les communications entre processeurs systoliques, les communications avec le processeur d'interface et le programme scalaire exécuté sur ce processeur. Nous examinons successivement ces aspects.

Comme une architecture systolique est *synchrone*, les processeurs exécutent conceptuellement les mêmes instructions : une même instruction du programme de calcul doit suffire à décrire le comportement de l'ensemble des processeurs de la machine de simulation.

Les communications entre processeurs systoliques ne peuvent pas être séparées du processus de calcul. Les opérandes d'une opération exécutée dans un processeur proviennent le plus souvent du résultat des calculs d'un voisin. C'est le cas dans notre exemple des distances partielles $d(i-1,j)$, $d(i-1,j-1)$. Les communications sont donc fréquentes et de faible granularité. Au niveau matériel, cela se traduit sur la machine de simulation par l'utilisation de dispositifs matériels dédiés afin de réduire les temps de synchronisation et de transfert entre processeurs [KUN 88]. L'expression des communications dans le langage de programmation doit permettre une compilation qui tire parti de ces dispositifs, faute de quoi la simulation sera totalement inefficace.

Les entrées-sorties entre le réseau systolique et son interface font aussi partie intégrante de l'algorithme. Dans notre exemple, les données échangées avec l'interface sont la chaîne de test, les chaînes de référence et les valeurs initiales. Pour satisfaire aux limitations de bande passante évoquées plus haut, les entrées-sorties sont réalisées uniquement aux frontières du réseau systolique. Les données sont introduites à une extrémité, puis progressent régulièrement à travers le réseau. De même les résultats produits par les calculs sur ces données transitent à leur propre rythme avant de sortir par une des cellules extrêmes. Entrées et sorties correspondent donc aux données reçues ou émises par les cellules extrêmes lors des opérations de communication. Pour simuler le réseau systolique, il faut programmer explicitement ces échanges avec l'interface ainsi que les synchronisations induites par ces échanges.

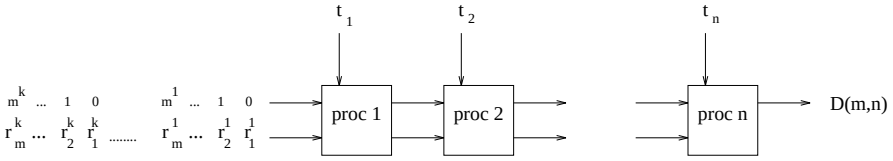

Figure 5. *Calcul de la distance de Levenshtein sur une architecture linéaire*

Une interface entre le circuit systolique et "le monde extérieur" (un PC, une mémoire, ou un autre circuit) est indispensable en calcul systolique. En effet le réseau systolique ne sait réaliser que des traitements simples et réguliers, et les processeurs ont une mémoire très limitée. Toutes les opérations de pré-traitement sur les données (comme la décompression, l'accès aléatoire à une mémoire, etc.) et les opérations de post-traitement sur les résultats (tri, seuillage, rétro-action, etc.) doivent être réalisées par l'interface. Des opérations de

contrôle communes à tous les processeurs peuvent y être avantageusement effectuées, de manière à minimiser le temps de calcul total. Pour l'exemple de la distance de Levenshtein, le test de début d'une nouvelle chaîne de référence est effectué sur l'interface. Le résultat de ce test est propagé avec chaque caractère de référence et les processeurs l'utilisent pour détecter les cas de réinitialisation partielle (équation 2). Sur la machine MicMacs par exemple (figure 4), cette fonction d'interfaçage est dévolue au module Macs qui dispose d'un processeur et d'une mémoire externe. La programmation de l'interface revêt une importance capitale dans le domaine systolique : les opérations qu'on y réalise ne doivent pas freiner les entrées-sorties et les calculs du réseau systolique; l'exploitation du parallélisme entre le processus de calcul et le processus de gestion des entrées-sorties permet de réduire le temps de simulation et de tester des transferts d'opérations du circuit systolique vers l'interface. Toutefois la présence de ce deuxième processus et la gestion des synchronisations nécessaires pour le contrôle global du système compliquent singulièrement la tâche du programmeur.

3. ReLaCS : un langage pour les Réseaux Linéaires de Cellules Systoliques

Le langage ReLaCS a été conçu pour programmer efficacement les algorithmes systoliques sur des architectures parallèles. Le modèle de programmation utilisé permet au programmeur de s'abstraire de la machine cible. Le parallélisme s'exprime par les données. Le contrôle est séquentiel et s'applique de façon globale à tous les processeurs. Des opérateurs d'affectation particuliers permettent de décrire explicitement les communications entre processeurs voisins et entre les processeurs extrêmes et l'extérieur du réseau. L'utilisateur écrit un seul programme en ReLaCS à partir duquel le compilateur génère à fois le code pour les processeurs du réseau et celui de l'interface d'entrées-sorties, en tenant compte du mode de fonctionnement de la machine ciblée (MIMD ou SIMD).

3.1. *Le modèle de programmation*

En première approximation le modèle de programmation que nous avons choisi s'apparente au modèle de programmation des machines SIMD. Le lecteur trouvera un exemple caractéristique d'architectures SIMD dans [HIL 86a]. Le programmeur perçoit la machine systolique comme un accélérateur de calcul connecté à une station hôte d'usage général (voir figure 6).

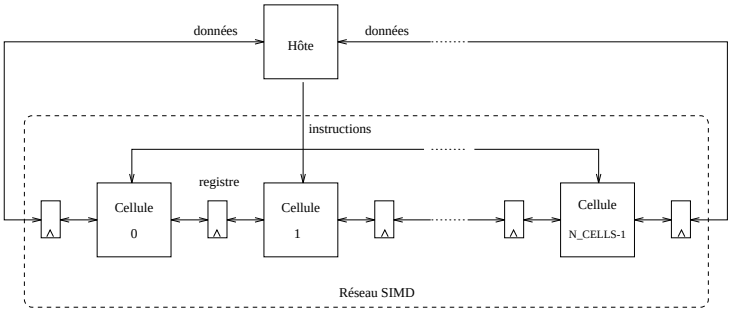

Figure 6. *Le modèle de programmation du langage* ReLaCS.

Cet accélérateur se compose d'un réseau de cellules (ou processeurs) identiques. Comme le nom du langage l'indique, nous limitons l'usage du langage ReLaCS aux réseaux de topologie linéaire. Le motif de ce choix résulte des constatations faites sur l'exemple introductif et notamment de l'impossibilité de gérer efficacement les entrées-sorties dans le cas des réseaux bi-dimensionnels. Le nombre de cellules est fixé à l'exécution et référencé par la valeur `N_CELLS`.

Chaque processeur dispose, outre des caractéristiques architecturales d'un processeur conventionnel, de deux ports de communication qui lui permettent d'échanger des données avec ses voisins immédiats par

l'intermédiaire d'un registre partagé. Seuls les deux processeurs situés aux extrémités du réseau sont connectés à l'hôte. L'hypothèse fondamentale du modèle de programmation SIMD réside dans le fonctionnement strictement synchrone des processeurs : l'hôte diffuse la même instruction à tous les processeurs qui l'exécutent simultanément avec leurs données locales comme opérandes.

La programmation d'un algorithme systolique sur ce modèle architectural consiste à décrire deux processus :

- Le *processus de calcul* réalise la tâche de calcul de manière systolique, i.e. en utilisant les valeurs d'entrées-sorties comme opérandes. Ce processus s'exécute sur le réseau linéaire de processeurs systoliques. Ces processeurs sont séquencés par un contrôleur unique qui diffuse la même instruction à chacun d'entre eux.

  Dans le modèle SIMD classique, il existe un mécanisme contrôlant l'activité des processeurs du réseau. Chaque processeur possède un bit d'activité qui indique si l'instruction reçue doit être exécutée. Ce bit est positionné localement lors d'opérations conditionnelles ou de masquage. En retour, un OU logique du bit d'activité de tous les processeurs permet au contrôleur de tester la présence d'un processeur actif et de déterminer s'il y a lieu de continuer une itération. Nous n'avons pas inclus pour l'instant ce mécanisme dans le modèle de programmation de ReLaCS. En revanche nous faisons l'hypothèse de disposer d'une instruction d'affectation conditionnelle dans le jeu d'instruction du processeur du réseau. Cette instruction réalise le multiplexage de deux registres sources vers un registre destination en fonction d'une condition du registre d'état. Ce type d'instruction est disponible dans certains processeurs de machines SIMD, comme la machine Blitzen [BLE 90] ou encore MicMacs [FRI 90], et plus récemment dans le processeur Alpha [COR 92].

  Une autre différence par rapport au modèle de programmation SIMD général est l'absence d'un routeur général de message. Dans notre modèle, la vitesse régulière des flôts de données et la localité des calculs systoliques se traduisent par des communications synchrones entre cellules voisines. Nous supposons que le jeu d'instruction du processeur ne contient que les instructions nécessaires à l'écriture et à la lecture des registres qu'il partage avec ses voisins.

- Le *processus de gestion des données* est responsable de l'envoi des données au réseau systolique et de la récupération des résultats. Ce processus se déroule sur un processeur indépendant. Il communique directement avec les cellules situées aux extrémités du réseau.

Les deux processus se synchronisent au moment des communications et des ruptures dans le flux de contrôle (les instructions conditionnelles). La machine sur laquelle s'exécutent les deux processus s'appelle une machine systolique. Même si le *processus de gestion des données* est physiquement localisé sur la machine systolique, du point de vue du programmeur, tout se passe comme s'il s'exécutait sur l'hôte. Le programmeur partage son application entre un ensemble de fonctions de calculs et une partie principale. L'hôte exécute la partie principale et accède par appels de fonctions à l'accélérateur des calculs systoliques.

Il convient de faire deux remarques importantes sur le choix du modèle synchrone :

- Comme le champ d'application du langage ReLaCS est volontairement limité aux calculs réguliers, le modèle synchrone n'apporte aucune contrainte supplémentaire sur la programmation d'algorithmes systoliques. Au contraire cette hypothèse simplifie le travail du programmeur en lui évitant la gestion des synchronisations entre processeurs du réseau.

- Le modèle synchrone ne restreint pas l'exécution des programmes ReLaCS aux seules machines SIMD. Le modèle doit être percu par le programmeur comme une méthode de programmation indépendante de la machine d'exécution. C'est au compilateur de réaliser ensuite les optimisations adaptées à l'architecture ciblée. Il peut notamment relâcher les contraintes de synchronisme dans le cas des machines MIMD.

3.2. *Structure de données*

Le langage ReLaCS est un langage impératif à structure de blocs proche du langage C [KER 78]. Le parallélisme est explicite et repose sur une extension sémantique des structures de données du langage C.

Le programmeur de la machine précédemment décrite a besoin d'un moyen pour différencier les variables de l'hôte (variables scalaires) de celles localisées sur le réseau systolique (variables systoliques). Nous avons défini une nouvelle classe de mémorisation, la classe `systolic`, réservée à l'allocation de variables dans chaque cellule du réseau ; l'autre classe, la classe `static`, spécifie l'allocation de variables scalaires sur l'hôte. Une instruction opérant sur des variables systoliques produit une exécution simultanée de cette opération sur toutes les cellules. Une instruction manipulant des variables scalaires ne se déroule que sur l'hôte. Ce modèle de programmation parallèle est référencé sous le terme de *parallélisme par les données* [HIL 86b]. Les variables impliquées dans une expression doivent être de même classe. L'échange de valeurs entre variables systoliques et variables scalaires s'exprime explicitement au moyen d'opérateurs de communication décrits dans le paragraphe 3.4. La notion de classe, telle qu'elle vient d'être définie, est orthogonale à la notion de type usuelle. Les types de base du langageReLaCS sont les entiers (`int`), les réels (`float`), les caractères (`char`) ; actuellement le seul construteur de type autorisé est le tableau (`[ ]`).

L'exemple représenté figure 7 illustre l'effet de l'affectation entre deux variables systoliques entières par `x= y`.

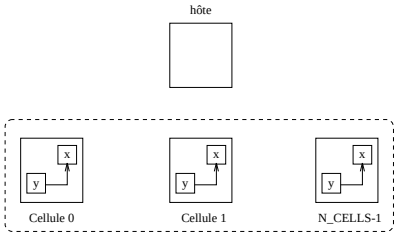

Figure 7. *Affectation entre variables systoliques.*

3.3. *Le flôt de contrôle*

Le contrôle est séquentiel et s'exprime au moyen des structures conditionnelles et itératives classiques. Dans le langage ReLaCS, comme dans le langage C, le programmeur dispose des instructions `while()`, `for()`, `if()`... Cependant la sémantique de ces instructions dans le langage ReLaCS doit tenir compte du contexte synchrone de la machine d'exécution. En effet, le fonctionnement SIMD qui impose à tous les processeurs de recevoir la même instruction, complique le traitement des sauts conditionnels.

Examinons le cas d'une instruction conditionnelle `if-then-else` dont le résultat booléen détermine les instructions à envoyer aux processeurs du réseau. Deux cas peuvent se présenter selon la classe de mémorisation associée à la condition.

- La condition est de classe `static`. Elle est évaluée sur l'hôte et tous les processeurs recoivent les instructions qui correspondent à la branche de code sélectionnée par le résultat de la condition.
- La condition est de classe `systolic`. Dans ce cas il se peut que tous les processeurs n'aient pas la même branche de code à exécuter. Compte tenu du modèle de programmation choisi (cf. paragraphe 3.1), le compilateur ReLaCS ne peut traiter ce cas et détecte une erreur de classe lors de l'analyse sémantique du programme.

Les langages SIMD classiques qui supposent la présence d'un masque d'activité, l'utilisent pour envoyer systématiquement les instructions des deux branches de l'instruction conditionnelle. En fonction de la valeur

de leur bit d'activité positionné lors de l'évaluation de la condition, les processeurs exécutent les instructions d'une des branches et restent inactifs pendant l'autre. L'imbrication de plusieurs instructions conditionnelles soulève le problème de l'empilement des contextes et de leurs restaurations. Des solutions efficaces à base de compteurs viennent d'y être apportées et validées [LEV 93]. Cependant nous avons choisi de laisser de côté ces problèmes de compilation et de ne supporter qu'un modèle de programmation SIMD simplifié. Notre expérience en matière de programmation systolique nous a confortés dans ce choix et les algorithmes que nous avons étudiés ne nécessitent pas d'introduire le concept de masque d'activité, d'autant que l'écriture d'instructions conditionnelles locales conduit à séquentialiser l'exécution des deux branches de code et par voie de conséquence, à réduire l'efficacité du réseau de processeurs. Par contre il nous est apparu utile de disposer d'un contrôle local plus limité, par exemple pour réaliser un calcul de maximum dans chacun des processeurs.

L'instruction d'affectation conditionnelle introduite dans notre modèle de programmation pourvoit à cette tâche. Son exploitation par le langage ReLaCS repose sur une sémantique différente des expressions conditionnelles du langage C. Dans sa forme syntaxique de base, l'expression conditionnelle s'écrit `c ? a:b` et prend pour valeur `a` si `c` est vrai, `b` sinon. En ReLaCS c'est le seul cas où la condition, ici `c` , peut être de classe `systolic`. Par exemple le calcul d'un maximum dans chaque cellule systolique se programme ici par l'instruction `z= x>y ? x:y`. En C, cette expression a la même signification que l'instruction conditionnelle `if(x>y) z= x; else z= y` et le compilateur la traduit comme telle. En ReLaCS, le compilateur transforme l'expression conditionnelle de base en une opération d'affectation conditionnelle. Cependant dans sa forme générale où chaque membre de la conditionnelle peut être une expression complexe, voire un appel de fonction avec des effets de bord, il n'est plus possible de conserver la même sémantique qu'en C. Le compilateur du langage ReLaCS génère successivement le calcul de chaque membre de la conditionnelle, le calcul de la condition et enfin l'opération d'affectation conditionnelle. Dans tous les cas les deux branches de la conditionnelle sont évaluées.

3.4. *Les communications*

Le modèle de programmation suppose un mode de fonctionnement SIMD et des communications synchrones entre processeurs voisins. L'utilisation correcte de ce modèle impose que toute émission d'une donnée par un processeur dans une direction soit suivie de la réception de la donnée émise par le processeur voisin situé dans la direction opposée. Cette séquence d'opérations est synthétisée dans des opérateurs d'affectation portant sur des variables de classe `systolic`:

- Opérateur d'affectation à gauche, exemple : `x =< y`. Chaque processeur émet une valeur vers la gauche et en reçoit une de la droite (figure 8a). Le processeur situé le plus à droite du réseau ne modifie pas le contenu de sa variable `x`.
- Opérateur d'affectation à droite, exemple : `x => y`. Chaque processeur émet une valeur vers la droite et en reçoit une de la gauche (figure 8b). Le processeur situé le plus à gauche du réseau ne modifie pas le contenu de sa variable `x`.

L'effet global de ces opérateurs est de réaliser un décalage des variables du réseau qui reflète les flôts de données caractéristiques des algorithmes systoliques. Dès lors il est naturel d'étendre la portée de ces opérateurs aux entrées-sorties du réseau en y ajoutant la valeur à envoyer au premier processeur et la variable de l'hôte destinée à garder la valeur émise par le dernier processeur. Cette variable et cette valeur initiale sont de classe `static` et apparaissent de manière optionnelle dans les opérateurs précedemment décrits :

- Dans l'opérateur d'affectation à gauche, exemple : `x:A =< y:B`. Le processeur situé le plus à droite du réseau reçoit la valeur de `B` émise par l'hôte. Celui situé le plus à gauche envoie sa valeur de `y` à l'hôte qui la range dans `A` (figure 9a).

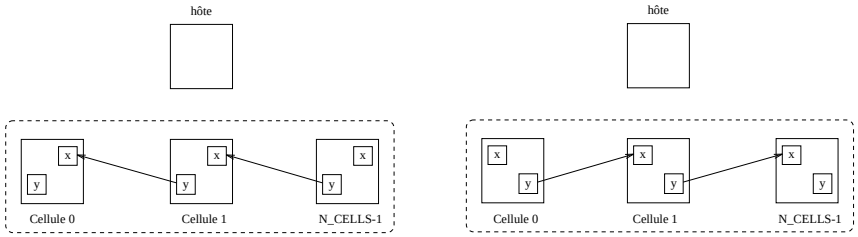

Figure 8. *Les communications internes au réseau: (a) vers la gauche, (b) vers la droite.*

- Dans l'opérateur d'affectation à droite, exemple : `x:A => y:B`. Le processeur situé le plus à gauche du réseau reçoit la valeur de `B` émise par l'hôte. Celui situé le plus à droite envoie sa valeur de `y` à l'hôte qui la range dans `A` (figure 9b).

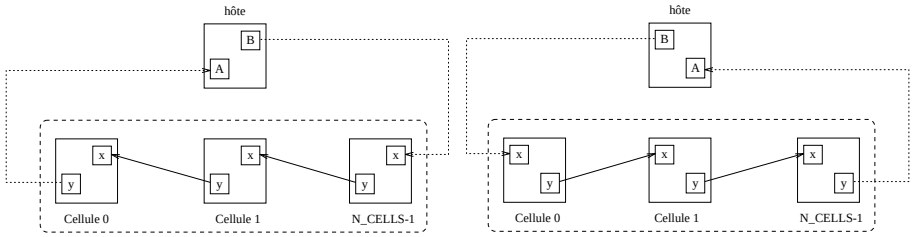

Figure 9. *Les communications étendues à l'hôte :(a) vers la gauche, (b) vers la droite.*

La seconde possibilité de communication entre l'hôte et le réseau consiste à diffuser la même valeur à tous les processeurs. Cette communication est explicite et s'exprime à l'aide d'un troisième opérateur d'affectation :

- l'opérateur d'affectation globale, exemple : `x =| B`. Tous les processeurs reçoivent la valeur de `B` émise par l'hôte et la rangent dans `x` (figure 10).

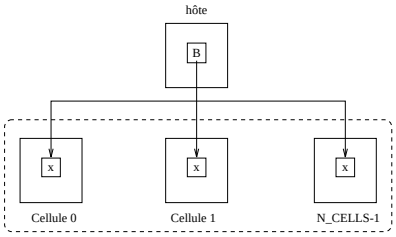

Figure 10. *L'opération de diffusion.*

Cette opération de diffusion n'est pas réellement une opération systolique dans la mesure où elle n'a pas un caractère local. Cependant elle possède trois avantages :

- Elle ne nécessite pas de dispositif matériel particulier pour être supportée par le modèle de programmation SIMD : la valeur à diffuser est véhiculée par le bus d'instructions.
- Elle permet de remplacer une séquence d'instructions de communication propagant la valeur par une seule opération. Ceci soulage l'écriture des programmes.

- Elle accélère l'exécution des programmes systoliques sur des machines SIMD.

L'introduction de ces nouveaux opérateurs libère le programmeur de la gestion des synchronisations au moment des communications. Processus d'alimentation et processus de calcul lui apparaissent comme séquentiels.

3.4.1. *Exemple*

A titre d'exemple nous reprenons l'algorithme du calcul de la distance de Levenshtein introduit dans le paragraphe 2. La mise en œuvre proposée consiste à charger un caractère de la chaîne de test par cellule puis à faire circuler l'ensemble des chaînes de référence. Nous donnons en figure 11 les parties essentielles de la boucle de base (après initialisation et chargement de la chaîne test dans le réseau) réalisant le calcul systolique des distances de Levenshtein entre la chaîne test et chacune des chaînes références. Les commentaires apparaissent entre les symboles délimiteurs `/*` et `*/` ; les parties du programmes non détaillées figurent entre `< >`.

```
<... initialisations ...>
  while ( J < B[M]+N_CELLS ){ /* parcourt du dictionnaire de reference */
    d_s= d_a; /* emulation du transfert diagonal d(i-1,j-1) -> d(i,j) */
    d_o= d; /* emulation du transfert vertical d(i,j-1) -> d(i,j) */
    <...initialisation de D0_a par d(i,0)...>
    if ( J-N_CELLS+1 == B[K+1] ) /* test de fin de calcul */
       d_a: D[K++] => d: D0_a; /* transfert horizontal, sauvegarde du resultat */
     else
       d_a => d: D0_a; /* transfert horizontal sans sauvegarde du resultat */
    <...transfert du symbol de reference...>
    <...calcul des couts d'edition...>
    d= MIN(c_s,c_a,c_o); /* calcul de l'equation */
    <...incrementation des indices...>
   }
```

Figure 11. *Programmation du calcul de la distance de Levenshtein.*

Les structures de données utilisées sur l'hôte et figurant dans le programme de la figure 11 : un tableau de caractères contenant l'ensemble des `M` mots du dictionnaire, un tableau `B` d'entiers indiquant le début de chaque mot dans le dictionnaire, un tableau `D` d'entiers des distances calculées, un indice `J` de parcours du tableau `B`, un indice `K` de parcours du tableau `D`, une variable entière `D0_a` contenant la valeur d'initialisation $d(0, j)$.

Les structures de données déclarées dans les processeurs du réseau se limitent : aux variables entières `d_s`, `d_a`, `d_o`, `d`, contenant respectivement les distances partielles $d(i-1, j-1)$, $d(i-1, j)$, $d(i, j-1)$, $d(i, j)$ et aux variables temporaires `c_s`, `c_a`, `c_o` utilisées pour le calcul des coûts, respectivement de substitution, d'addition et d'omission.

Le calcul des distances est programmé de la manière suivante (on ne commente ici que les instructions exprimées dans le programme de la figure 11) :

- Une boucle englobante réalise le parcours des caractères de toute les chaînes de référence (dictionnaire). Elle se termine quand l'indice `J` des caractères de référence dépasse la longueur du dictionnaire augmentée de la taille du réseau (`N_CELLS`). Cette majoration tient compte de la latence du réseau et a pour objectif d'effectuer la phase de vidage.

- Les deux premières affectations opèrent sur des variables systoliques. Elles émulent les connections diagonales et verticales d'un réseau en grille. On rappelle que chaque processeur du réseau linéaire réalise les calculs d'une colonne de processeur d'un réseau bi-dimensionnel.
- Le test suivant détermine si le calcul réalisé par le processeur le plus à droite termine le calcul d'une distance. Si oui, le résultat de la propagation des distances $d(i-1,j)$ est mémorisé sur l'hôte. Sinon on réalise tout de même la propagation des distances $d(i-1,j)$ mais on ne conserve pas le résultat sur l'hôte.
- Le minimum exprime le calcul partiel des distances sur chaque processeur en fonction des coûts calculés localement.

3.5. *Compilation de* ReLaCS

La phase d'analyse d'un programme ReLaCS utilise les techniques éprouvées en matière de compilation [AHO 86] : analyse lexicale et syntaxique dirigée par la syntaxe, héritage et synthèse d'attributs dans l'arbre décoré de la représentation abstraite. Au cours de cette phase sont rejetés tous les programmes ne respectant pas la syntaxe du langage ou les règles liées au typage et aux classes de mémorisation.

La phase de synthèse du compilateur produit des programmes sources C. Elle suppose l'existence d'un compilateur C et une librairie de routines de communications sur la machine ciblée. Cela simplifie le développement du compilateur et en assure le portage sur la plupart des machines actuelles. Le compilateur produit deux programmes C pour une machine cible parallèle ou un seul programme C pour une machine cible séquentielle.

3.5.1. *La compilation pour machines parallèles*

La compilation d'un programme ReLaCS pour une architecture parallèle consiste :

- à séparer les opérations de gestion des entrées-sorties (processus d'alimentation en données du réseau) de celles de calcul des processeurs (processus de calcul systolique),

```
<initialisations>
  while (set_flag(J < B[M]+N_CELLS)){
    <initialisation de D0_a par d(i,0)>
    if (set_flag(J-N_CELLS+1==B[K+1])){
      snd_right(D0_a);
      rcv_left(&D[K++]);
    }
    else
      snd_right(D0_a);
    <transfert du symbole de reference>
    <incrementation des indices>
  }
```

```
while (is_flag()){
  d_s= d_a;
  d_o= d;
  if (is_flag())
    shift_right(d,&d_a);
  else
    shift_right2(d,&d_a);
  <transfert du symbole>
  <calcul des couts>
  d= MIN(c_s,c_a,c_o);
}
```

Figure 12. *Programmes de l'algorithme de Levenshtein générés pour les machines* SIMD *: (a) programme d'alimentation, (b) programme de calcul.*

```
<initialisations>
  while (J<B[M]+N_CELLS){

    <initialisation de D0_a>
    if (J-N_CELLS+1==B[K+1]){
      snd_right(D0_a);
      rcv_left(&D[K++]);
    }
    else
      snd_right(D0_a);
     <transfert du symbole>

    <incrementation des indices>
  }
```

```
<initialisations>
  while (J<B[M]+N_CELLS){
    d_s= d_a;
    d_o= d;

    if (J-N_CELLS+1==B[K+1])
      shift_right(d,&d_a);

    else
      shift_right2(d,&d_a);
    <transfert du symbole>
    <calcul des couts>
    d= MIN(c_s,c_a,c_o);
   <incrementation des indices>
  }
```

Figure 13. *Programmes de l'algorithme de Levenshtein générés pour les machines* MIMD *: (a) programme d'alimentation, (b) programme de calcul.*

- à produire un programme C pour chaque processus et à générer les opérations de communication et de synchronisation nécessaires.
- à compiler indépendamment chaque programme C avec les compilateurs et les librairies de communication de la machine cible.

Par exemple, la compilation du programme de calcul de la distance de Levenshtein représenté en figure 11 produit les deux programmes de la figure 12.

Le programme de gauche contient les opérations à réaliser par le processus d'alimentation en données ; le programme de droite, celles du processus de calcul. On retrouve les instructions du programmes original, plus des fonctions de contrôle et de communication. Le processus d'alimentation gère les instructions conditionnelles et communique les résultats des tests au processus de calcul par l'intermédiaire des fonctions `set_flag()` et `is_flag()`. Les opérateurs d'affectations à droite du programme ReLaCS sont traduits en opérations de communication élémentaires dans le programme de calcul (`shift_right()` ou `shift_right2()`) et dans le programme d'alimentation quand il envoie une donnée (`snd_right`) ou reçoit un résultat (`rcv_left`). La différence entre `shift_right()` et `shift_right2()` tient au fait que dans le premier cas, le processus le plus à droite émet une valeur et que dans l'autre cas il n'en émet pas.

Comme nous l'avons indiqué dans le paragraphe 3.1 il est possible de compiler ReLaCS pour une architecture MIMD. Dans ce cas, le code du programme d'alimentation en données est chargé sur un processeur, le code du programme de calcul sur les autres processeurs disponibles et un réseau virtuel en forme d'anneau est établi entre tous les processeurs. L'équivalence entre l'exécution asynchrone sur une machine MIMD et l'exécution synchrone sur une machine SIMD est garantie par la présence des points de synchronisation que constituent les communications et la diffusion des conditions. Cependant il n'existe pas de mécanisme matériel sur les machines MIMD supportant efficacement la diffusion d'une valeur d'un processeur vers tous les autres. Par contre chaque processeur d'une machine MIMD possède un séquenceur et est à même de calculer les conditions dans la mesure où elles ne dépendent pas d'opérations de communication. Reprenant la technique utilisée dans [HAT 91], la compilation des programmes ReLaCS pour les machines MIMD est optimisée en dupliquant les opérations intervenant dans le contrôle à l'intérieur du programme de calcul.

Pour illuster cette technique d'optimisation, nous donnons en figure 13 les programmes générés lors de la compilation de l'algorithme de Levenshtein sur une machine MIMD. Par rapport aux programmes de la

figure 12, les fonctions `set_flag()` et `is_flag()` ont disparu des deux programmes, les tests de condition et les opérations d'incrémentation sont rajoutées dans le programme du processus de calcul.

3.5.2. *La compilation pour machines séquentielles*

La compilation de programmes parallèles pour des machines séquentielles est essentielle pour la mise au point des algorithmes. Elle permet de réaliser une exécution déterministe et de disposer d'un environnement d'observation de l'état global de la machine. La compilation d'un programme ReLaCS sur une machine séquentielle produit un programme C.

Ce programme C est obtenu en indexant les variables systoliques et en séquentialisant toutes les opérations qui s'effectuent de manière parallèle sur le réseau de processeurs. L'indexation d'une variable systolique consiste à la transformer en un tableau de taille `N_CELLS` (ou à lui rajouter une dimension supplémentaire s'il s'agit déjà d'un tableau). Toute opération opérant sur des variables systoliques est enroulée dans une boucle itérative parcourant l'indice des processeurs. La séquentialisation d'une opération de communication décompose l'échange de données en `N_CELLS` décalages successifs. En guise d'exemple, nous donnons en figure 14 un extrait du programme séquentiel généré par le compilateur ReLaCS pour l'algorithme de Levenshtein. Seuls y figurent le test de fin de calcul d'une distance, et le calcul de la minimisation.

```
< ... >
    if ( J-N_CELLS+1 == B[K+1] ){
      D[K++]= d[N_CELLS];
      for (I_CELL= 1; I_CELL < N_CELLS; I_CELL++)
        d_a[(N_CELLS+1)-I_CELL]= d[(N_CELLS+1)-(I_CELL+1)];
      d_a[1]= D0_a;
    }
    else{
      for (I_CELL= 1; I_CELL < N_CELLS; I_CELL++)
        d_a[(N_CELLS+1)-I_CELL]= d[(N_CELLS+1)-(I_CELL+1)];
      d_a[1]= D0_a;
    }
    < ... >
    for (I_CELL= 1; I_CELL < N_CELLS+1; I_CELL++)
      d[I_CELL]= MIN(c_s[I_CELL],c_o[I_CELL],c_a[I_CELL]);
< ... >
```

Figure 14. *Programme de l'algorithme de Levenshtein généré pour les machines séquentielles.*

4. Conclusion

Nous avons présenté le langage ReLaCS pour la programmation des algorithmes systoliques sur des architectures parallèles. Ce langage utilise le concept de parallélisme de données afin de faciliter la programmation des calculs réguliers. L'introduction de nouveaux opérateurs permet d'exprimer les communications systoliques et les entrées-sorties comme de simples affectations. Le compilateur réalise la traduction du programme source ReLaCS en deux programmes C concurrents, le programme du calcul systolique et le programme de gestion des entrées-sorties. La méthodologie employée pour la compilation autorise le portage de ReLaCS sur plusieurs machines parallèles, SIMD ou MIMD. L'environnement de programmation est complété par un schéma de compilation pour une exécution séquentielle pour la mise au point. Nous avons montré sur un exemple typique, l'algorithme de Levenshtein, la programmation d'un calcul systolique et les résultats successifs de sa compilation en ReLaCS pour des machines de type SIMD, MIMD et séquentiel.

L'état actuel des développements autour de ReLaCS comprend une version opérationnelle du compilateur sur les machines iPSC/2, ArMen et iWarp. A défaut d'être efficace, le portage de ReLaCS sur la machine à mémoire distribuée iPSC/2 nous a permis d'expérimenter le processus de compilation pour les architectures MIMD. Par la suite nous avons proposé un mécanisme matériel de communication plus efficace. Ce mécanisme a été testé et validé sur la couche de logique reprogrammable de la machine ArMen [RAI 93]. Le portage plus récent sur iWarp, la seule machine systolique commercialisée à ce jour, et les performances obtenues sur un problème de programmation linéaire [AND 93] prouvent l'efficacité du modèle de programmation ReLaCS en matière de vitesse et d'accélération. Parmi les autres applications qui ont été développées en ReLaCS nous pouvons citer le calcul matriciel, le traitement vidéo numérique et le traitement des chaînes de caractères.

Les travaux de recherche en cours visent le développement d'une interface graphique pour faciliter la mise au point des algorithmes systoliques, la génération et l'optimisation de code pour les processeurs d'une machine spécialisée dans le traitement des séquences du génome [LAV 92].

5. Bibliographie

[AHO 86] AHO A., SETHI R., et ULLMAN J. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, , 1986.

[AIR 90] AIRIAU R., BERGE J.M., OLIVE V., et ROUILLARD J. *VHDL : du langage à la modélisation.* Presses Polytechniques et Universitaires Romandes, , 1990.

[AND 93] ANDONOV R., RAIMBAULT F., et QUINTON P. "Dynamic Programming Parallel Implementation for the Knapsack Problem". Internal Report 740, IRISA, , Juillet 1993. Submitted to JPDC. Available by anonymous ftp on irisa.irisa.fr.

[ARN 85] ARNOULD E., KUNG H.T., MENZILCIOGLU O., et SAROCKY K. "A Systolic Array Computer". *Proc. IEEE Int. Conf. ICASSP 85*, p. 232–235, Tampa, FL, U.S.A., , 1985.

[BLE 90] BLEWINS D., DAVIS E., HEATON R., et REIF J. "BLITZEN : A Highly Integrated Massively Parallel Machine". *Journal of Parallel and Distributed Computing*, vol. 8 n° 2, Février 1990, p. 150–160.

[BOR 88] BORKAR S., COHN R., COX G., GLEASON S., GROSS T., KUNG H.T., LAM M., MOORE B., PETERSON C., PIEPER J., RANKIN L., TSENG P., SUTTON J., URBANSKI J., et WEBB J. "iWarp :An integrated Solution to High-Speed Parallel Computing". *International Conference on Supercomputing*, , 1988.

[CHR 90] CHRISTY P. "Software to Support Massively Parallel Computing on the Maspar MP-1". *COMPCON SPRING*, IEEE, éditeur, , Février 1990.

[COR 92] CORPORATION Digital Equipement. "Alpha Architecture Handbook". Digital Equipement Corporation, , 1992.

[FRI 90] FRISON P., GAUTRIN E., LAVENIER D., et SCHARBARG J.L. "Designing specific systolic array with the api15c chip". *ASAP 90*, , Septembre 1990.

[FRI 93] FRISON P., LAVENIER D., et RAIMBAULT F. "I/O Data Management on SIMD Systolic Arrays". *International Conference on Application Specific Array Processors*, , Octobre 1993.

[GOK 91] GOKHALE M., HOLMES W., KOPSER A., LUCAS S., MINNICH R., SWEELY D., et LOPRESTI D. "Building and Using a Highly Parallel Programmable Logic Array". *Computer*, vol. , Janvier 1991.

[HAM 88] HAMEY L., WEBB J., et WU I. *Low-level vision on Warp and the APPLY programming model*, chapter 10, p. 185–199. Kluwer Academic Publishers, , 1988.

[HAT 91] HATCHER P. et QUINN M. *Data-Parallel Programming on MIMD Computers.* MIT Press, , 1991.

[HIL 86a] HILLIS D. *The Connection Machine.* MIT Press, , 1986.

[HIL 86b] HILLIS W. et STEELE G. "Data Parallel Algorithms". *Communications of the ACM*, vol. 29 n° 12, Décembre 1986, p. 1170–1183.

[HUG 92] HUGHEY R. "Programming Systolic Arrays". *International Conference on Application Specific Array Processors*, , 1992.

[KER 78] KERNIGHAN, BRIAN W., et RITCHIE D. M. *The C programming language.* Prentice-Hall, , 1978.

[KUN 80] KUNG H.T. et LEISERSON C.E. "Algorithm for VLSI Processor Arrays". *Introduction to VLSI systems*, chapter 8.3. Addison-Wesley, , 1980.

[KUN 88] KUNG H.T. "Systolic Communication". *International Symposium on Computer Architecture*, p. 695–703, , Mai 1988.

[LAM 89] LAM M. *A Systolic Array Optimizing Compiler.* Kluwer Academic Publisher, , 1989.

[LAV 89] LAVENIER Dominique. "MicMacs : un réseau systolique linéaire programmable pour le traitement des chaines de caractères". Thèse de l'Université de Rennes 1, , Juin 1989.

[LAV 92] LAVENIER Dominique. "Architectures systoliques et traitements des séquences biologiques". LIRMM - Montpellier, , Novembre 1992.

[LAW 75] LAWRIE D. H., LAYMAN T., BAER D., et RANDAL J. M. "Glypnir - A Programming Language for Illiac IV". *Communications of the ACM*, vol. 18 n° 3, Mars 1975, p. 157–164.

[LEV 93] LEVAIRE J.L. "Contribution à l'étude sémantique des langages à parallélisme de données; application à la compilation". Thèse de l'Université de Paris 7, , Février 1993.

[MEA 80] MEAD C.A. et CONWAY L.A. *Introduction to VLSI system*. Addison-Wesley, , 1980.

[NIC 90] NICKOLLS J. R. "The Design of the MasPar MP-1 : A Cost Effective Massively Parallel Computer". *COMPCON SPRING*, IEEE, éditeur, p. 25–28, , Février 1990.

[Occ 88] *Occam 2, Reference Manual*, , 1988.

[O'H 91] O'HALLARON D. "The ASSIGN Parallel Program Generator". Technical report cmu-cs-91-141, Carnelie Mellon University, , Mai 1991.

[PAR 92] PARIS N. *Définition de POMPC (version1.99)*. Laboratoire d'Informatique, Ecole Normale Supérieure, , Mars 1992.

[PER 85] PERROTT R., CROOKES D., MILLIGAN P., et PURDY M. "A Compiler for an Array and Vector Processing Language". *IEEE Transactions on Software Engineering*, vol. SE-11 n° 5, Mai 1985, p. 471–478.

[POT 91] POTTIER B. "Armen, une machine parallèle intégrant un réseau de circuits reconfigurables". Thèse de l'université de Rennes I, , Juin 1991.

[QUI 89] QUINTON P. et ROBERT Y. *Algorithmes et architectures systoliques*. Masson, , 1989.

[RAI 93] RAIMBAULT F., LAVENIER D., RUBINI S., et POTTIER B. "Fine grain parallelism on a MIMD machine using FPGAs". *IEEE Workshop on FPGAs for Custom Computing Machines*, p. 2–8, Napa, California, , Avril 1993. Available by anonymous ftp on irisa.irisa.fr under internal report number 728.

[TSE 90] TSENG P. S. "A systolic Array Programming Language". *International Conference on Application Specific Array Processors*, , Septembre 1990.

[TUC 88] TUCKER L. et ROBERTSON G. "Architecture and Applications of the Connection Machine". *Computer*, vol. 21 n° 8, Août 1988, p. 26–38.

[TUC 90] TUCK R. *Porta-SIMD : An Optimally Portable SIMD Programming Language*. PhD thesis, University of North Carolina, , Mai 1990.

[WAG 76] WAGNER R. A. et FISHER M. J. "The string to string correction problem". *Communications of the ACM*, vol. 21, 1976, p. 168–173.

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