Université PARIS XI
Faculté d'Orsay
Laboratoire de Recherche en Informatique

Virtualisation et Programmation Physique dans le Langage POMPC : Application à l'Opération Préfixe

Rapport de Stage de DEA
effectué au :

Laboratoire d'Informatique
Ecole Normale Supérieure

45, Rue d'Ulm
75230 PARIS Cedex 05

présenté par :
Boukhalfa HADIM

dirigé par :
M. Nicolas PARIS

soutenue le : 16.09.92

Remerciements

Je tiens à remercier Nicolas Paris, mon directeur de stage et Ronan Keryell, thésard dans l'équipe architecture du LIENS. J'ai appris énormément de choses au contact de ces deux personnes qui n'ont été à aucun moment avares en directives constructives. La masse d'informations que j'ai pu acquérir grâce à Nicolas et Ronan, tout au long de ce stage, est immense. Qu'ils trouvent ici l'expression de ma gratitude la plus sincère. Je remercie aussi toutes les personnes du LIENS qui ont bien voulu répondre à mes questions de débutant en toute gentillesse.

Je tiens à exprimer mon entière reconnaissance à Mr J.F.Myoupo, qui a bien voulu accepter mon encadrement du coté du LRI et qui m'a orienté tout au long de ce stage tout au long des discussions que j'ai eu avec lui.

Enfin je remercie tous ceux qui de loin ou de prés m'ont aidé dans mon travail.

Chapitre 1

Introduction

Dès l'avènement de l'ère informatique, avec les premières ébauches de J. VON NEUMANN de son calculateur séquentiel, les chercheurs ont initié une réflexion sur des modèles de calcul non forcément séquentiels, à savoir des modèles *parallèles*. Comme les besoins en puissance des utilisateurs sont illimités, les incessants progrès technologiques ne sont pas (et ne seront jamais) suffisants pour répondre à cette demande. Le calcul parallèle est depuis longtemps envisagé comme seconde approche pour augmenter encore plus la puissance des machines du moment. Pourtant, les idées sur le parallélisme n'ont pu être concrétisées que récemment grâce aux progrès de la technologie.

Deux courants de pensées se sont toujours dégagés vis-à-vis de la manière de calculer en parallèle [San91] :

- le mode de pensée « systèmes concurrents » dont le parallèlisme est obtenu par la réalisation de plusieurs actions à la fois. Il conçoit un programme comme un ensemble de tâches s'exécutant en parallèle de manière asynchrone et coopérant pour le calcul du résultat du programme. C'est le parallélisme dû à un *flot de contrôle* multiple.
- le mode de pensée « algorithmique parallèle » qui s'intéresse à l'exploitation du parallèlisme provenant des données d'un programme. Il essaye d'exhiber dans un programme une uniformité de traitement sur ses données. C'est le *parallélisme de données*.

A partir de ces deux courants de pensées, plusieurs modèles de calcul parallèle ont été construits, parfois réalisant une hybridation. Au niveau architecture, on retrouve ce clivage : M.J.FLYNN [Fly66] a proposé une classification des ordinateurs basée sur le flot de contrôle et le flot de données. Il distingue quatre types de calculateurs :

- Single Instruction Stream Single Data Stream (SISD) : il s'agit de l'architecture classique. Elle est calquée sur le modèle de VON NEUMANN avec un seul flot d'instructions opérant sur un seul flot de données.
- Single Instruction Stream Multiple Data Stream (SIMD) : l'amélioration la plus simple que l'on puisse faire, par rapport au type précédent, est d'augmenter le nombre de flots de données pour augmenter la puissance des calculs, tout en gardant un flot de contrôle unique. Une instruction est exécutée sur un ensemble de données de manière parallèle. Chaque donnée est associée à un processeur.
- Multiple Instruction Stream Single Data Stream (MISD) : cette classe regroupe les machines où une série d'opérations est effectuée sur un même flot de données. Les données d'entrées passent par une chaine d'unités de traitement et sont transformées en résultat final à la manière d'un *Pipe-Line*.
- Multiple Instruction Stream Multiple Data Stream (MIMD) : il s'agit de la classe d'architecture la plus générale. Chaque processeur possède son propre flot d'instructions et

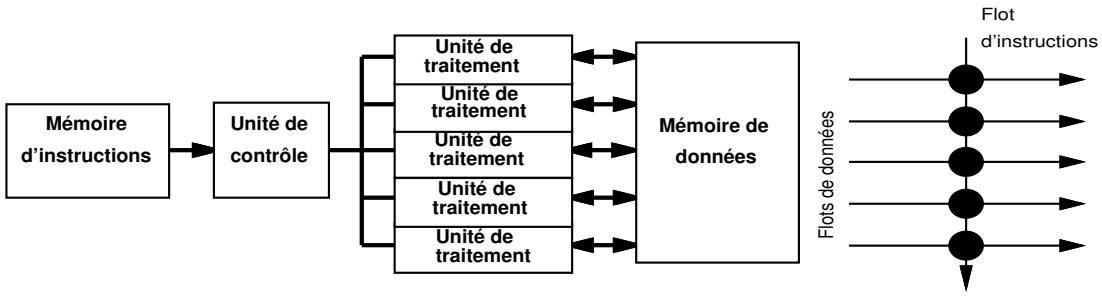

FIG. 1.1 - Architecture typique de machine SIMD avec ses flots.

son propre flot de données. Les processeurs exécutent leurs codes de manière totalement asynchrone les uns par rapport aux autres.

On pourrait rajouter un autre type d'architecture à la taxinomie de Flynn qui serait quelque part entre le SIMD et le MIMD : Single Programme Stream Multiple Data Stream (*SPMD*). Cette architecture pourrait permettre une exécution optimale du modèle à parallélisme de données, habituel par exemple en calcul numérique : globalement on exécute un seul programme mais il peut y avoir des variations locales qu'il faut être capable d'exécuter rapidement [Ker92].

Une définition d'architecture pour ce dernier cas serait globalement une machine MIMD avec des mécanismes efficaces de synchronisation [Ker92].

Chacun de ces types d'architecture est bien adapté pour une classe d'application. Le cas de l'architecture SIMD, auquel nous nous intéressons ici, trouve son intérêt dans des applications où le parallélisme de données est important, comme par exemple des applications scientifiques et graphiques.

1.1 L'Architecture SIMD

La figure 1.1 montre le schéma typique de cette architecture et sa représentation sous forme de flots.

Plusieurs unités de traitements sont pilotées par une unité de contrôle. Celle-ci lit une instruction de la mémoire d'instructions et la diffuse vers la totalité ou une partie des unités de traitement, selon la condition locale à chacune de celles-ci. Chaque unité de traitement recevant l'instruction opère alors sur ses données locales. Ce cycle se répète jusqu'à la fin de l'exécution du programme. En particulier, nous remarquons qu'il existe une synchronisation entre les différentes unités de traitement.

Les calculateurs massivement parallèles (même de type MIMD) sont souvent programmés selon un modèle à parallélisme de données pour sa simplicité et les avantages liés à sa sémantique déterministe. Ce modèle de programmation basé sur la synchronisation des processeurs, a d'abord été mis en œuvre pour des architectures SIMD pour lesquelles la synchronisation est implicite.

1.2 Le Modèle de Programmation à Parallélisme de Données

Ce modèle permet de préciser que certaines opérations vont s'exécuter sur plusieurs données à la fois, d'où son nom. C'est une abstraction du modèle d'architecture SIMD. Le flot de contrôle reste séquentiel comme dans le cas classique. Cela permet la définition des notions de temps global et d'état global. Ces deux notions permettent alors d'une part, la définition d'une sémantique précise pour les langages issues de ce modèle et d'autre part ils permettent

une exécution déterministe de programmes. A un niveau plus bas, cette sémantique précise et ce déterminisme offrent, entre autre, un «*déboggage*» de programme relativement simple. Pour résumer on peut dire que ce modèle de programmation offre une grande puissance d'expressivité tout en restant simple.

POMPC (voir chapitre suivant) a choisie ce modèle de programmation. Il introduit le parallélisme de manière explicite par la définition de classe de variables parallèles à l'aide du mot-clés `collection`. Il fournit également un mécanisme de *virtualisation* qui permet l'écriture de programmes de manière indépendante d'une architecture fixe. Cette virtualisation résulte de la définition de la notion de *processeur virtuel*.

Nous nous intéressons dans le présent rapport à l'implantation de l'opération *préfixe* dans ce langage en portant une attention particulière aux problèmes de virtualisation. D'autre part à l'étude de problèmes sémantiques résultant de l'interaction entre domaine physique et domaine virtuel.

L'opération préfixe admet pour argument une suite de valeurs ordonnées $(a_0, ..., a_{n-1})$ et un opérateur binaire associatif $\oplus$, et retourne la suite ordonnée des produits successifs :

$$P((a_0, ..., a_{n-1}), \oplus) = (p_0, ..., p_{n-1})$$

avec

$$\forall i \in [0, n-1], \ : p_i = a_0 \oplus a_1 \oplus ... \oplus a_i$$

Cette implantation doit se faire en deux temps :

- une première implémentation qui ne considère pas la virtualisation.
- une seconde implémentation qui gére explicitement cette virtualisation.

Le langage POMPC offre aussi la possibilité de la non-virtualisation, au besoin du programmeur. Cette possibilité est nécessaire dès qu'il s'agit de programmer des ressources physiques. Ce type de programmation fait interagir des variables *virtuelles* et des variables *physiques*. De cette interaction peuvent résulter des problèmes sémantiques.

Nous commencerons ce rapport par la présentation au chapitre 2 du langage POMPC. Nous passons au chapitre 3 à la dèscription de l'implantation de l'opération préfixe avec ses deux versions. Au chapitre 4 est détaillée l'interaction entre le domaine physique et le domaine virtuel. Enfin nous terminons, au chapitre 5 par une conclusion où l'on fera le bilan sur le travail effectué tout au long de ce stage.

Chapitre 2

Le Langage POMPC

2.1 Introduction

Le langage POMPC est une extension du langage C pour la programmation de machines parallèles selon un modèle de programmation à parallélisme de données. C'est une esquisse d'un langage *data parallel* C **standard** pour les machines massivement parallèles, puisqu'il rassemble l'ensemble des idées des versions précédentes des langages C pour le parallélisme de données [Mas91] [Wav91] [Thi90] et y inclut des nouvelles. Les concepts clés de POMPC peuvent se résumer en :

- extension de la philosophie de C,
- un modèle de programmation synchrone,
- un parallélisme (de données) explicite,
- un mécanisme de virtualisation,
- accès directe à la machine (la non-virtualisation),
- définition de structures de contrôle de flot parallèles.

2.1.1 Le Modèle de Programmation de POMPC

POMPC est adapté à la programmation de machines massivement parallèles SIMD ou SPMD à mémoire distribuée telles que les machines POMP, CM2 et MasPar MP-1. L'expression du parallélisme en POMPC est explicite; les variables parallèles sont déclarées parallèles de manière explicite et les mouvements de données entre processeurs élémentaires sont également explicites. POMPC permet d'exprimer le modèle de programmation *data parallel*, par un typage fort sur les variables: le parallélisme est exprimé sur les données et non sur le flot d'instructions.

2.1.1.1 Programmation Synchrone

Chaque processeur élémentaire exécute le même programme de manière synchrone. Suivant les possibilités de l'architecture cible, l'exécution sera SIMD ou SPMD. Du point de vue du programmeur, le modèle de programmation est SIMD. La synchronisation SIMD du modèle de POMPC est assurée par la présence dans le modèle de programmation d'un processeur scalaire qui contrôle le séquencement de la machine et définit un flot de contrôle séquentiel.

L'instruction parallèle :

```
A = B + C;
```

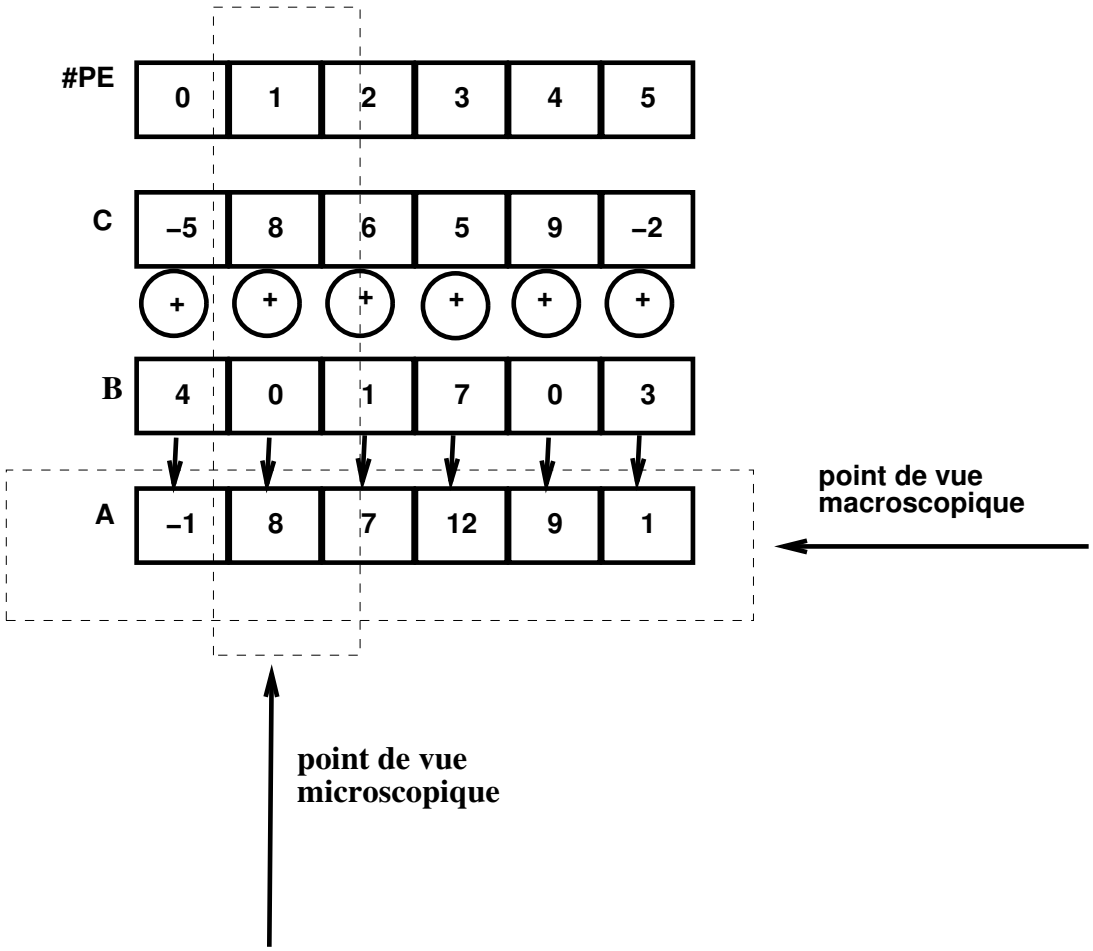

FIG. 2.1 - Une addition parallèle.

de la figure 2.1 réalise l'addition des variables parallèles `A`, `B` et `C`; chaque processeur physique (PP) contient un élément des trois variables parallèles et exécute l'opération d'addition sur ses éléments qui résident dans sa mémoire locale. On peut considérer une instruction parallèle selon deux points de vue :

- le point de vue microscopique (vertical sur la figure 2.1): point de vue local à chaque processeur physique, avec ses données propres.

- le point de vue macroscopique (horizontal sur la figure 2.1): point de vue qui considère une variable parallèle comme une entité indissociable du programme.

En POMPC les variables parallèles sont considérées comme des entités du langage. Le modèle de programmation *data parallel* se prête bien à des applications nécessitant un traitement identique sur les éléments d'un tableau. Un tel tableau peut être vue, en POMPC, comme une variable parallèle permettant ainsi l'expression naturelle du traitement en question. Chaque élément du tableau est affecté à un processeur élémentaire. Néanmoins la taille de ce tableau est, a priori, une donnée du problème mais ne coïncide pas forcément avec le nombre de processeurs de la machine; une solution consiste à offrir au programmeur une vue *virtuelle* de la machine, adaptée à son application et à gérer celle-ci au niveau de la compilation.

2.1.1.2 Les Processeurs Virtuels (PV)

POMPC offre au programmeur une vue virtuelle de la machine sur laquelle il traite son application. S'il doit faire des calculs sur des variables parallèles de taille n, alors il dispose d'une machine virtuelle dont le nombre de *processeurs virtuels* est n. Le programmeur considère que chaque élément du tableau dispose d'un processeur (virtuel) pour effectuer les calculs parallèles. La virtualisation est gerée par le compilateur POMPC. Si une même application utilise des variables parallèles de taille différentes, on définit pour celle-ci, autant de machines virtuelles qu'il existe de types différents de variables parallèles. On découpe ainsi l'ensemble des variables parallèles d'un programme POMPC en classes; chaque classe est associée à une machine virtuelle et tout les éléments d'une même classe possèdent le même nombre d'éléments et la même topologie. Dans le vocabulaire POMPC une classe est définie par le mot-clé `collection`. Cet attribut définit une machine virtuelle. Il existe de ce fait, dans un programme POMPC, autant de machines virtuelles qu'il y a de déclaration différentes de classes de variables parallèles, déclarées par le mot-clé `collection`. Les variables déclarées n'appartenant pas à une collection sont des variables scalaires, allouées dans le processeur scalaire.

Les calculs dans un programme POMPC font intervenir des variables appartenant à différentes collections. Les interactions vont être de deux types :

- directes : éléments à éléments à l'intérieur d'une même collection. Ces interactions sont locales à chaque PV;
- indirectes : entre éléments appartenant à des collections différentes ou entre éléments d'une même collection mais suivant un schéma d'adressage non local à chaque PV. Ce sont des cas de communication entre machines virtuelles.

2.1.1.3 Les Communications Explicites

Pour accroître la lisibilité des programmes sources et pour des raisons de performances, POMPC propose une syntaxe où les communications sont explicites. Seules les interactions directes sont autorisées dans l'extension de la syntaxe de C; les interactions indirectes sont spécifiées explicitement par des opérateurs de communication.

2.2 Les Collections

Le parallélisme est explicitement introduit dans POMPC par la définition de *collections* qui sont des classes de variables parallèles. Définir une collection revient à décrire l'organisation typique d'une variable de cette classe. La déclaration:

```
collection [1152,900] pixel;
```

définit la classe de variables parallèles `pixel`, qui est une machine virtuelle sous forme d'une grille à deux dimensions de 1152×900 processeurs virtuels. L'identificateur `pixel` devient alors dans la suite du programme un symbole utilisé :

- comme identificateur de la collection;
- comme attribut de type lors de la déclaration de variables parallèles.

Une variable de cette collection possède alors 1152×900 éléments, organisés sous forme d'une grille de 1152 colonnes et de 900 lignes. Deux informations essentielles sont attachées à une collection :

- la géométrie et la taille de cette collection;
- une activité parallèle, qui définit pour chaque processeur virtuel s'il est actif ou non pour exécuter l'instruction parallèle courante. Le constructeur `where` (voir la section suivante) permet cette sélection entre processeurs virtuels actifs et inactifs.

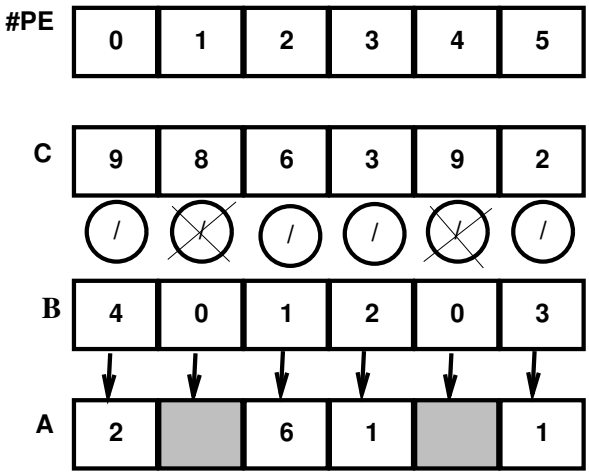

FIG. 2.2 - Une division parallèle.

La notion de typage d'une expression C est étendue en POMPC avec un attribut supplémentaire : la collection. A chaque expression est associée une collection. L'expression est alors évaluée sur la machine virtuelle associée à cette collection, exception faite pour une expression n'utilisant que des variables scalaires qui est évaluée au niveau du processeur scalaire.

La déclaration

```
collection pixel [100,100];

pixel char ecran;
int j;
```

indique que la variable `ecran` est une variable parallèle de collection `pixel` et la variable `j` est une variable scalaire.

2.3 Le contrôle de flot

Le caractère synchrone du modèle de programmation de POMPC implique un flot de contrôle unique. Il interdit l'existence d'un flot de contrôle parallèle. Le contrôle du flot reste la tâche du processeur scalaire et est réduit, en première approximation, au contrôle de flot scalaire classique de C et à l'opérateur de manipulation de l'activité des processeurs virtuels `where`. Tout programme POMPC est constitué d'une ossature C sur laquelle sont greffés des instructions parallèles. Les processeurs virtuels sont tributaires de ce contrôle de flot unique et ne disposent que de l'opérateur `where` pour adapter le flot unique à une condition locale. Le `where` ne modifie pas le flot d'instruction mais uniquement l'activité des processeurs virtuels de la collection à laquelle il se rapporte. En particulier une instruction scalaire à l'intérieur d'un bloc `where` est automatiquement exécutée même si aucun processeur élémentaire n'est actif. En effet cette instruction ne concerne que le processeur scalaire. Le constructeur `where` peut être aussi compris comme un `if` parallèle avec la différence qu'il ne modifie que l'activité des processeurs virtuels. La figure 2.2 montre un cas de nécessité de cet opérateur, la division

```
A = C / B;
```

s'effectue de manière anormale à cause de certaines valeurs nulles de la variable parallèle `B`. La construction permettant de spécifier le traitement voulu est la suivante :

```
where(B != 0) {
    A = C / B;
}
```

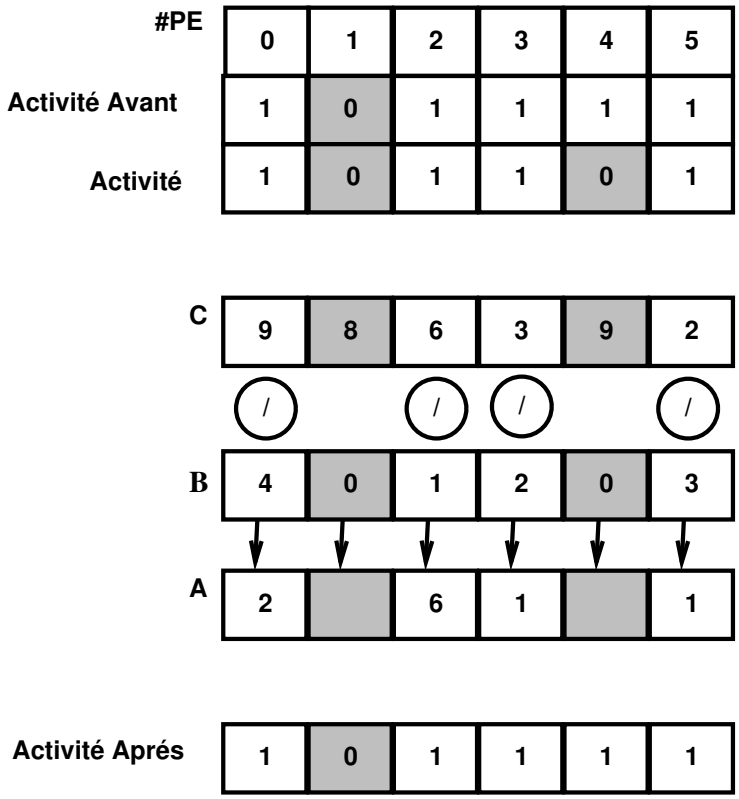

FIG. 2.3 - Une division parallèle.

qui entraine un déroulement des opérations comme sur la figure 2.3.

A l'entrée du bloc `where` l'activité des processeurs est sauvegardée et mise à jour en fonction de la condition paralléle `(B != 0)` pour ce bloc. A la fin de l'exécution de celui-ci l'ancienne activité est restaurée.

A partir du contrôle de flot classique de C et de l'opérateur `where`, un pseudo-contrôle de flot parallèle a été défini; par exemple le constructeur `whilesomewhere` exprime en quelque sorte un `while` parallèle :

```
whilesomewhere(exp) {
   ...
}
```

où `exp` est une expression parallèle. La boucle est exécutée tant qu'il existe un processeur virtuel actif vérifiant l'expression booléenne `exp` ; la non vérification, par un processeur a priori actif de cette condition rend celui-ci inactif jusqu'à ce que tous les PV deviennent inactifs.

Ce pseudo-contrôle de flot parallèle est utile pour des raisons de lisibilité et d'optimisation.

2.4 Les communications

POMPC propose une syntaxe où les communications entre machines virtuelles sont explicites : c'est l'unique moyen de faire interagir des objets appartenant à des collections différentes ou des objets appartenant à une même collection mais suivant un schéma d'interaction non local à chaque processeur virtuel.

Il existe six types de communications en POMPC :

– diffusion scalaire : c'est l'unique cas de communication qui n'est pas explicitement spé-

cifié par un opérateur de communication. La diffusion scalaire consiste à distribuer une valeur scalaire à l'ensemble des processeurs actifs. Le mécanisme de typage de POMPC transforme cette valeur en une valeur parallèle.

- émission scalaire : elle sert à lire une valeur particulière d'une variable parallèle.
- réception scalaire : elle sert à écrire une valeur particulière d'une variable parallèle.
- réduction scalaire associative : les réductions scalaires associatives servent à récupérer le résultat de la combinaison des éléments actifs d'une variable parallèle, à l'aide d'un opérateur binaire associatif.
- émission parallèle : c'est un cas de communication entre deux variables parallèles suivant un schéma de routage donné par une troisième. L'émission parallèle s'écrit :

```
pixel char Destination;
line unsigned int Address;
line char Source;

...
[Address]Destination <- Source;
...
```

L'émission parallèle se caractérise par la présence de l'opérateur d'indirection (`[ ]`) en membre gauche de l'expression. Le typage impose que la source et l'adresse de la communication appartiennent à la même collection. Elle fonctionne de la manière suivante :

1. Chaque processeur virtuel de la collection `line` (celle de `Source` et `Address`) envoie la valeur de l'élément local de la variable `Source` au processeur virtuel de la collection `pixel` (celle de destination) dont l'adresse est la valeur de l'élément local de la variable `Address`.
2. Chaque processeur virtuel de la collection d'arrivée (`pixel`), qu'il soit actif ou non, range chaque donnée reçue dans l'élément de la variable `Destination` qu'il possède.

- réception parallèle : c'est le même type de communication qu'une émission parallèle, sauf que dans ce cas il s'agit d'une lecture de valeur et non pas d'une écriture. La réception parallèle s'écrit :

```
line char Destination;
line unsigned int Address;
pixel char Source;

...
Destination=[Address]Source;
...
```

Elle fonctionne de la manière suivante :

1. Chaque processeur virtuel de la collection de destination émet une requête au processeur virtuel de la collection de source, dont le numéro est la valeur de l'élément local de la variable d'adresse.
2. Lorsqu'un processeur virtuel (actif ou non) de la collection de source reçoit une requête, il renvoie la valeur locale de l'élément qu'il posséde au processeur virtuel qui lui en a fait la demande.
3. Lorsqu'un processeur virtuel de la collection de destination reçoit le résultat de sa requête, il le range dans la variable destination.

Mise à part la diffusion scalaire, toutes ces communications sont caractérisées dans la syntaxe POMPC, soit:

- par la présence d'une flèche vers la gauche (`<-`)
- soit par la présence d'un opérateur de *réarrangement* (`[[ ]]`,`[ ]`), pour l'indexation.

2.5 La Virtualisation

La virtualisation est le processus qui permet de cacher à l'utilisateur la taille et la topologie réelles de la machine sur laquelle il programme son application, le programmeur définit la géométrie et la taille de ses variables parallèles selon ses besoins et n'a pas à s'occuper de la gestion de celles-ci sur la machine physique. Ce processus de virtualisation résulte de la liberté de déclarer les collections en POMPC de topologie et de dimension quelconques. En effet, le programmeur travaille sur les machines virtuelles associées aux collections qu'il a définies dans son programme, et non pas directement sur la machine physique (voir le chapitre suivant).

POMPC offre aussi la possibilité d'accès aux ressources physiques de la machine pour des cas de programmation de bas niveau. En effet la déclaration

```
collection physical processor;
```

déclare la collection `processor` comme étant une collection physique. Toute variable de cette collection est allouée en un élément sur chaque processeur physique et possède de ce fait autant d'éléments qu'il y a de processeurs physiques.

Il est possible aussi de déclarer une variable dans la projection physique d'une collection virtuelle :

```
collection pixel [100,100];

physical pixel int tab;
```

La variable `tab` admet pour topologie celle de la collection pixel projetée sur le domaine physique mais possède un nombre d'éléments égal au nombre de processeurs physiques.

Le processus de virtualisation est au centre du problème de compilation de POMPC et est un point très important pour permettre une portabilité de programmes POMPC efficace.

Une description succinte de POMPC peut être trouvée dans [Par92a]. On peut aussi consulter le document [Par92b] pour une description des concepts clés de POMPC.

Chapitre 3

L'Opérateur Préfixe : SCAN

3.1 Introduction

Dans beaucoup de domaines d'application, deux problèmes fondamentaux apparaissent en phase ultime d'écriture d'un algorithme :

- le calcul d'un produit de valeurs scalaires : une *réduction*,
- le calcul d'une opération préfixe sur un ensemble de données également scalaires : un *scan*.

On peut définir la réduction comme suit :

Définition 1 *La réduction suivant l'opérateur binaire associatif $\oplus$ des n éléments scalaires a_i i=0..n-1, est le calcul du produit :*

$$\bigoplus_{i=0}^{n-1} a_i = a_0 \oplus a_1 \oplus \cdots \oplus a_i \oplus \cdots \oplus a_{n-1}$$

De la même manière, on peut définir l'opération préfixe comme suit :

Définition 2 (Kruskal[KRS85]) *L'opération préfixe admet pour argument une suite de valeurs ordonnées $(a_0, ..., a_{n-1})$ et un opérateur binaire associatif $\oplus$, et retourne la suite ordonnée des produits successifs :*

$$S_{Kruskal}((a_0, \cdots, a_{n-1}), \oplus) = (p_0, \cdots, p_{n-1})$$

avec

$$\forall i \in [0, n-1],\ p_i = a_0 \oplus a_1 \oplus \cdots \oplus a_i$$

Une autre définition est proposée par Guy Blelloch en ces termes :

Définition 3 (Blelloch[Ble89b]) *Une opération préfixe prend un opérateur binaire associatif $\oplus$, ayant pour élément neutre ε, et une suite ordonnée $(a_0, \cdots, a_{n-1})$ et retourne la suite ordonnée des produits successifs :*

$$S_{Blelloch}((a_0, \cdots, a_{n-1}), \oplus) = (p_0, \cdots, p_{n-1})$$

avec

$$\forall i \in [1, n-1],\ p_i = a_0 \oplus a_1 \oplus \cdots \oplus a_{i-1}$$

et

$$p_0 = \varepsilon$$

En ce qui concerne notre travail d'implantation de fonctions scan dans le cadre du stage, nous adopterons la première définition qui nous semble un peu plus cohérente que la deuxième (en effet cette dernière ne tient pas compte du dernier élément a_{n-1} de la suite $(a_0, \cdots, a_{n-1})$) et la nécessité d'un élément neutre pour des besoins d'implantation.

L'opération préfixe est désignée sous le terme d'opérateur parallèle préfixe, quand le calcul de la suite $a_0 \oplus a_1 \oplus \cdots \oplus a_i$, $i = 0 \ldots n-1$ s'effectue de manière parallèle.

Historiquement, l'opérateur préfixe a été originalement introduit par IVERSON vers le milieu des années 50 dans le langage APL sous le terme de «*scan* ». Plusieurs termes ont été adoptés pour cette opération dans la littérature, mais le terme « *scan* » (qui veut dire : balayer, parcourir,...) nous semble être le plus concis et le plus adéquat. Nous l'utiliserons dans toute la suite du rapport.

Bien qu'à première vue, le calcul d'un scan sur une suite de valeurs a_i, $i = 0..n-1$, avec un opérateur diadique associatif $\oplus$, semble être un processus purement séquentiel, cela n'est pas vraiment le cas. En effet on pourrait penser que la séquentialité du processus vient de la constatation naturelle que le calcul de la valeur

$$p_i = a_0 \oplus a_1 \oplus \cdots \oplus a_i = p_{i-1} \oplus a_i$$

ne peut démarrer avant que le calcul de la valeur p_{i-1} soit terminé. C'est le cas lorsque l'on adopte une approche totalement séquentielle sur une machine classique de type VON NEUMANN. En particulier la complexité d'une telle méthode de calcul est en $o(n)$. Dans le cas qui nous intéresse, nous cherchons à améliorer le temps de calcul de cette opération sur une machine parallèle.

Nombreuses sont les applications qui, comme pour le cas du scan, apparaissent en première approche intrinséquement séquentielles, donc nécessitant un temps de calcul de l'ordre de la taille des données. Des exemples de telles applications [HS86] sont :

- la recherche du dernier élément d'une liste chaînée;
- le calcul de la somme de tous les éléments d'un tableau.

mais cela est probablement dû à notre mode de pensée, qui en premier constat, aborde un problème donné de manière séquentielle et non pas parallèle, et concentre la solution algorithmique de celui-ci sur le flot de contrôle et non pas sur la donnée: c'est le mode de pensée séquentiel. En revanche en essayant d'identifier un certain type de traitement qui se répéterait dans un contexte donné dans les deux applications citées ci-dessus, donc un certain type de parallèlisme, nous remarquerons bien qu'il existe un contexte, la donnée élémentaire, sur laquelle est effectuée à chaque fois un traitement similaire, c'est le parallélisme de données. En effet en adoptons une approche **SIMD** (en se donnant assez de processeurs), pour les deux problèmes ci-dessus, nous pouvons ramener leur complexité à un ordre $\log n$ (n étant la taille du problème) [HS86].

Pour illustrer cela supposons une architecture de type SIMD classique, avec un nombre de processeurs qui est une puissance de deux et chaque processeur contenant un élément du problème à traiter. Nous ne tenons pas compte du mécanisme de gestion de données dans le cas où la taille de celles-ci dépasse le nombre de processeurs :

recherche du dernier élément d'une liste chaînée: dans [HS86] est décrit un algorithme en $o(\log(n))$ pour la recherche du dernier élément d'une liste chaînée (en supposant que toute les communications entre processeurs physiques prennent le même temps quel que soit la distance entre ces processeurs). Chacune des cellules est placée dans un processeur et possède un pointeur vers la prochaine cellule dans la liste. La dernière cellule contenant la valeur `null`. On suppose aussi que chaque cellule contient un champ `chum` utilisé pour des calculs temporaires. L'idée principale est la suivante :

chaque processeur copie l'adresse de son suivant dans la liste dans son champ `chum`. Il remplace ensuite, de manière répetétive, la valeur de cette variable par la valeur de `chum[chum]`. Cependant si la valeur du champ `chum` est `null`, celle-ci reste inchangée.

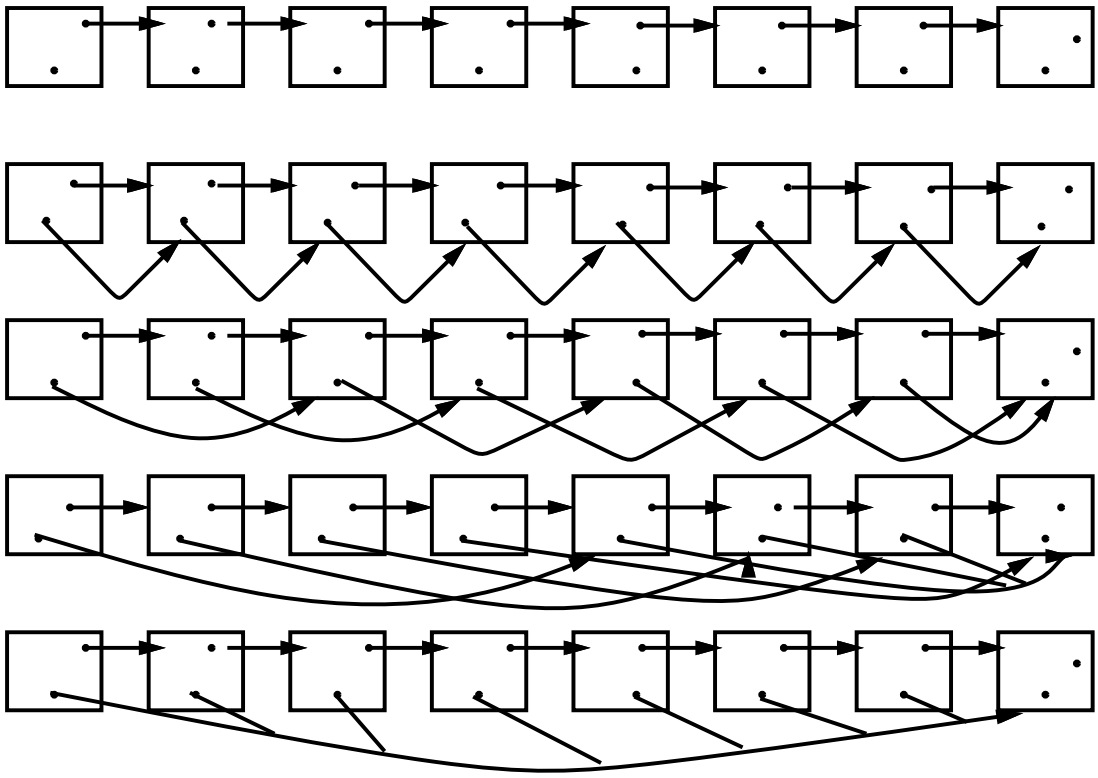

FIG. 3.1 - Recherche du dernier élément d'une liste chainée.

Pour assurer que la première cellule trouve l'adresse de la dernière, un processeur ne change pas la valeur du champ `chum` si la valeur de `chum[chum]` est `null`. le traitement est illustré graphiquement sur la figure 3.1 qui correspond à l'algorithme ci-dessous.

```
for all k in parallel do
    chum[k] = next[k]
    while (chum[k] ≠ null and chum[chum[k]] ≠ null) do
          chum[k] = chum[chum[k]]
    od
od
```

La boucle `while` est exécutée par tous les processeurs tant qu'ils vérifient la condition. Dès qu'un processeur ne la verifie pas, il devient inactif.

le calcul de la somme de tous les éléments d'un tableau: dans le même article est décrit un algorithme pour le calcul de la somme des éléments d'un tableau en $o(\log n)$. La figure 3.2 représente un déroulement des opérations pour un cas de 16 éléments, suivant l'algorithme:

```
for j=1 to log(n) do
    for all k in parallel do
        if ((k+1) mod 2^j)=0 then
           x[k]= x[k-2^(j-1)]+x[k]
        fi
    od
od
```

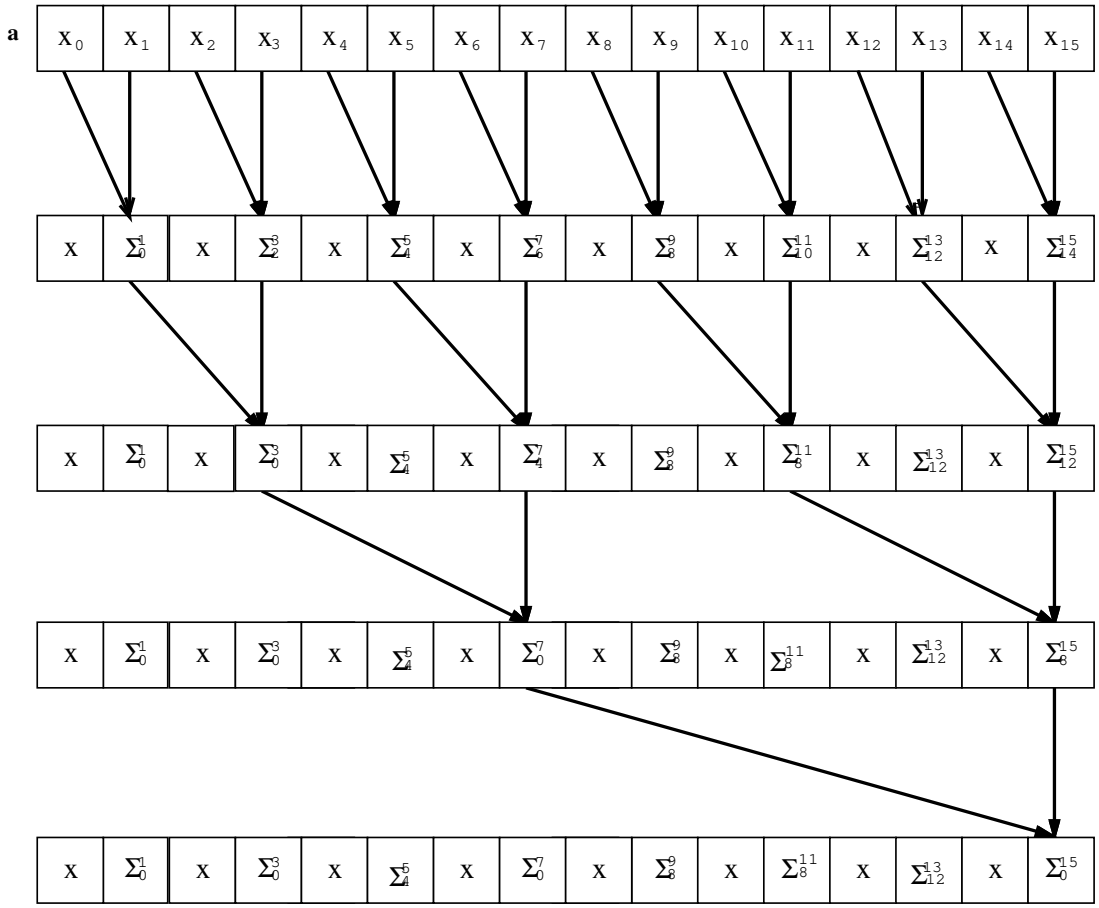

FIG. 3.2 - Calcul de la somme des éléments d'un tableau.

$\log(n)$ itérations sont effectuées. A chaque itération j, $n/2^j$ processeurs sont selectionnés et effectuent une somme partielle, jusqu'à obtention du resultat.

De même, pour l'opérateur scan, une implantation efficace permettant de réduire des chaînes de dépendances de taille linéaire en des arbres de dépendance de taille logarithmique est envisageable sur une architecture de type SIMD.

Il est à noter que la possibilité du calcul parallèle de l'expression

$$S((a_0, \cdots, a_{n-1}), \oplus) = (p_0, \cdots, p_{n-1})$$

(un scan suivant l'opérateur $\oplus$, sur la suite $(a_0, \cdots, a_{n-1})$), vient de la propriété d'associativité de l'opérateur $\oplus$; en effet la possibilité d'écrire par exemple

$$p_3 = \bigoplus_{i=0}^{3} a_i = (((a_0 \oplus a_1) \oplus a_2) \oplus a_3) = a_0 \oplus a_1 \oplus a_2 \oplus a_3$$

nous permet de déduire le schéma de calcul suivant pour cette valeur :

$$p_3 = ((a_0 \oplus a_1) \oplus (a_2 \oplus a_3))$$

donc, évaluation en parallèle des deux expression $(a_0 \oplus a_1)$ et $(a_2 \oplus a_3)$, suivie de l'évaluation du résultat c'est à dire deux étapes de calculs au lieu de trois sur une machine séquentielle classique.

Avant de discuter de l'implantation de la fonction scan, interessons nous à l'état de l'art dans le domaine.

3.2 L'Etat de L'Art

Dans l'état de l'art, on trouve particulièrement les travaux de Blelloch. En effet, dans [Ble89b] il mentionne l'importance d'inclure, dans le modèle PRAM, certaines opérations de scan comme des primitives élémentaires. A travers une série d'algorithmes, il montre la puissance d'expression du scan. L'article est complètement repris dans sa thése [Ble89a] (voir section suivante) où il développe un modèle de machine utilisant fortement le scan.

L'article de [HS86] nous a été d'une grande utilité. Les auteurs décrivent une série d'algorithmes classiques suivant un modèle à parallélisme de données. Le but de l'article étant de montrer un style de programmation, en l'occurence, le modèle data parallel (l'article date de 1986). L'algorithme choisi pour l'implantation de la fonction scan est l'un des algorithmes décrit dans cet article.

Dans [KRS85] est décrit un algorithme déterministe pour le calcul du scan dans le cas où les éléments sont rangés dans une liste chainée. Cette algorithme admet une complexité en $o((\log n)/\log(2n/p)) \cdots (n/p))$; n étant la taille de la donnée en entrée et p le nombre de processeur.

Dans [WN91] est présenté un algorithme optimal pour le calcul du scan. Le temps d'exécution de l'algorithme est $\lceil (2n/(p+1)) \rceil$ pour p processeurs et n données d'entrée avec la contrainte $n \geq p(p+1)/2$. Les auteurs prouvent l'optimalité de leur résultat.

Une construction récursive est utilisée dans [LF80] pour obtenir un circuit calculant le scan. Ce circuit admet une *profondeur* de $\lceil \log n \rceil$ exacte et une taille bornée par $4n$, si l'on suppose que le scan est effectué sur une donnée de taille n.

Nous avons aussi consulté deux articles sur le langage `Fortran`. L'article de [Ber91] présente une critique du langage `Fortran 90`. L'auteur fait une comparaison entre ce langage et le langage `APL` sur divers aspect. Il argumente que dans son orientation de langage pour la manipulation de tableau, `Fortran 90` n'a pas atteind ses objectifs. Le second article [AKLGLS88] décrit des techniques de compilation pour le langage `Fortran 8x` dont la cible est la Connection Machine.

Notons aussi qu'un certain nombre d'articles traitant de la relation entre les recurrences linéaires et les scans ont été consultés. Cela dans le but d'entamer le second volet de la partie théorique, mais à défaut de temps, nous n'avons pas pu mener à terme cette partie [Cal91] [Kog74] [KS73].

3.3 La Thèse de Blelloch

Dans sa thèse, Blelloch définit une classe de modèles de machines qu'il dénomme *parallel vector* et argumente que ces modèles sont excellents aussi bien d'un point de vue modèle algorithmique pour l'étude de la complexité des algorithmes, que d'un point de vue ensemble d'instructions pour une machine virtuelle sur laquelle des langages de programmation de haut niveau seraient compilés. Il partage sa thèse en quatre chapitres : en développant dans un premier chapitre les différents modèles, définissant un modèle comme étant une machine abstraite et un ensemble d'instructions primitives. Ces différents modèles ne diffèrent alors que par leur ensemble d'instructions; la machine abstraite étant la même pour tous les modèles. Il introduit

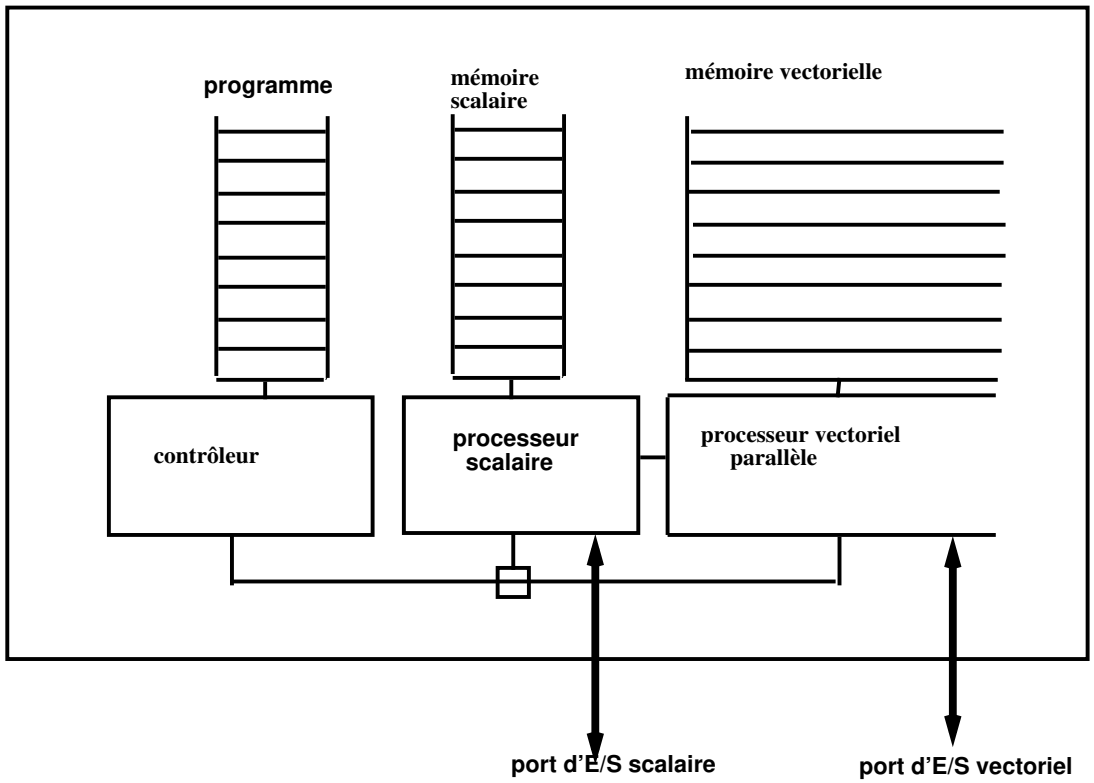

FIG. 3.3 - L'architecture de la machine parallel vector model.

ensuite le modèle *Scan Vector*, dont l'une des instructions primitives est le scan. Au second chapitre, il donne une version d'un ensemble d'algorithmes classiques parmi les sujets de calcul géométrique, graphes, et analyse numérique dans son modèle, en l'occurrence utilisant le scan. Le chapitre 3 est consacré aux langages et compilateurs: il décrit un compilateur pour le langage *Paralation Lisp*, dont la cible est le modèle scan vector model, enfin il termine par un chapitre sur l'architecture où un circuit pour le scan est proposé, dont la complexité de calcul est en $2\log(n)$.

3.3.1 Le Modèle Scan Vector

Comme pour les autres cas de description de modèle de calcul (la machine de Turing, le modèle PRAM,...), le modèle scan vector est décrit en terme d'une machine abstraite et d'un ensemble d'instructions élémentaires (ce que peut faire la machine, i.e. son langage). Le modèle scan vector manipule comme donnée élémentaire de base le *vecteur*. Un programme est une suite d'instructions séquentielles manipulant des vecteurs et non pas un ensemble de processus séquentiels s'exécutant en parallèle, le parallélisme dans ce modèle vient alors de l'exécution séquentielle de primitives parallèles et non pas de l'exécution parallèle de primitives séquentielles. Nous remarquons en particulier que le flot de contrôle est séquentiel.

3.3.1.1 La Machine Parallel Vector

La figure 3.3 illustre le schéma de base de la machine parallel vector.

C'est une machine séquentielle à accès aléatoire (une RAM classique), augmentée d'une mémoire vectorielle (pour le stockage des vecteurs), d'un processeur vectoriel parallèle (pour la manipulation de vecteurs) et d'un port d'entrées/sorties vectorielles.

La mémoire vectorielle est une séquence de sites, chacune d'elles contenant un vecteur. Un vecteur est une collection linéaire ordonnée de valeurs scalaires. Aucune limite n'est faite sur la longueur d'un vecteur, pour une implantation donnée sur une machine réelle il est alors nécessaire d'exhiber un moyen pour *mapper* ces vecteurs sur l'architecture cible (cela est un peu équivalent à la gestion des collections dans le langage POMPC, c'est à dire la gestion de la virtualisation).

Le processeur vectoriel parallèle exécute des instructions primitives sur un nombre fixé de vecteurs et de scalaires. Il peut par exemple lire un vecteur de la mémoire vectorielle, effectuer une réduction suivant la multiplication et ranger le résultat dans la mémoire scalaire. Il peut aussi lire deux vecteurs ordonnés, effectuer leur fusion-ordonnancement et ranger à nouveau le résultat dans la mémoire vectorielle.

3.3.1.2 Les Primitives de Base

L'ensemble des instructions de ce modèle peut être divisé en trois sous-ensembles de primitives élémentaires: les instructions vectorielles, les instructions scalaires et les instructions scalaires-vectorielles. Les instructions vectorielles sont à nouveau partagées en trois classes : élément-à-élément, scan et permutation.

3.3.1.2.1 Les Instructions Scalaires. Les instructions scalaires sont les instructions classiques d'une machine séquentielle.

3.3.1.2.2 Les Instructions Elément-à-Elément. Une instruction élément-à-élément prend deux vecteurs de même longueur et les combine élément-à-élément en utilisant un opérateur diadique, le résultat est un vecteur de même longueur.

3.3.1.2.3 Les Instructions de Permutation. Une instruction de permutation prend un vecteur de données et permute ces éléments suivant un deuxième vecteur index de même taille.

3.3.1.2.4 Les Instructions Scalaires-Vectorielles. Une instruction de ce type peut par exemple extraire une valeur scalaire d'un vecteur de données en utilisant un index scalaire.

3.3.1.2.5 Les Instructions Scan. C'est le type d'opération déjà défini. Un scan avec l'addition sur un vecteur A = [1 2 3 4 5 6 7 8] à la forme suivante :

$$\begin{array}{rcccccccccc} A & = & [& 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 &] \\ +-\text{scan}(A) & = & [& 0 & 1 & 3 & 6 & 10 & 15 & 21 & 28 &] \end{array}$$

Toutes ces instructions sont supposées (dans le modèle) être des instructions unitaires, c'est à dire qu'elles prennent une unité de temps.

3.4 Implantation de l'Opérateur Scan

Nous discutons dans cette section de l'implémentation de la bibliothèque de scan. La fonction scan a été implantée avec les sept opérateurs associatifs suivants :

- l'addition,
- la multiplication,
- le maximum,
- le minimum,
- le «et» logique,

- le «ou» logique,
- le «ou» exclusif,

avec les types de données suivants :

- caractère (`char`),
- mot (`short`),
- entier (`int`),
- caractère non signé (`unsigned char`),
- mot non signé (`unsigned short`),
- entier non signé (`unsigned int`),
- flottant (`float`),
- flottant double précision (`double`),

en simulation sur une station de travail SUN. La simulation (pour le langage POMPC) consiste à émuler l'activité d'un processeur physique de l'architecture SIMD par un processus UNIX.

Deux versions de la fonctions scan ont été implantées :

- **sans tenir compte de la virtualisation** : On écrit le scan comme si on avait autant de PV que de processeurs physiques. On ne tient pas compte de la virtualisation. Cette méthode est efficace si le nombre de PV est égale au nombre de PP (`vp-ratio`= 1). Soit un scan physique; On l'obtient en remplaçant la collection virtuelle par une collection physique. Si on veut être optimal, il faut tenir compte de la virtualisation, d'où la seconde version.
- **avec gestion de la virtualisation** : dans ce cas, la gestion de la virtualisation est faite explicitement. On suppose qu'un processeur physique est partagé par plusieurs éléments de la donnée à traiter, c'est à dire en terme de POMPC, qu'il existe plusieurs processeurs virtuels par processeur physique. Il faut donc tenir compte du nombre et mode d'arrangement des processeurs virtuels sur un même processeur physique.

On note donc que la première version est un scan physique qui travaille sur le réseau de processeur et est de plus bas niveau que la seconde version. Celle-ci fait appel à la première routine pour récupurer un premier résultat. Au niveau complexité des deux versions, si l'on suppose que globalement le scan physique de p valeurs admet une complexité en $o(\log p)$ sur un reseau à p processeurs (ce qui est le but de l'implantation), la deuxième version aura alors une complexité en $o(n/p + \log p)$ pour une donnée paralléle de taille n. En effet une phase séquentielle locale à chaque processeur physique est nécessaire, induisant, de manière générale, un facteur de complexité de $o(n/p)$ en plus.

Pour la description de l'implantation de la première version du scan, nous supposons un modèle de machine de type SIMD classique avec un nombre de processeurs qui est une puissance de deux et égal à la taille du vecteur en entrée sur lequel nous voulons effectuer le scan, de sorte que chaque élément du vecteur est stocké dans un processeur élémentaire. Nous utiliserons comme opérateur l'addition pour la description des algorithmes, avec une collection de huit éléments (dans la majorité des cas) pour ne pas alourdir les schémas.

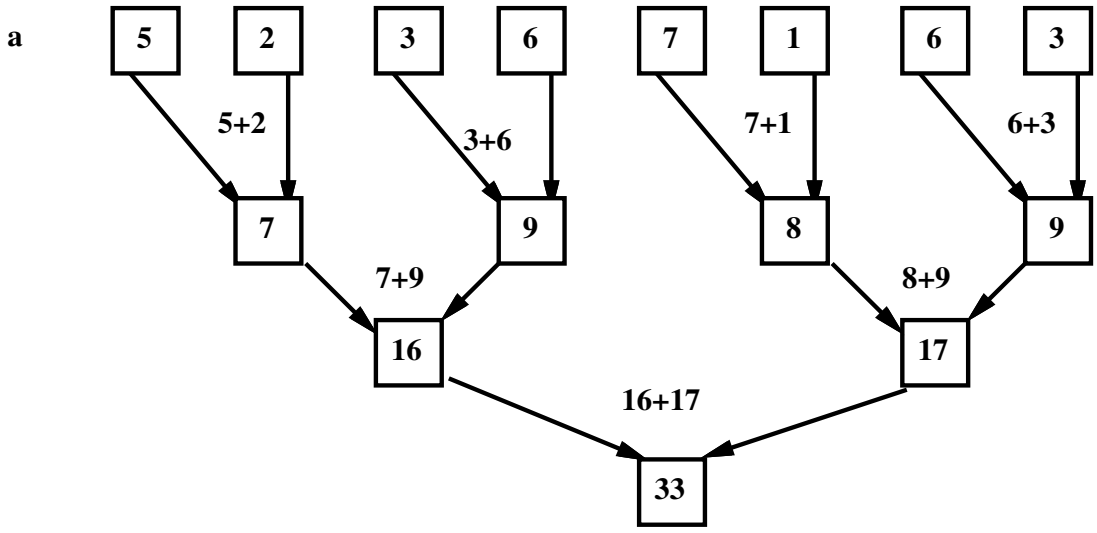

FIG. 3.4 - La phase montante de l'algorithme.

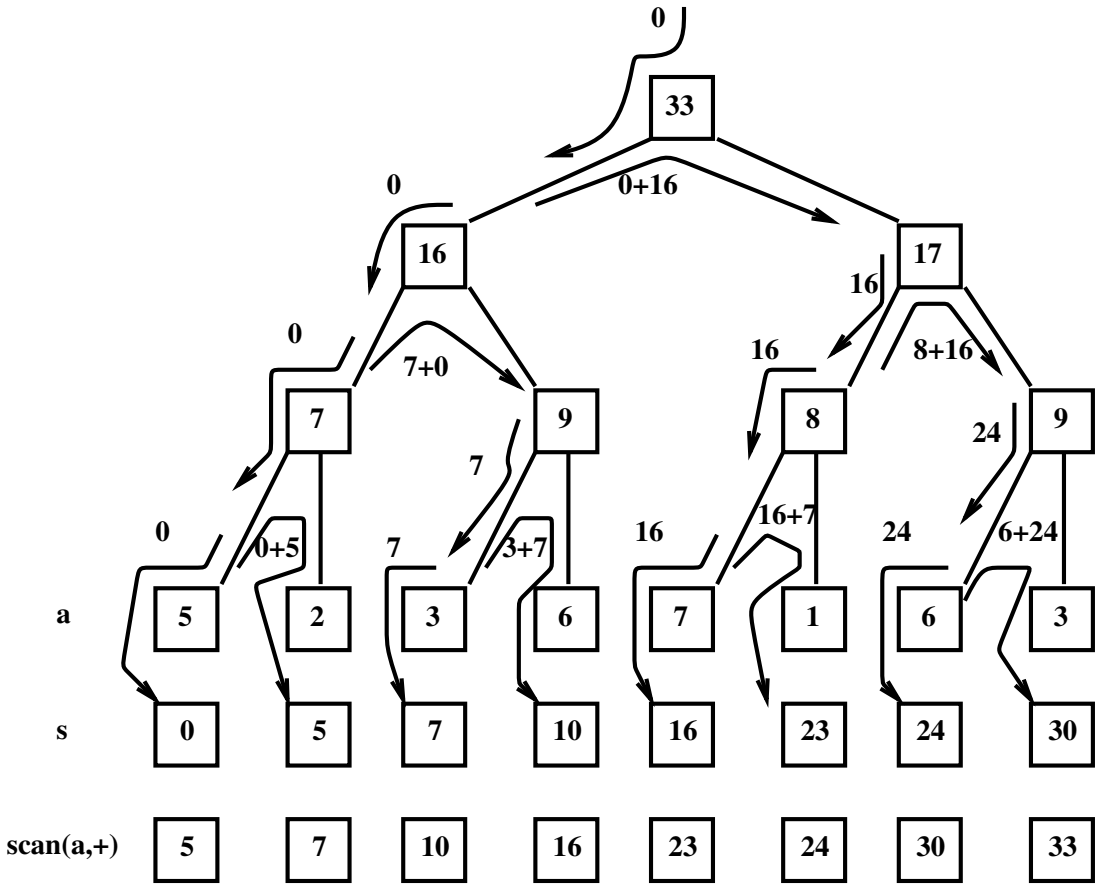

FIG. 3.5 - La phase descendante de l'algorithme.

3.4.1 Premier Algorithme : Algorithme à Deux Passes

Le premier algorithme ayant été étudié pour l'implantation de la bibliothèque de scan est l'algorithme à deux passes décrit par les deux figures 3.4 et 3.5.

Cet algorithme est constitué de deux phases: une première phase montante qui construit un arbre binaire sur le réseau de processeurt et une seconde phase descendante qui parcourt l'arbre depuis la racine et calcule le résultat final.

- **phase montante**: Cette phase calcule, dans une variable temporaire, un arbre binaire. L'arbre est construit en combinant suivant l'opérateur diadique associatif du scan, à chaque itération i, un élément avec son voisin situé à une distance de 2^i, $i = 0..\log(n)-1$ et en stockant le résultat dans le processeur situé à égale distance des deux premiers, qui va constituer alors leur père (exception faite pour la première étape, où le processeur père est confondu avec le processeur fils droit), cet élément représente alors la réduction, suivant l'opération associée, de toutes les feuilles du sous arbre dont il est la racine. Cette première phase prend donc $\log n$ étapes (nombre de niveaux dans l'arbre) et possède de ce fait une complexité en $o(\log n)$.

  Il important de noter ici la manière dont est plongé l'arbre binaire sur la structure linéaire du réseau de processeurs, de sorte que les processeurs utilisés à l'itération i ne sont pas utilisés à l'itération $i+1$; ce qui permet de stocker l'arbre dans une variable parallèle.

- **phase descendante**: Cette phase part de la racine de l'arbre, en affectant à celle-ci l'élément neutre de l'opérateur $\oplus$ et parcourt l'arbre niveau par niveau. A chaque itération i, les processeurs sélectionnés (les processeurs du niveau correspondant à l'itération i), transmettent au fils droit le produit (suivant l'opérateur $\oplus$) de leurs valeurs et celles du fils gauche, et transmettent au fils gauche leurs valeurs. A la fin de ce parcours un premier résultat est obtenu (la variable s pour le cas de la figure 3.5) qui correspond au scan au sens de la définition de Blelloch. En combinant alors ce résultat avec le vecteur initial du scan élément-à-élément, on obtient le résultat final (la variable $scan(a,+)$ sur la figure 3.5). La propriété récursive à noter pour cette seconde phase, est que lorsqu'un processeur transmet le produit de sa valeur et celle du fils gauche, au fils droit, ce qu'il transmet en fait est la réduction de tous les éléments qui précèdent ce processeur. Cette seconde phase a également une complexité en $\log(n)$.

Au total cet algorithme a une complexité en $2\log(n)$. Il a été implanté avec les trois options décrites ci-dessous.

3.4.1.1 L'Activité des Processeurs

Il peut arriver que dans un problème donné, au moment de l'appel d'une fonction scan, un processeur de la collection de la variable concerné par ce scan ne soit pas actif, l'algorithme décrit ci-dessus a été implanté en tenant compte de ce cas. La figure 3.6 illustre la manière dont a été prise en compte l'activité des processeurs.

A l'entrée de la procédure tous les processeurs sont rendus actifs, les processeurs initialement inactifs sont affectés de l'élément neutre de l'opérateur $\oplus$ et le scan est effectué normalement. A la fin du traitement l'état des processeurs est restauré et les processeurs non actifs à l'entrée de la procédure retournent une valeur non définie.

3.4.1.2 Le Sens du Scan

La variable d'entrée `sens` de la fonction scan, définit le sens du scan; en effet, pour une application donnée, nous pouvons être ramenés à faire aussi bien un scan de gauche à droite, qu'un scan de droite à gauche, la figure 3.7 illustre la manière dont intervient le sens du scan dans le résultat final.

Le sens du scan intervient au niveau de l'algorithme de la manière suivante :

- **première phase**: pour un scan de gauche à droite (la variable `sens` vaut 1 dans ce cas), l'arbre a la forme indiquée a la figure 3.4 (nous avons dans ce cas un cadrage à droite).

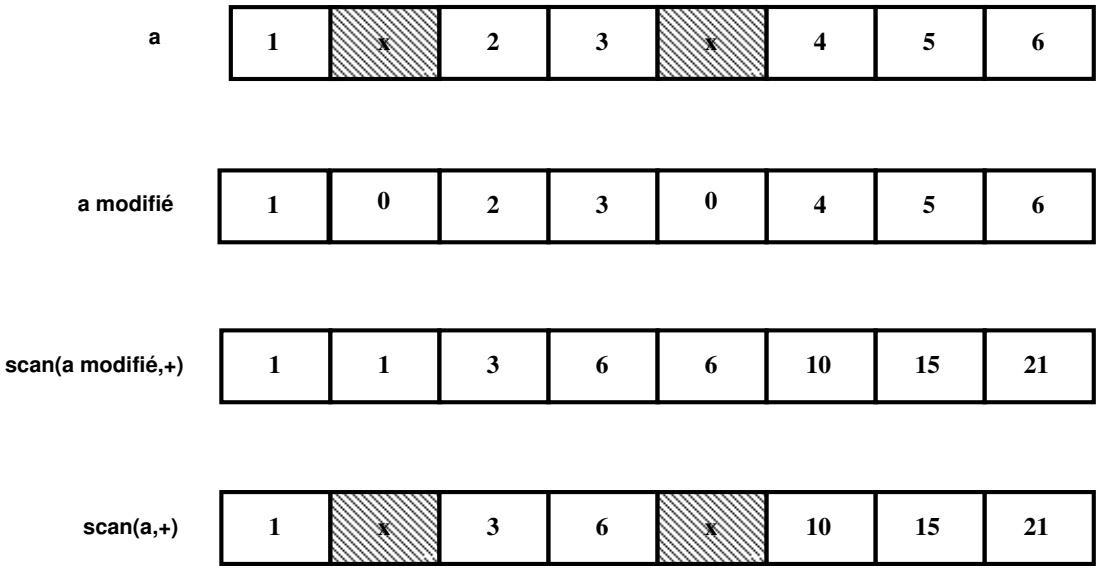

FIG. 3.6 - Un scan avec des processeurs inactifs.

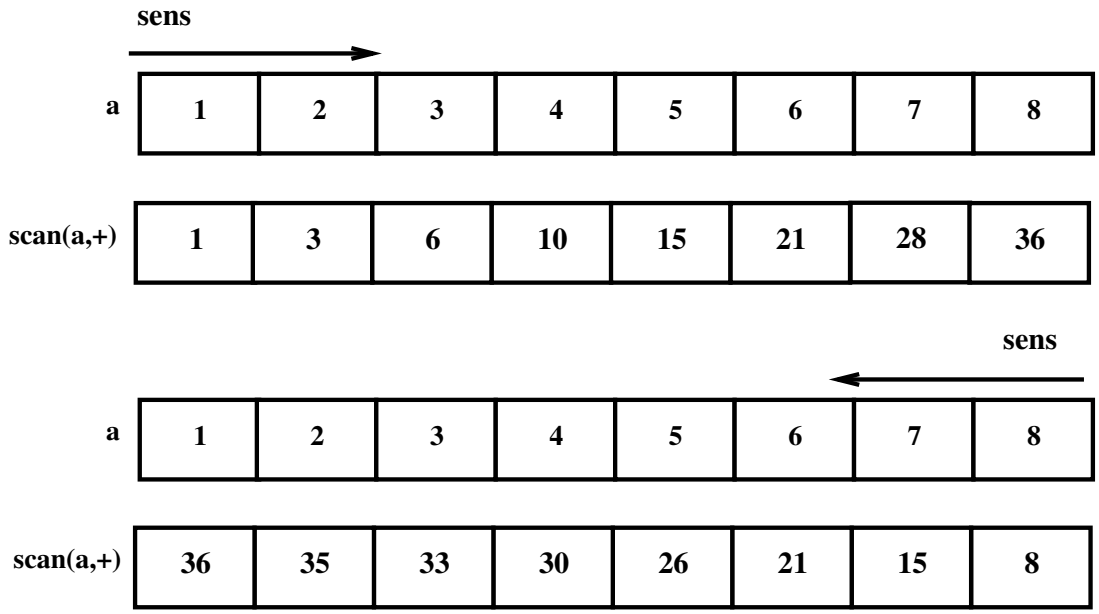

FIG. 3.7 - Le sens du scan.

Pour un scan de droite à gauche (la variable `sens` vaut -1 dans ce cas et le cadrage est à gauche) l'arbre de la phase montante a alors la forme indiquée à la figure 3.8.

- **seconde phase**: en ce qui concerne la phase descendante, le parcours de l'arbre se fait toujours à partir de la racine, mais dans le cas où la variable sens vaut -1, les mouvements de données changent de sens, i.e. nous avons des transferts de données comme sur la figure 3.9.

Au niveau de l'implémentation de l'algorithme, la variable `sens` intervient dans la formule de calcul d'adresse qui permet de sélectionner, à chaque itération, les processeurs adéquats comme schématisé sur les figures 3.8, 3.4 et 3.9.

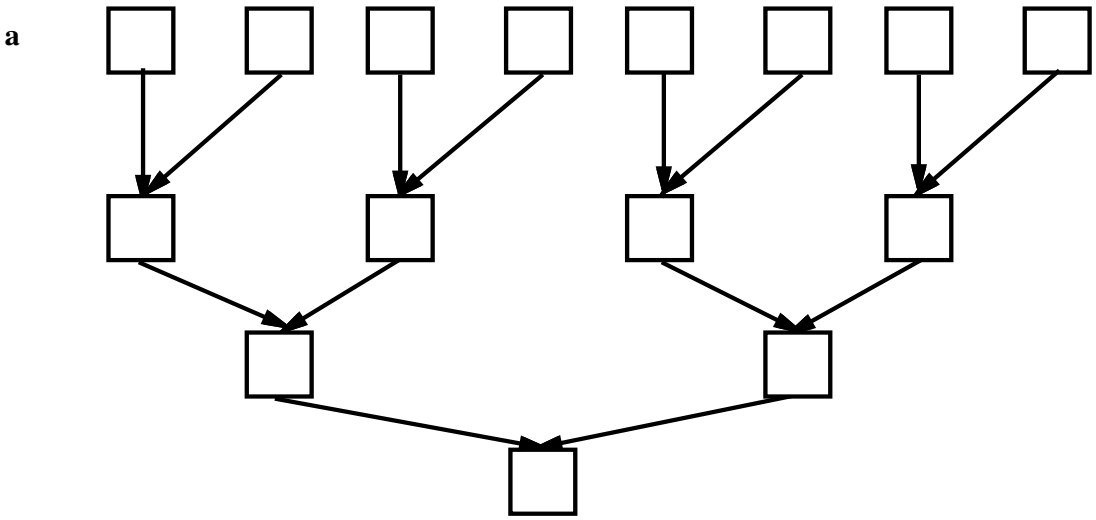

FIG. 3.8 - Phase montante pour un scan de droite à gauche.

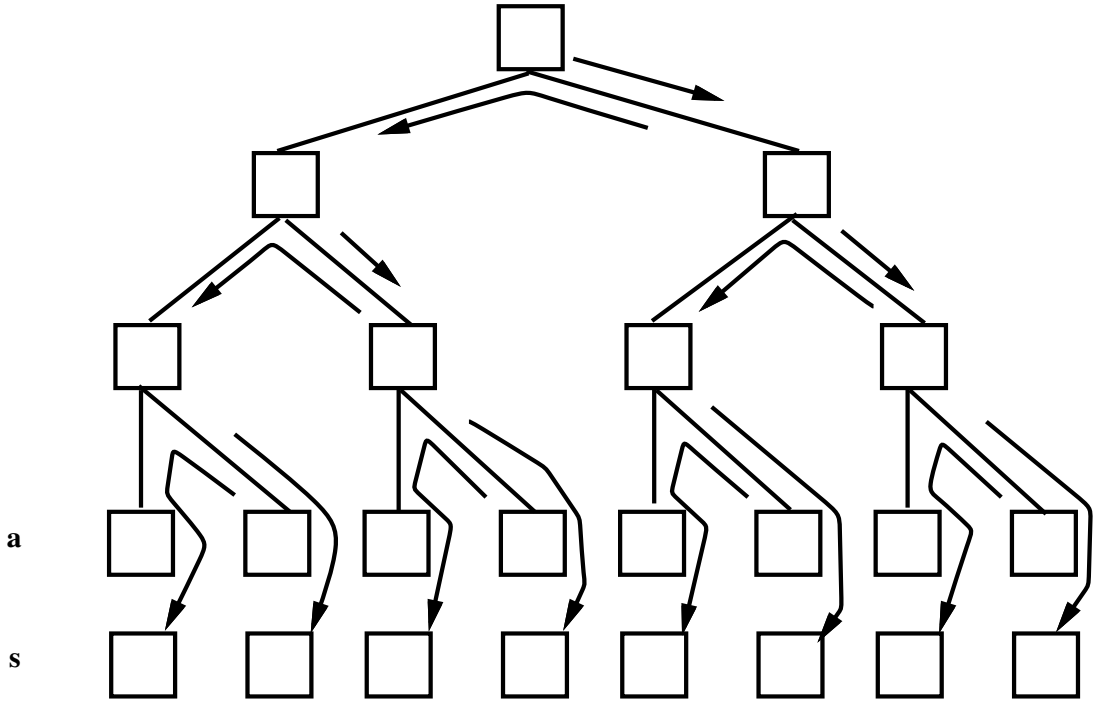

FIG. 3.9 - Mouvements de données pour la phase descendante.

3.4.1.3 Le Scan avec Bit de Start (sbit)

Le bit de start est un vecteur à deux valeurs 0 ou 1, de même dimension que le vecteur d'entrée du scan. il spécifie pour chaque élément de la donnée d'entrée si on démarre un nouveau scan à partir de cet élément ou pas. Par exemple un scan avec le vecteur `sbit` = (0 , 0 , 1 , 0 , 0 , 0 , 1 , 0) comme bit de start se présente comme sur la figure 3.10.

Les modifications apportées à l'algorithme pour tenir compte du bit de start, se résument

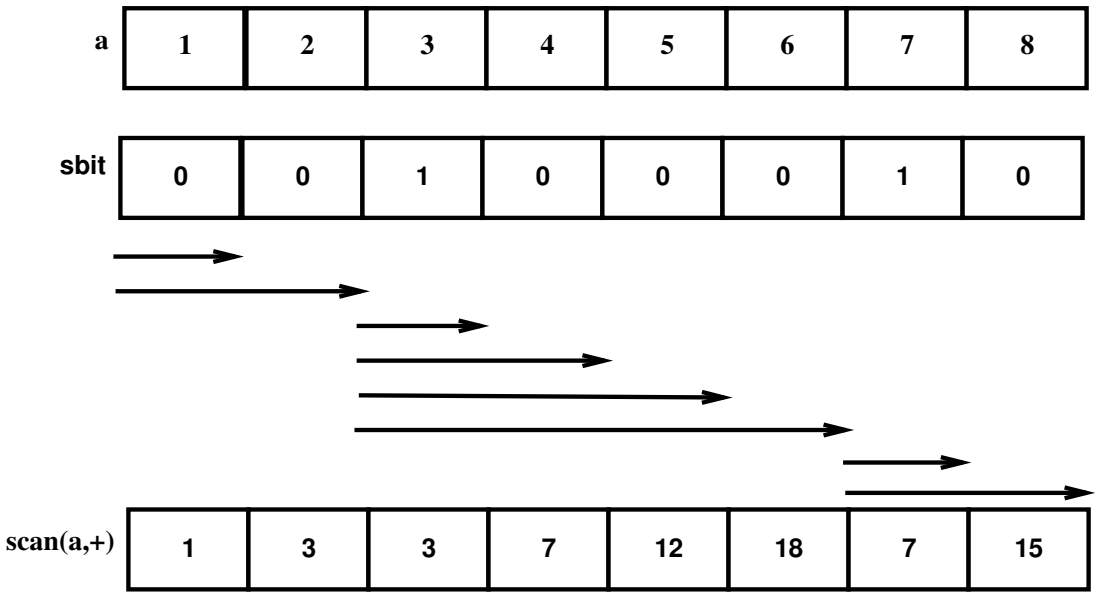

FIG. 3.10 - Le scan avec bit de start.

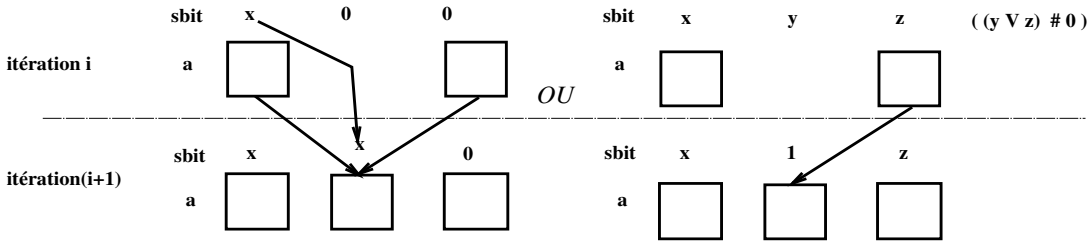

FIG. 3.11 - Modifications pour la phase montante.

en :

- phase montante : la figure 3.11 montre, selon la configuration du bit de start à l'itération i, les mises à jours effectuées aussi bien sur la donnée en entrée, que sur le bit de start pour l'itération $(i+1)$. Une interprétation de cette figure serait :

```
Si la valeur du bit de start du père et du fils droit est 0 à l'itération i
    Alors
      - affecter la réduction des valeurs du fils gauche et droit au père.
      - affecter la valeur du sbit du fils gauche au sbit du père.
    Sinon
      - affecter la valeur du fils droit au père.
      - affecter 1 au sbit du père.
Fsi.
```

- phase descendante : de même pour la phase descendante, les modifications apportées sont illustrées sur la figure 3.12, qui peuvent être interprétées ainsi :

```
Si la valeur du sbit du père est 0 à l'itération i
 Alors
```

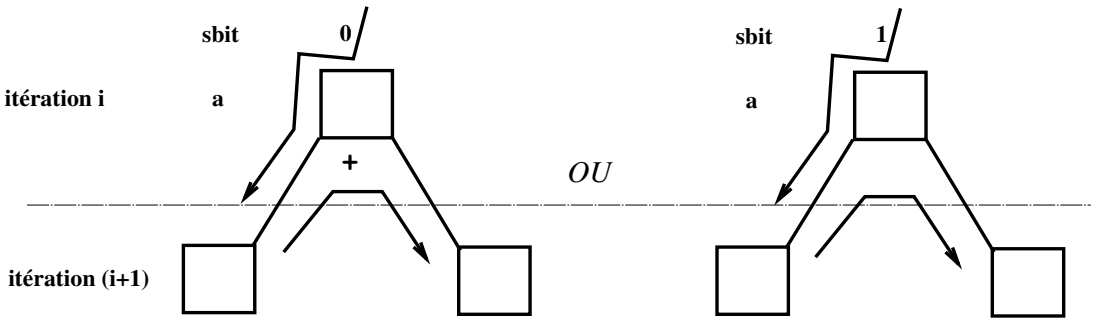

FIG. 3.12 - Modifications pour la phase descendante.

```
  - affecter la réduction des valeurs du fils gauche et pére au fils droit.
  - affecter la valeur du pére au fils gauche.
 Sinon
  - affecter la valeur du fils gauche au fils droit.
  - affecter la valeur du pére au fils gauche.
Fsi.
```

Nous avons commencé l'implantation de la bibliothèque de scan à l'aide de l'algorithme qui vient d'être présenté; mais la lecture du document [HS86] nous a permis la découverte d'un algorithme meilleur (en temps) que nous avons alors adopté pour l'implémentation de cette bibliothèque. Les paragraphes qui suivent décrivent l'algorithme en question et l'implantation des différents paramètres.

Un listing de la première implantation de la fonction scan se trouve en annexe A.1.

3.4.2 Deuxiéme Algorithme

L'algorithme décrit dans [HS86] pour le calcul de toutes les sommes partielles d'un tableau (un scan avec l'addition sur un vecteur de données), présente dans une certaine mesure, de meilleures performances que l'algorithme décrit dans la section précédente, en particulier cet algorithme a une complexité en $\log(n)$. Il est aussi beaucoup plus facilement compréhensible que le premier, et permet ainsi une implémentation plus aisée des différents paramètres d'entrée de la fonction scan (`sens`, `sbit`, etc ...), c'est pour cela qu'il a été adopté en dernier choix pour l'implantation de la bibliothèque de scan.

3.4.2.1 Principe de l'Algorithme

L'algorithme décrit à l'introduction du présent chapitre pour le calcul de la somme de tous les éléments d'un tableau (une réduction), en un temps $o(\log n)$, peut être facilement adapté au calcul du scan avec la même complexité. En effet nous remarquons qu'à l'itération j, seulement $n/2^j$ processeurs sont actifs, et que pour ce cas de calcul d'une réduction, nous avons un arbre binaire, dont les feuilles sont les données d'entrée et la racine est l'élément le plus à droite (élément qui contient le résultat final). Le calcul du scan de la même manière part de la constatation suivante: il suffit, à chaque itération, de superposer autant d'arbres que nécessaire pour le calcul du scan. C'est l'algorithme du *recursive doubling* [LF80]. Le déroulement des opérations a alors la forme présentée sur la figure 3.13.

A chaque itération i ($i = 0..\log n - 1$), pour les processeurs sélectionnés, les sommes entres éléments distants de 2^i sont effectuées. Cet algorithme maintient plus de processeurs actifs que le premier.

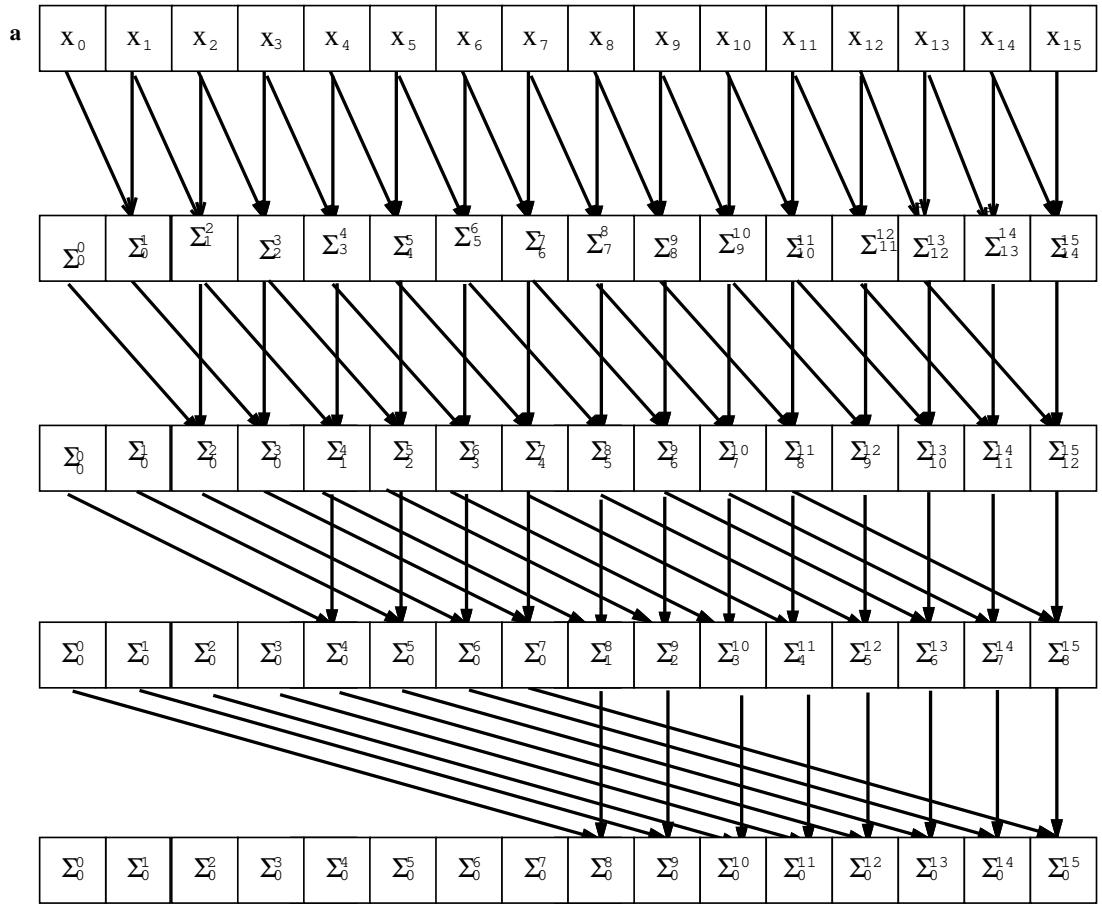

FIG. 3.13 - Algorithme du recursive doubling.

La version finale de la fonction scan a été implantée avec cet algorithme, en tenant compte de l'activité des processeurs avec les quatre paramètres suivants :

- l'axe du scan (pour une collection multi-dimensionnelle),
- le sens du scan,
- le scan avec inclusion/exclusion,
- le bit de start.

3.4.2.1.1 L'Activité des Processeurs Comme pour le cas du premier algorithme, la méthode utilisée pour tenir compte de l'activité des processeurs est de rendre tous les processeurs actifs à l'entrée de la procédure, d'affecter l'élément neutre de l'opérateur $\oplus$ aux processeurs initialement inactifs, d'effectuer le scan et de restaurer l'activité sauvegardée avant la sortie de la fonction. Les processeurs initialement inactifs transmettent alors en sortie une valeur non définie.

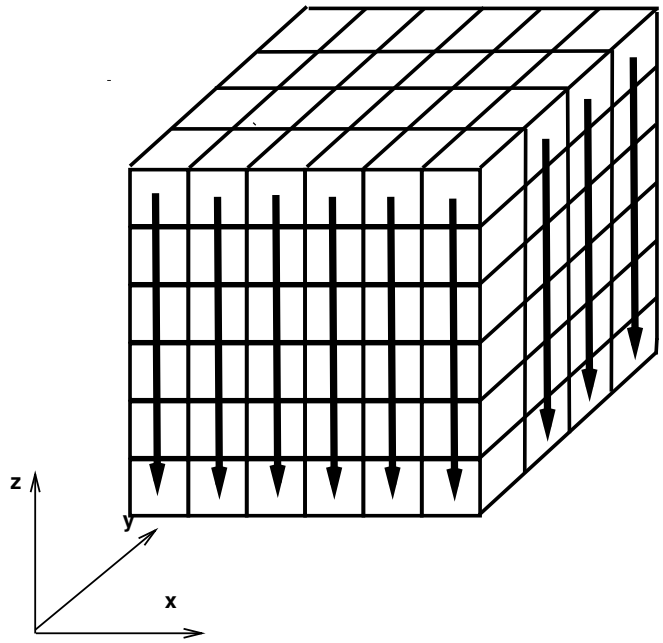

FIG. 3.14 - Axe d'un scan.

3.4.2.1.2 L'Axe du Scan Cette variable spécifie l'axe support du scan. En effet, comme dans le langage POMPC les collections peuvent être de dimensions supérieures à 1, il peut être nécessaire d'effectuer un scan sur une variable dont la collection admet n dimensions, $d_1, d_2, \cdots, d_n$ selon l'une de celles-ci, d_i, nous avons alors à faire à $\Pi_{j \neq i} d_j$, scan en parallèle. La figure 3.14 illustre le cas d'un scan suivant l'axe z, sur une variable parallèle dont la collection est à trois dimensions (en notant que dans ce cas la variable `sens` vaut -1).

Cette variable `axe` est utilisée dans l'algorithme, comme variable d'entrée lors d'appels de primitives de communications de niveau plus bas.

3.4.2.1.3 Le Sens du Scan Une fois l'axe support du scan défini, la variable `sens` spécifie l'orientation du scan :

- 1 pour un scan vers les coordonnées croissantes,
- -1 pour un scan vers les coordonnées décroissantes.

Comme pour le cas du premier algorithme, cette variable est utilisée lors du calcul d'addresse pour la sélection des processeurs à chaque itération.

3.4.2.1.4 Le Scan avec Inclusion/Exclusion La définition du scan avec exclusion est équivalente à la définition donnée par Blelloch. La figure 3.15 montre le cas d'un calcul d'un scan avec exclusion, et le cas de scan avec exclusion et bit de start (3.16). Dans le cas ou cette variable d'entrée vaut 0, c'est un scan avec inclusion et le traitement est effectué normalement.

Au niveau de l'implantation, la méthode utilisée consiste à partir d'une version avec inclusion et à tester en fin d'algorithme si la variable d'entrée inclusion/exclusion est mise à 1, auquel cas, pour les processeurs dont le bit de start est à 0, nous faisons un décalage d'un pas, suivant la valeur de la variable sens, et pour les processeurs dont le bit de start est à 1, nous affectons l'élément neutre de l'opérateur $\oplus$.

3.4.2.1.5 Le Scan avec Bit de Start La définition d'un scan avec bit de start ayant été donnée précédemment, nous discutons dans cette section de l'implantation du bit de start dans

FIG. 3.15 - Scan avec exclusion.

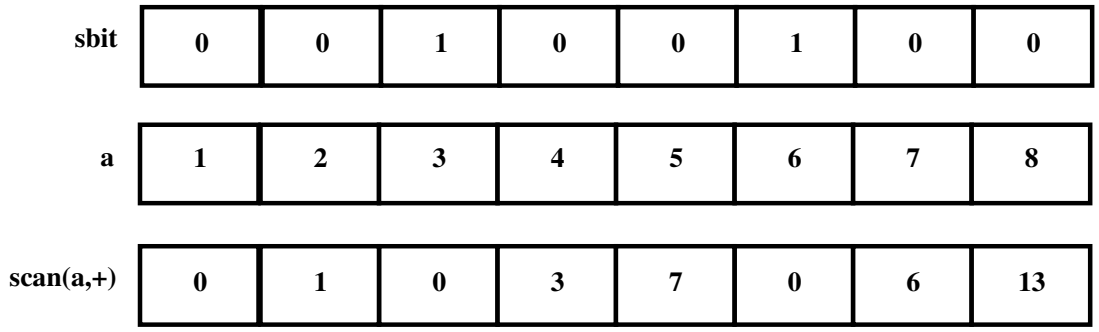

FIG. 3.16 - Scan avec exclusion et bit de start.

le cas du second l'algorithme. La figure 3.17 illustre fort bien la manière du déroulement des calculs pour cet algorithme. A chaque itération (les différents niveaux de la figure 3.17), le bit de start est consulté pour les processeurs selectionnés, si ce bit est à 1, le produit (suivant l'opérateur $\oplus$) avec le processeur correspondant de l'itération actuelle n'est pas effectué, la valeur de ces processeurs reste alors inchangée ; pour les processeurs dont le bit de start est à 0, la réduction est effectuée et le bit de start est affecté de la valeur du bit de start du processeur correspondant ; cela pour en tenir compte à l'itération suivante.

Pour le cas de la figure 3.17, nous remarquons que le résultat est obtenu au bout de la troisième itération au lieu de la quatrième (i.e. $\log 16$), cela est dû au fait que l'on tient compte du bit de start.

Cela achève la première partie de la bibliotèque de scan. Avant de passer au second volet qui traite de la virtualisation, nous faisons une comparaison entre le premier et deuxième algorithme. Nous donnons en annexe A.2, le code source de la fonction scan qui vient d'être décrite.

3.4.3 Comparaison des Deux Algorithmes

Le tableau 3.1 établit une comparaison entre les deux algorithmes décrits précédemment. En première estimation, nous remarquons que le second algorithme possède de meilleures performances que le premier au niveau du temps d'exécution et de l'espace mémoire utilisé. Mais cela est relativisé par l'architecture sur laquelle on implémente ces algorithmes à cause du coût des communications. En particulier, le second algorithme effectue beaucoup de communications régulières mettant en œuvre tous les processeurs, donc son temps d'exécution effectif dépend du coût de ces communications. Il est bien adapté par exemple sur un réseau de type hypercube puisque, si l'on suppose que la taille du vecteur en entrée est 2^D, un plongement existe tel que tous les décalages selon une puissance de 2 se fassent en un temps unité. Ce n'est pas le cas sur un réseau de type grille où le coût des communications dépend de la distance. Le premier algorithme est plus intéressant dans ce cas, sourtout à cause du fait que toute communication de distance 2^d n'utilise qu'un message tous les 2^d processeurs. Chaque PE ne voit alors passer qu'un seul paquet, ce qui permet l'utilisation d'un mode de communication *pipeliné*, tel que la primitive `xnetp` de la machine MasPar [Mas91].

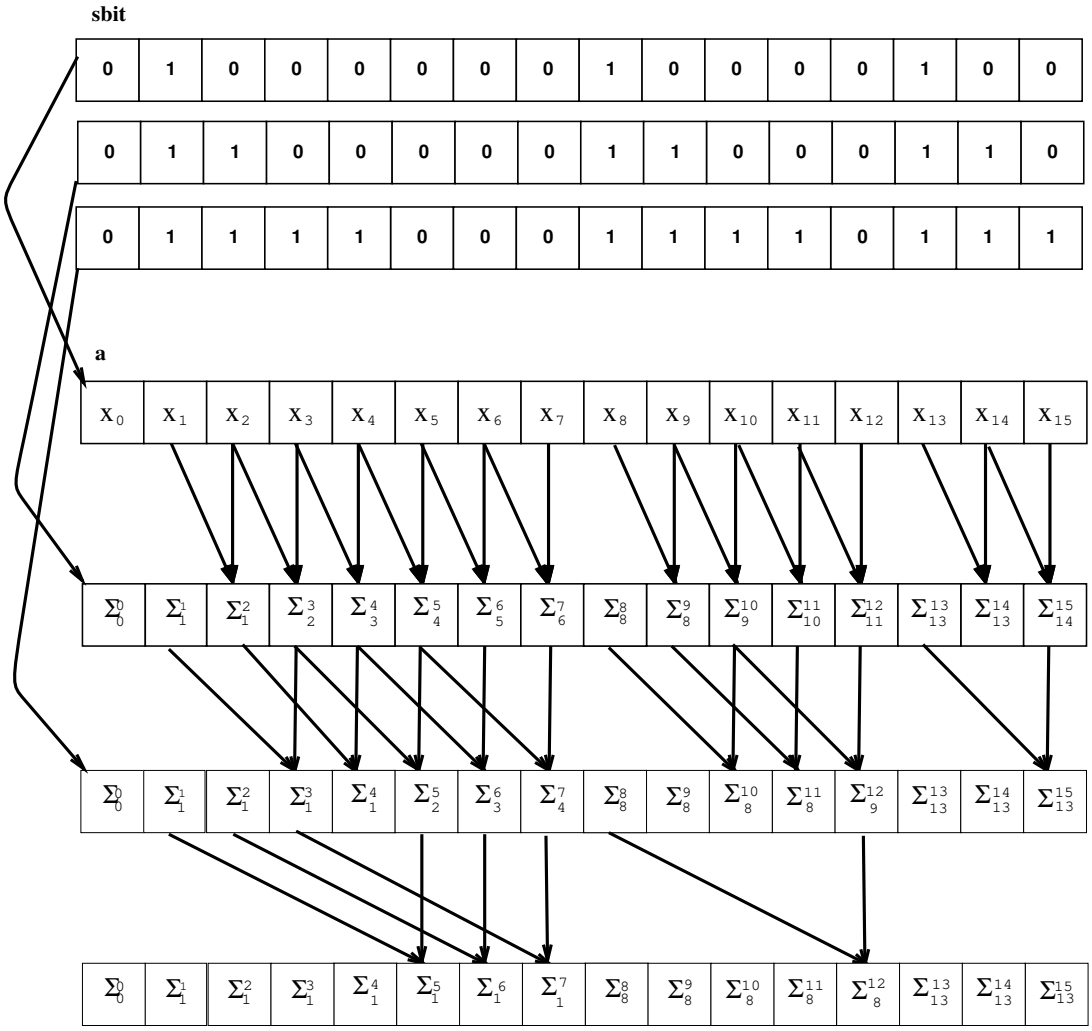

FIG. 3.17 - Traitement du bit de start.

TAB. 3.1 - Une comparaison des deux algorithmes.

Complexité	Premier Algorithme	Deuxième Algorithme
nombre de pas d'exécution	$2\log n$	$\log n$
espace mémoire	$2n$	n
nombre de pas de communication	$2(\log n - 1)$	$\log n - 1$

3.5 La Virtualisation

L'hypothèse admise du modèle de programmation SIMD est que le parallélisme est un parallélisme massif. Puisque celui-ci porte sur les données et non pas sur le flot, son degré est conceptuellemment de l'ordre de la taille des données de l'application. Cette taille de données peut différer d'une application à une autre. La topologie des données parallèles peut aussi différer d'une application à une autre (on peut être amené à travailler sur des tableaux à 1, 2, ...n dimensions). La machine physique sur laquelle ces applications seront implantées possède un nombre de processeurs limité, inférieur à la taille des données; le concept de *virtualisation* apparaît alors comme une solution de recours au disproportionnement qui existe souvent entre

la taille des données et le nombre de processeurs de la machine.

La virtualisation est le mécanisme qui permet d'offrir à l'utilisateur une vue virtuelle de la machine physique sur laquelle il programme son application. On simule pour celui-ci une machine virtuelle dont la taille (en nombre de processeurs) est égale à la taille de ses données. Le programmeur exprime alors son algorithme de manière naturelle et ne s'occupe pas de la gestion de ces données sur les processeurs physiques ; on décharge celui-ci de cette tâche pour la traiter à un autre niveau du cycle du programme. On essaye ainsi de rester dans *l'esprit des langages de haut niveau*. En contre partie, cela implique un certain coût au niveau du processus de compilation.

La gestion de cette virtualisation apparaît alors dans la manière d'allouer et de répartir ces données sur les processeurs physiques de la machine et dans le traitement algorithmique utilisé pour conserver la sémantique du programme initial.

Dans les extensions du langage C pour le parallèlisme de données, existent différents langages qui permettent plus ou moins une virtualisation de domaines :

- MPL (MasPar Parallel Application Language) [Mas91] : le langage n'offre pas de mécanisme de virtualisation, les variables déclarées à l'aide de l'attribut `plural` sont des variables allouées sur chaque PE ; la taille d'une variable parallèle est donc égale au nombre de PE de la machine. Cela entraine des problèmes de portabilité entre machines de tailles différentes.

- MultiC (pour la WaveTracer) [Wav91]: une virtualisation à un seul domaine, la construction `multi perform(size_x,size_y,size_z, sub_program)` définit un sous programme `sub_program` dont toutes les variables parallèles utilisées à l'intérieur de celui-ci seront des variables parallèles à trois dimensions et de taille `size_x` $\times$ `size_y` $\times$ `size_z`; il ne peut y avoir de ce fait de variables parallèles de topologies et de dimensions différentes qui intéragissent à l'intérieur d'une fonction. De plus une variable parallèle ne peut posséder plus de trois dimensions.

- C* (pour la Connection Machine) [Thi90] : offre une possibilité de virtualisation aussi souple que le langage POMPC, les variables parallèles peuvent être de topologies et de dimensions quelconques, le constructeur `shape` permet de définir une topologie et une taille de variables parallèles, c'est l'équivalent de `collection` de POMPC. Ce qu'apporte de plus POMPC par rapport à C*, est la non virtualisation. Elle est nécessaire dès qu'il s'agit de faire de la programmation de bas niveau. En effet, lorsqu'on programme des ressources physiques, comme par exemple des routines de communication ou des fonctions scan, la virtualisation devient un concept aberrant.

3.5.1 La Virtualisation en POMPC

Comme il a été mentionné ci-dessus la nécessité de la virtualisation vient de la taille de la machine qui ne correspond pas à la taille des données parallèles. Aussi, pour le cas du langage POMPC, il peut y avoir plusieurs tailles et plusieurs topologies de variables parallèles dans un programme qui peuvent même interagir dans une même instruction. Pour le programmeur qui programme une application de haut niveau, la virtualisation apporte un élément de confort sans égal. En effet celui-ci écrit son code de manière totalement indépendante de la machine et n'a pas à se préoccuper de détails de portabilité de son programme. Cependant pour la programmation POMPC de bas niveau (programmation système, fonctions scan), la virtualisation devient génante. En effet on travaille sur le domaine physique de la machine, il est alors indispensable de s'affranchir du mécanisme de virtualisation pour pouvoir accéder aux ressources physiques de la machine. La possibilité de non-virtualisation existe en POMPC en déclarant une collection avec l'attribut `physical` ou en projetant une collection virtuelle sur son domaine physique. Toute variable de cette collection est alors une variable physique dont le nombre d'éléments est exactement le nombre de processeurs de la machine cible.

En conclusion, la virtualisation est un concept nécessaire mais pas dans tous les cas. C'est un élément de plus qu'apporte le langage POMPC par rapport aux versions antécédentes des langages C pour le parallélisme de données : la possibilité de ne pas virtualiser.

La virtualisation en POMPC est obtenue en déclarant une variable comme appartenant à une collection donnée, cet attribut fixe alors la machine virtuelle à laquelle la variable appartient, comme il fixe la topologie et le nombre d'éléments de la variable. Tout les traitements effectués sur les variables d'une collection donnée sont alors effectués dans la machine virtuelle associée à cette collection.

La gestion de la virtualisation apparait dans l'allocation des variables parallèles sur les processeurs physiques. Celles-ci sont réparties de manière équitable sur les différents processeurs quelle que soit la topologie de la variable parallèle. On définit le `vp_ratio` pour une collection donnée, comme étant le nombre de processeurs virtuels (N) de cette collection par processeur physique (P).

$$vp\_ratio = \lceil N/P \rceil$$

Les variables sont allouées comme des tableaux de `vp_ratio` éléments sur les processeurs physiques, le mode d'allocation garantit un schémas d'adressage unique sur les différents processeurs physiques. La gestion de la virtualisation apparaît aussi au niveau du code, puisque les éléments d'une variable parallèle deviennent des éléments de tableau qu'il faudra alors énumérer par une boucle pour effectuer les traitements nécessaires.

3.5.2 Algorithme avec Virtualisation

La seconde phase dans l'implantation de la bibliothèque de scan consiste en l'écriture des même fonctions mais en tenant compte de la virtualisation et de la manière de gérer celle-ci ; en effet dans ce cas on suppose que la taille de la variable sur laquelle nous voulons effectuer le scan, n'est pas nécessairement égale au nombre de processeurs physique de la machine, d'où la nécessité de partager un processeur physique par plusieurs éléments de la donnée à traiter. Nous pouvons schématiser de manière générale la gestion de la virtualisation comme indiqué sur la figure 3.18.

- La première phase consiste en un précalcul local sans communication et établit la contribution locale de chaque processeur physique.
- La seconde phase calcule le scan physique, sur le réseau de communication.
- La dernière phase retranscrit le résultat de la phase précédente en le résultat final, cette phase étant locale à chaque processeurs.

En particulier, au niveau de la seconde phase, il est fait appel à la fonction scan de la première étape, c'est à dire la fonction scan physique.

Sur un réseau à p processeurs physiques, pour une variable d'entrée de dimension n, la première et la dernière étapes ont une complexité en $o(n/p)$, alors que le scan physique est en $o(\log p)$ d'où la complexité totale de la méthode $o(2(n/p) + \log p)$.

La figure 3.19 illustre le cas d'une gestion de virtualisation de variable parallèle sur un réseau de processeurs sous forme d'une grille à deux dimensions :

- le schéma 1 représente la variable parallèle virtuelle, cette variable est à trois dimensions et admet $38 \times 28 \times 3$ éléments qui sont "*mappés*" en tableaux de $5 \times 6 \times 3$ éléments sur chaque processeur physique.
- le schéma 2 représente la projection physique de la collection. C'est une grille à deux dimensions dont le nombre d'éléments est 8×5 (qui peut éventuellement représenter la topologie de la machine physique).

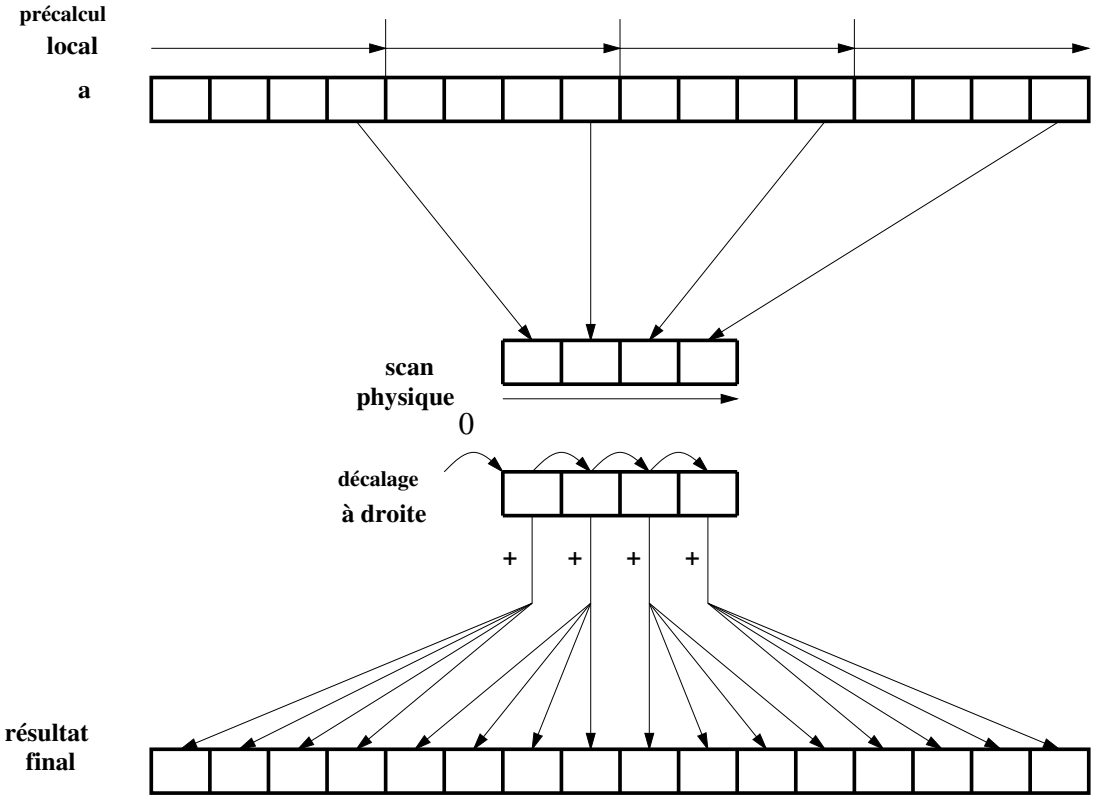

FIG. 3.18 - Gestion de la virtualisation.

- le schéma 3 illustre, dans le cas générale d'une collection à n dimensions, la manière dont sont ordonnés, en POMPC, les différents éléments appartenant à un même processeur physique dans la mémoire de celui-ci.
- le schéma 4 illustre le pavé associé à un processeur physique.

Pour ne pas alourdir l'exposé de l'implantation de la fonction scan tenant compte du mécanisme de virtualisation, nous ne discutons pas l'implémentation des différents paramètres d'entrée de la fonction; nous donnons juste les grandes lignes de l'algorithme, du fait que l'implantation de ces paramètres n'est pas le point important pour cette version et ne change d'ailleurs pas le «squelette» de l'algorithme.

Par rapport à la figure 3.18, le problème de la gestion de la virtualisation apparaît lorsque la collection de la variable d'entrée du scan n'est pas à une dimension et de manière générale, elle est de dimension n.

On suppose donc une collection à n dimensions et un scan suivant l'une de ces dimensions, soit j. Le pavé associé à la collection est alors paratagé en un *hyperplan*, regroupant l'ensemble des dimensions différentes de j et la dimension j. D'autre part, la projection physique de la collection virtuelle permet d'associer à chaque processeur physique un sous-pavé de même dimension que le pavé initial (schéma 4 de la figure 3.19) dont le nombre d'éléments est : $\prod_{i=1}^{i=n} a_i$, chaque a_i correspondant à la dimension i. Ces éléments étant rangés dans la mémoire du processeur selon le schéma 3 de la figure 3.19. Chaque processeur physique exécute $\prod_{i \neq j} a_i$ scans de taille a_j, de manière séquentielle.

Le problème est alors, en premier lieu, dans l'énumération des éléments du scan et en second lieu dans l'énumération des différents scans. En d'autre termes, il faut calculer le «*pas du scan*» étant donné le mode d'arrangement du schéma 3 de la figure 3.19 et le «*pas de la boucle*» qui permet l'énumeration des scans.

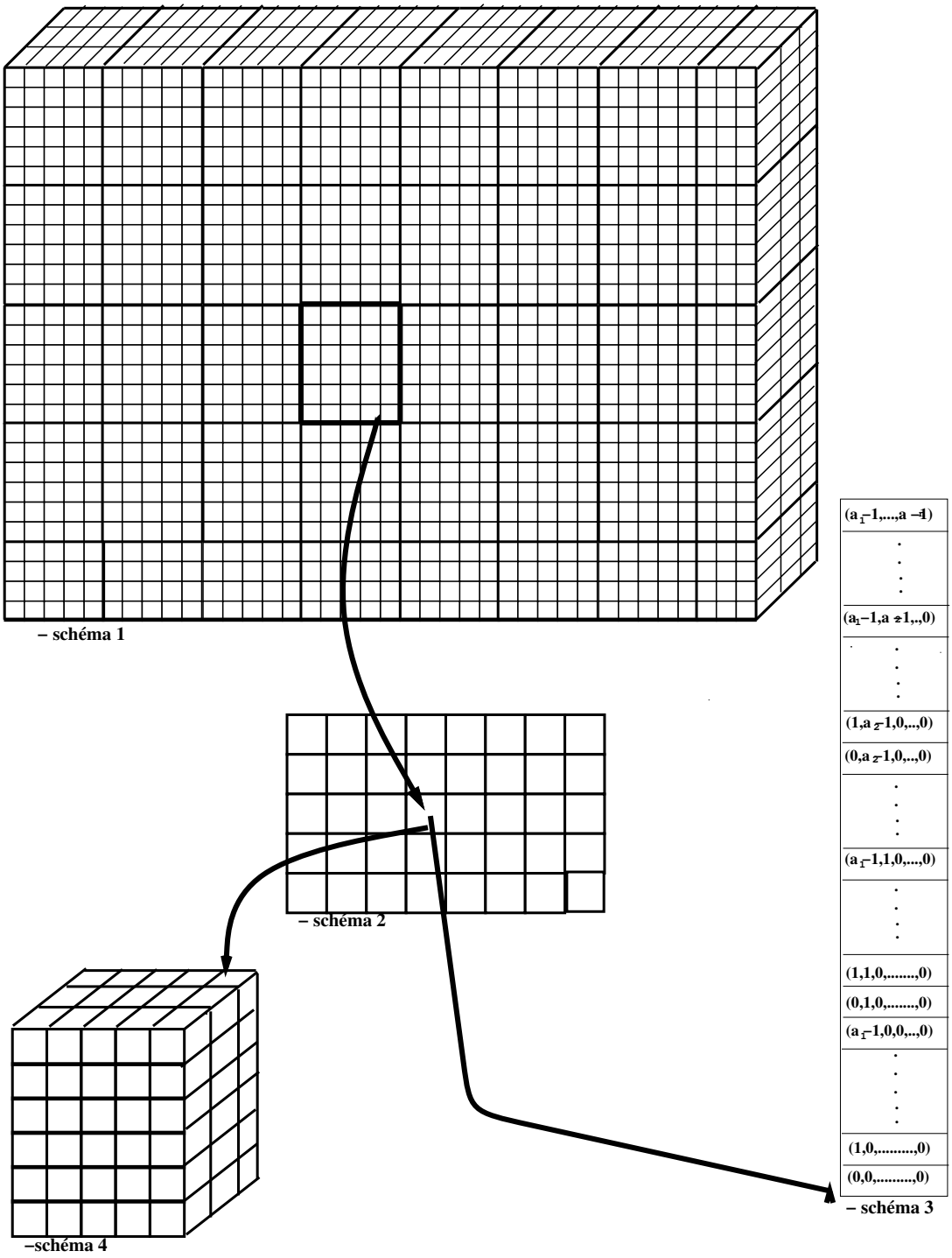

FIG. 3.19 - Virtualisation d'une variable parallèle.

Le *pas du scan* est le produit des a_i des dimensions inférieures à l'axe du scan, i.e. $\prod_{i=0}^{i<j} a_i$. L'algorithme utilise deux boucles extérieures pour l'énumération des scans; respectivement, une boucle pour les dimensions inférieures à l'axe du scan dont le pas est de 1 et une boucle pour les dimensions supérieures dont le pas est le produit de toutes les premières dimensions jusqu'à l'axe du scan. Le corps principal des deux boucles réalise les trois phases de la figure 3.18, en

respectant le pas du scan et faisant un appel à la fonction scan physique.

Un listing pour la version du scan qui tient compte de la virtualisation est donné en annexe A.3.

La méthode adoptée pour le développement de ces fonctions scan était, à chaque fois, de travailler avec une fonction scan pour l'opérateur plus avec le type entier et de dériver ensuite un scan générique suivant le type et l'opérateur grâce à un système de macro-expansion.

Chapitre 4

Interaction Entre le Domaine Virtuel et le Domaine Physique

4.1 Introduction

Nous discutons dans ce chapitre des problèmes sémantiques liés à l'interaction lors d'expressions/affectations réparties sur plusieurs instructions mêlant variables physiques et variables virtuelles; qui entraine ensuite d'autres problèmes de portée de *boucle de virtualisation*. En effet pour la raison de différence entre la taille d'une variable physique et d'une variable virtuelle, l'affectation du premier type de variable par une variable du second type ne peut se faire de manière bijective: élément à élément. Pourtant ce type d'affectation est nécessaire dans certain cas pour des raisons de performances.

Pour cela commencons d'abord par préciser le contexte.

4.1.1 Collections Physiques et Projection d'une Collection

Soit la suite de declaration suivante:

```
collection pixel [100,100];
collection physical processor;

pixel int virtuelle;
physical pixel int projection_pixel;
processor int processeur;
```

La collection `pixel` est une collection virtuelle. La collection `processor` est physique. La variable `virtuelle` est une variable parallèle virtuelle. La variable `projection_pixel` est physique appartenant à la projection physique de la collection `pixel`. Enfin, la variable `processeur` est physique appartenant à une collection physique.

Au niveau allocation, la variable `virtuelle` est allouée comme un tableau de `vp_ratio` (de la collection `pixel`) éléments sur chaque processeur physique. Alors que les deux variables `processeur` et `projection_pixel` sont allouées en un élément sur chaque processeur physique, éventuellement dans des registres.

En ce qui concerne les traitements, toute expression utilisant la donnée `virtuelle` se trouve «plongée» (au niveau du code géneré) dans une *boucle de virtualisation*, qui énumère les différents processeurs virtuels associés à un processeur physique. Alors que toute expression contenant les variables `projection_pixel` ou `processeur` ne doit pas l'être, puisqu'elles possédent un éléments par processeur physique. Donc l'exécution d'une instruction contenant l'une ou l'autre des deux variables s'effectue en une seule étape.

4.1.2 La Boucle de Virtualisation

Toute instruction, dans un programme POMPC, contenant une variable virtuelle est incluse, au niveau du code géneré, dans un boucle qui permet l'énumération des différents PV associés à chaque PP. Cette boucle est référencée par le terme de *boucle de virtualisation*. Elle permet la gestion de la virtualisation, du point de vue traitement algorithmique. Une boucle de virtualisation concerne toujours une seule collection. Son nombre d'itérations est le `vp_ratio` de la collection en question.

Cette boucle de virtualisation représente un certain surcoût qu'il est important de pouvoir minimiser. Cette minimisation est obtenue en rassemblant le plus d'instructions consécutives dans la même boucle. Si cet élargissement atteint un bloc complet (délimité par des accolades), alors toutes les variables parallèles de ce bloc ne sont allouées, utilisées et libérées que pour l'indice courant du PV géré lors de l'itération courante de la boucle de virtualisation. Ces variables locales peuvent être allouées avec uniquement un seul élément par processeur physique: elles peuvent être allouées dans des registres de chaque processeur physique. On obtient, de ce fait, une réduction de ressource mémoire nécessaire ainsi qu'un gain en temps d'accés à ces variables. L'élargissement, au maximum, d'une boucle de virtualisation est donc un but à atteindre au niveau du compilateur POMPC. Cependant, il est limité par les restrictions suivantes :

- 1. une boucle de virtualisation ne concerne qu'une seule collection;
- 2. elle ne contient pas de communication;
- 3. elle ne contient pas d'affectation de variables scalaires;
- 4. elle ne contient pas d'appels de fonction susceptibles de cacher les restrictions précédentes.

Il est à noter que l'élargissement d'une boucle de virtualisation n'est qu'une optimisation du code généré; en aucun cas cet l'élargissement ne doit changer la sémantique d'un programme.

4.2 Interaction Entre Variables Physiques et Virtuelles

L'interaction entre les domaines physiques et virtuels d'une collection est nécessaire pour des raisons de performance. Des exemples d'occurrences d'une telle interaction sont :

- l'écriture de drivers de communication (pour le portage de POMPC);
- l'écriture de programmes en partie virtualisés (fonctions scans).

Une interaction de manière générale se réduit au schéma suivant :

```
variable = expression;
```

Quatre cas sont à distingués pour le sujet de notre discussion; suivant que les membres gauches et droits de l'affectation sont virtuels ou non. Deux cas sont triviaux et ne sont pas interessants. Lorsque le membre gauche est virtuel et l'expression est physique, celle-ci est promue en expression virtuelle en diffusant la valeur de chaque PP vers ses PV correspondants. L'exemple qui suit montre un enploi trés intéressant d'un tel cas :

```
physical pixel int lookup[256];
pixel unsigned char p;
pixel unsigned int rgb;

...
rgb = lookup[p];
...
```

La variable `lookup` est un tableau de variables parallèles physiques. On peut ainsi partager la tabulation d'une fonction entre tous les processeurs virtuels associés à un processeur physique.

Par contre, modifier une variable physique à l'aide d'une expression virtuelle n'a a priori aucun sens et soulève des problèmes d'interprétation : au niveau du code pouvant être généré pour une telle instruction, la partie droite n'a de sens que si l'instruction est «plongée» dans une boucle de virtualisation; alors que la partie gauche ne doit pas l'être. Pourtant l'usage d'une telle affectation est nécessaire pour des raisons de performances :

Supposons la réduction additive,

```
s +<- data;
```

où `data` est une variable parallèle virtuelle et `s` une variable scalaire (l'instruction récupère dans la variable `s` la réduction additive de tous les éléments de la variable `data`).

Une implantation possible de cette réduction serait (en faisant abstraction des problèmes d'activité des PV) :

```
addr = 0;
for(n=pc_sizeof(pixel);n!=0;n--){
s += [addr++]data;
}
```

`pixel` étant la collection virtuelle de la variable `data`.

Cette implantation a une conplexité en $o(n)$, si l'on suppose que le nombre d'éléments de la collection `pixel` est n. C'est une implantation trivialement inacceptable sur une architecture parallèle puisqu'elle effectue le nombre maximal de communication en série et implique aucune prise en compte du parallélisme offert par la machine.

La meilleur implantation (en temps d'execution) [KRS85] est celle qui implémente le raisonnement suivant :

puisque la variable `data` est répartie sur les différents processeurs physiques, en tableau un de `vp_ratio` éléments ;

- **chaque processeur physique calcule, dans une variable locale physique, la somme de ses éléments locaux de la variable `data`.**
- **une réduction additive sur le réseau physique permettra ensuite de récupèrer le résultat final dans la variable `n`.**

on minimise alors au mieux le nombre de communications.

Il faut un moyen syntaxique pour exprimer cette méthode. Une construction qui permette de récupérer les différentes sommes locales dans une variable parallèle physique. En d'autre termes, on gère explicitement la virtualisation. La communication sur le réseau de processeurs physiques ne pose pas de problème.

L'opérateur `with` de POMPC joue ce rôle de gestion explicite de la virtualisation. La réduction additive peut être compilée comme suit :

```
int global_sum(collection pixel int data)
{
    physical pixel int local_sum = 0;
    with(pixel) local_sum += data;
    return physical_sum(local_sum);
}
```

On constate donc un cas d'affectation d'une variable physique par une expression virtuelle. Cette affectation étant, en dernier recours, plongée dans une boucle de virtualisation avec la différence que l'énumération ne concerne que le membre droit et la variable `local_sum` est allouée dans un registre sur chaque processeur physique et non pas comme un tableau.

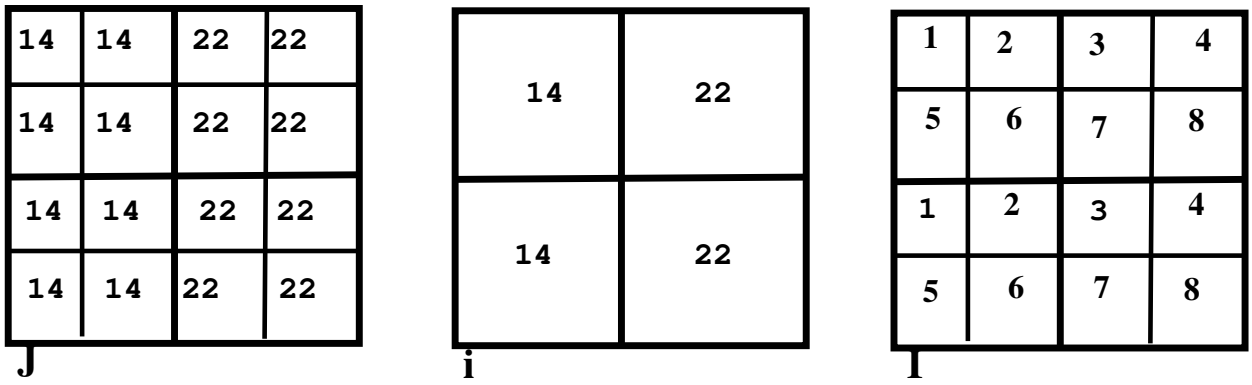

FIG. 4.1 - Résultat du premier programme.

Le constructeur `with` admet donc pour sémantique l'explicitation d'une boucle de virtualisation. Il indique au compilateur l'inclusion explicite d'une instruction dans une boucle de virtualisation. Cependant, une attention particulière est requise quand à l'utilisation de cet opérateur. Considérons le fragment de programme suivant :

```
pixel int I,J;
physical pixel int i;
...
i=0;
with(pixel){ i+=I;}
J=i;
```

La première instruction (`i+=I`) est plongée dans une boucle de virtualisation de manière explicite, à l'aide du constructeur `with`. La seconde instruction (`J=i`) aussi, puisqu'il s'agit d'une affectation d'une variable virtuelle par une variable physique.

Ce fragment de programme n'est pas équivalent au fragment de programme suivant, qui à première vue semble admettre la même sémantique :

```
pixel int I,J;
physical pixel int i;
...
i=0;
with(pixel){
i+=I;
J=i;
}
```

Dans ce dernier exemple l'opérateur `with` porte sur les deux instructions (`i+=I;`) et (`J=i`); les deux instructions sont plongées dans la même boucle de virtualisation.

Le premier exemple accumule dans `i` la somme de tous les éléments de la variable virtuelle `I` localisés sur le même processeur physique et range cette somme dans tous les éléments locaux de la variable virtuelle `J`. Alors que le second accumule toujours dans `i` la somme de tous les éléments de la variable virtuelle `I` localisés sur le méme processeur physique mais range les différentes accumulations successives dans les processeurs virtuels correspondants de la variable J. la figure 4.1 montre le résultat final pour le cas du premier exemple. la figure 4.2 montre le résultat du second programme.

Nous remarquons bien que la valeur de la variable `J` n'est pas la même dans les deux exemples.

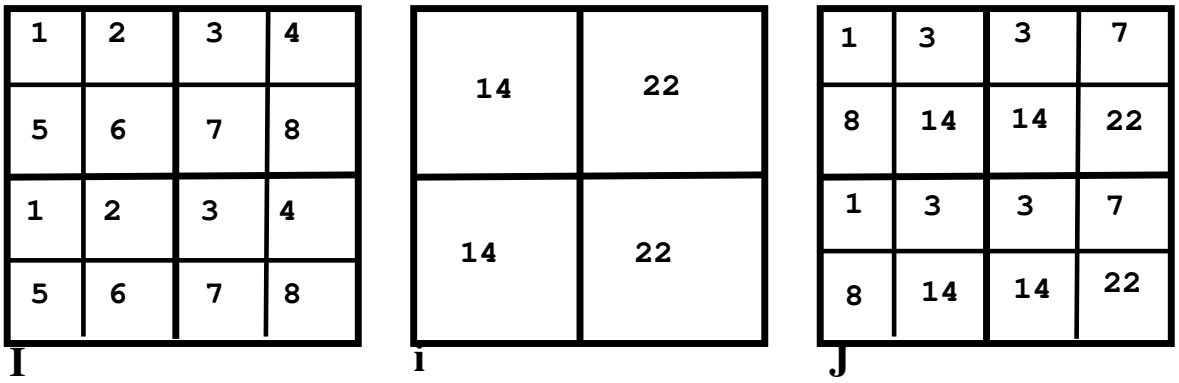

FIG. 4.2 - Résultat du second programme.

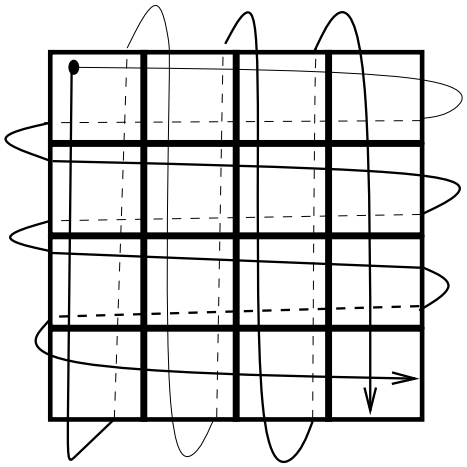

FIG. 4.3 - Énumerations des PV associés à un PP.

La différence entre les deux programmes est la portée de l'opérateur `with`. Dans le premier exemple, l'opérateur porte sur une seule instruction alors que dans le second il porte sur deux instructions. On en déduit donc que pour cette opérateur la spécification de la portée (les deux accolades), dans un programme, n'est pas sans importance. Elle requiert une attention particulière du programmeur.

4.2.1 Le Problème de l'Opérateur `with`

La construction,

```
with(pixel){ i+=I;}
```

ne permet pas d'exprimer explicitement la manière dont sont énumerés les différents PV associés à un PP dans une boucle de virtualisation. En effet pour une collection virtuelle donnée, nous pouvons avoir différentes manières d'énumeration des PVs associés à un PP; la figure 4.3 illustre le cas.

La figure montre le cas d'une collection virtuelle dont la projection physique donne un `vp_ratio` de 16 processeurs virtuels par processeur physique. Les deux lignes en trait fin montrent deux énumerations possible. Pour le type d'application nécessitant des interactions entre le domaine physique projeté et le domaine virtuel d'une collection, il peut être important d'avoir une construction qui permette de contrôller parfaitement cette énumeration.

4.2.2 La Construction Adoptée

La construction ayant été adoptée essaye de remédier au probléme évoqué. Elle est inspirée de la fonction scan qui tient compte de la virtualisation. En effet cette fonction illustre fort bien un cas d'interaction entre le domaine virtuel et le domaine physique d'une collection.

Si l'on essaye de voir de manière générale le code de cette fonction, nous pouvons distinguer un pseudo-organigramme comme suit :

```
1. boucle qui énumère les dimensions supérieures à l'axe du scan.
2. {
3.  boucle qui énumère les dimensions inférieures à l'axe du scan.
4.  {
5.   sauvegarde de l'adresse de début du scan.
6.   boucle qui énumère les éléments du scan.
7.   {
8.    traitement spécifique au scan.
9.    passage à l'élément suivant.
10.  }
11.  scan physique.
12.  restauration de l'adresse de début du scan.
13.  boucle qui énumère les éléments du scan.
14.  {
15.   traitement spécifique au scan.
16.   passage à l'élément suivant.
17.  }
18.  passage au prochain scan (dans les dimensions inférieures à l'axe du
     scan)
19. }
20. passage au prochain scan (dans les dimensions supérieures).
21.}
```

On constate un traitement relatif à `l'hyperplan` de l'axe du scan (les cinq premieres lignes et les quatre dernières), un traitement relatif à l'axe du scan (les lignes 6,7,9,10,13,14,15 et 16) et un traitement spécifique au scan (la réduction).

La solution adoptée est constituée de quatre constructeurs qui simulent les lignes du pseudo-organigramme ci-dessus qui n'effectue pas de traitement spécifique au scan. A chaque opérateur a été associé une partie du code à l'aide de la macro substitution de C (qui est aussi valable en POMPC) `#define`. Ce sont en quelques sortes des opérateurs d'abstraction de boucles énumératives.

- l'opérateur `open_hyperplan`: il simule les cinq premières lignes du pseudo-organigramme ci-dessus. Il prend comme argument une collection et un axe de cette collection.

  syntaxe : `open_hyperplan(collec,axis)`

- l'opérateur `close_hyperplan`: il regroupe les quatre dernières lignes. De même, il admet pour argument une collection et un axe de cette collection.

  syntaxe : `close_hyperplan(collec,axis)`

L'union de ces deux opérateurs est sémantiquement équivalente à l'imbrication des deux boucles externes avec leurs pas associés (lignes 18 et 29). On ne peut avoir, dans un programme, l'occurrence de l'un sans que l'autre y figure; puisque dans le cas contraire, cela «amputerait» la partie supérieure ou inférieure des deux boucles. L'encadrement d'un corps de programme par ces deux opérateurs,

```
open_hyperplan(collec,axe)
    corps de programme;
close_hyperplan(collec,axe)
```

permet dans ce corps de programme l'énumération des dimensions de la collection `collec` autres que la dimension `axe`.

Les deux autres opérateurs sont :

- l'opérateur `open_along` : il est équivalent au regroupement des deux lignes 6 et 7, ou encore des lignes 13 et 14. Il admet pour argument un axe, un sens et une variable adresse qui indique le numéro du PV courant.

  syntaxe : `open_along(axis,sens,adr)`

- l'opérateur `close_along` : c'est l'equivalent des lignes 9 et 10 ou 16 et 17. Comme pour le cas du précédent opérateur, il posséde trois arguments: un axe, un sens et une variable adresse.

  syntaxe : `close_along(axis,sens,adr)`

De même, ces deux opérateurs doivent toujours apparaîtrent ensemble. Ils réalisent la même fonction que la boucle qui énumere les éléments du scan. De manière plus précise, l'encadrement d'un corps de programme par ces deux opérateurs,

```
open_along(axis,sens,sdr)
    corps de programme;
close_along(axis,sens,adr)
```

permet l'énumeration des éléments de l'axe `axis` suivant le sens `sens`; en retournant les résultats successifs de cette énumeration dans la variable `adr` à chaque fois.

La combinaison de ces quatres opérateurs permet d'expliciter la manière dont sont énumerés les PV associés à un PP pour une collection donnée. Puisque le choix de l'axe `axis` est du ressort du programmeur, ainsi que le sens.

Le corps principal de la fonction scan qui gère la virtualisation peut être écrit à l'aide de ces quatres opérateur comme suit :

```
        int global_sum(collection pixel int data)
{
            physical pixel int local_sum, *pointer ;
            int adr;

            pointer = (physical pixel int *) &data;
            local_sum = 0;
            open_hyperplan(pixel,0)
               open_along(0,1,adr)
                  local_sum += pointer[adr];
               close_along(0,1,adr)
            close_hyperplan(pixel,0)
            return physical_sum(local_sum);
        }
```

Nous donnons en annexe A.4 le code associé à chaque opérateur.

4.2.3 Les Régles d'Utilisation

Les quatre opérateurs qui viennent d'être définis ne peuvent figurer dans un programme de manière arbitraire. Ils doivent respecter un certain nombres de contraintes que nous énumerons ci-dessous :

- 1. Chaque opérateur doit figurer avec son opérateur correspondant;

- 2. Les opérateurs `open_hyperplan` et `open_along` précédent au niveau du code, leurs opérateurs associés;

- 3. Toute occurrence du couple (`open_along`,`close_along`) doit être encadrée par une occurrence du couple (`open_hyperplan`,`close_hyperplan`). L'inverse étant faux; le dernier couple d'opérateur peut apparaitre seul;
- 4. Les opérateurs doivent se rapporter à une même collection;
- 5. L'argument `axis` est un nombre compris entre 0 et $n-1$, si n est le nombre de dimensions de la collection `collection`.

L'inclusion de ces opérateurs dans le compilateur POMPC ne cause pas de grand problèmes. En utilisant *YACC* il suffit de :

1. rajouter dans la grammaire les régles :

```
statement : hyperplan '(' exp ',' exp ')' statement
statement : along '(' exp ',' exp ',' exp ')' statement
```

2. rajouter les régles de vérification de type et d'imbrication.
3. rajouter dans le générateur de code, pour l'opérateur `hyperplan` :

```
case hyperplan :
    out ("open_hyperplan(");
    out_collection(s->collection);
    out(",");
    out_exp(s->exp);
    out(")");
    out_statement(s->statement_fils);
    out("close_hyperplan");
    out_collection(s->collection);
    out(",");
    out_exp(s->exp);
    out(")");
```

De même pour l'opérateur `along`, il faut associer un code semblable.

Chapitre 5

Conclusion

Ce stage de DEA a été pour nous l'occasion de découvrir de nombreux domaines qui nous étaient par encore familiers.

Nous avons découvert le parallélisme de donnée au travers du langage POMPC. Cela a été l'occasion de bien comprendre le modèle de programmation ainsi que les modèles d'excécution SIMD ou SPMD associés. Nous nous sommes familiarisés avec la programmation en POMPC par l'écriture de plusieurs petits programmes en POMPC pour l'étiquetage de composantes connexes selon différentes méthodes. Cela nous a permis de comprendre l'intérêt du scan.

Nous avons étudié le scan d'une part au travers d'une recherche bibliographique et d'autre part au travers de nombreuses implantations en POMPC. Outre les aspects théoriques nous avons été confrontés à des problèmes pratiques. En effet, il n'existe pas de «meilleur» algorithme pour toute architecture massivement parallèle. Nous avons dû prendre en considération des aspects très fins des primitives de communications de machines existantes.

Nous avons été amenés de plus à étudier des aspects fins liés au mécanisme de virtualisation de POMPC. Cela a conduit à une implantation plus performante et plus fine du scan virtualisé, qui utilise la version non-virtualisée du scan.

Nous avons à cette occasion découvert les problèmes liés à l'interaction de variables virtuelles et physiques en POMPC. Nous avons étudier les problèmes de sémantique associés, jusqu'alors résolus par l'incomplète primitive `with`. Au regard de ces problèmes, nous avons proposé une construction qui spécifie finement l'ordre d'énumération des processeurs virtuels. Nous avons étudié les conditions et les règles d'emploi d'une telle construction et proposé une méthode pour l'implanter dans le langage POMPC.

Ce stage, par ses aspects aussi bien théoriques que pratiques, nous a enrichi et nous a ouvert sur le monde passionnant du calcul massivement parallèle, de l'aglorithmique parallèle, du parallélisme de données, de sa compilation et des problèmes de sémantiques rencontrés dans les langages à parallélisme de données. Outre l'enrichissement personnel, le travail effectué à l'occasion de ce stage a permis le développement d'une bibliothèque de 56 fonctions scans qui vient enrichir l'environnement POMPC et qui est maintenant utilisé par la communauté des programmeurs POMPC.

Annexe A

Codes des Scans

A.1 Algorithme à Deux Passes

```
collection generic int scan_int_with_plus(generic int a,int sens, generic int
sbit)
{
        generic int c,r,b,x,savesbit;
        generic int save;
        int i,l;

        everywhere b = 0;
        b = a;
        everywhere {
                pc_save_context(generic,save);
                 pc_clear_context(generic);
        }
        c = b;
        where(save!=0) sbit=0;
        savesbit=sbit;
        i = (generic->full_size);
        for(l=0;i > 1 << l;)l++;
        x=pc_coord(0);
        where(((x&1)==(1+(sens>>1)))&(savesbit==0)){
                savesbit=[[.-sens]]savesbit;
                b=b + [[.-sens]]b;
        }
        for(i=1;i<l;i++)
        {
                where(x%(1<<(i+1))==((1<<i)+(sens>>1))){
                        where((savesbit==0)
                                        & ([[.+(sens*(1<<(i-1)))]]savesbit==0))
                        {
                                b=[[.-(1<<(i-1))]]b + [[.+(1<<(i-1))]]b;
                                savesbit=[[.-(sens*(1<<(i-1)))]]savesbit;
                        }
                        elsewhere
                        {
                                b=[[.+(sens*(1<<(i-1)))]]b;
                                savesbit=1;
                        }
                }
        }
        r = 0;
        for(i=l-1;i>0;i--) {
                where((x & ((1 << i+1)-1)) == (1 << i)-sens*(1<<(i-1))+(sens>>1)){
                        r = [[.+(sens*(1 << i-1))]] r;
                        where(savesbit==1){
                                [[.+(sens*(1 << i))]] r <- b;
                        }
```

```
                elsewhere{
                        [[.+(sens*(1 << i))]] r <- r + b;
                }
        }
}
where((x&1)==(-(sens>>1))){
        where(sbit==0){
                r = [[.+sens]] r;
                where([[.+sens]]sbit==0){
                        [[.+sens]]r <- r + c;
                }
                elsewhere{
                        [[.+sens]]r<-0;
                }
        }
        elsewhere{
                r=0;
                where([[.+sens]]sbit==0){
                        [[.+sens]]r <- c;
                }
                elsewhere{
                        [[.+sens]]r<-0;
                }
        }
}
r = r + c;
everywhere  {
        pc_restore_context(generic,save);
}
return r;
}
```

A.2 Algorithme du Scan Physique

```
collection physical generic type pc_physical_scan_with_/**/opstring
        (generic type a, int axis,int sens, int exclusion,
                generic int sbit)
{
        generic type b,c;
        generic int x,save,sb;
        int i,l,step,threshold;

        everywhere b = init;
        b = a;
        everywhere {
                pc_save_context(generic,save);
                i = 0;
                pc_clear_context(generic);
        }
        where(!pc_is_active(save)) sbit=0;
        i = pc_dimof(generic,axis);
        for(l=0;i > 1 << l;)l++;
        x=pc_coord(axis);
        if(sens == 1) i = 0;
        else i = 1-pc_dimof(generic,axis);
        where(x==i){
                sbit=1;
        }
        x *=sens;
        for(i=0;i<l;i++)
        {
                step = -sens*(1<<i);
                pc_physical_get_axis(&c,&b,axis,step);
                pc_physical_get_axis(&sb,&sbit,axis,step);
                threshold = (1<<i)+((pc_dimof(generic,axis)-1)*(sens>>1));
```

```
                where(x >= threshold & sbit==0){
                        b = c op b;
                        sbit = sb;
                }
        }
        if (exclusion==1){
                pc_physical_get_axis(&c,&b,axis,-sens);
                where(sbit==0){
                        b = c;
                }
                elsewhere{
                        b=init;
                }
        }
        everywhere  {
                i = 0;
                pc_restore_context(generic,save);
        }
        return b;
}
```

A.3 Algorithme du Scan avec Virtualisation

```
collection generic type pc_scan_with_/**/opstring
(generic type a, int axis, int sens, int exclusion, generic int sbit)
{
 physical generic type *A,*R,s1,s;
 physical generic int *S,x,physical_sbit;
 generic type r;
 generic int save;
 int accum, nb_scan1=1, nb_scan2=1, adr, d_b_s=1, taille_l_scan, i, j, k, d_e,
 nb_dim, sauv_adr;

 everywhere{
        pc_save_context(generic,save);
        i = 0;
        pc_clear_context(generic);
 }

 where(!pc_is_active(save)) {
        sbit=0;
        a=init;
 }
 A=(physical generic type *)&a;
 R=(physical generic type *)&r;
 S=(physical generic int *)&sbit;
 x=pc_coord(axis);

 nb_dim=(generic->nb_dims)-1;
 for(j=axis+1;j<=nb_dim;j++){
        nb_scan1 *=generic->geometry[j].ldim;
 }
 for(j=0;j<axis;j++){
        nb_scan2 *=generic->geometry[j].ldim;
 }
 adr=0;
 d_b_s=nb_scan2;
 taille_l_scan=generic->geometry[axis].ldim;
 d_e=nb_scan2*(taille_l_scan-1)+1;

 for (k=1;k<=nb_scan1;k++){
        for (i=1;i<=nb_scan2;i++){
                sauv_adr=adr;
                if (sens==-1) adr+=d_b_s*(taille_l_scan-1);
                physical_sbit = 0;
                s = init;
                if(exclusion == 0) {
                        for(j=0;j<taille_l_scan;j++){
```

```
                        where(S[adr]) s = init;
                        s op= A[adr];
                        R[adr]=s;
                        physical_sbit |= S[adr];
                        adr+=sens*d_b_s;
                    }
                } else {
                    for(j=0;j<taille_l_scan;j++){
                        where(S[adr]) s = init;
                        R[adr]=s;
                        s op= A[adr];
                        physical_sbit |= S[adr];
                        adr+=sens*d_b_s;
                    }
                }
                s=pc_physical_scan_with_/**/opstring(s,axis,sens,0,
                  physical_sbit);
                pc_physical_get_axis(&s1,&s,axis,-sens);
                where(x==(sens>>1)*(1-pc_dimof(physical generic,axis))) s1=init;
                adr = sauv_adr;
                if (sens==-1) adr+=d_b_s*(taille_l_scan-1);
                physical_sbit = 0;
                for(j=1;j<=taille_l_scan;j++){
                    physical_sbit |= S[adr];
                    where (physical_sbit == 0) R[adr]op=s1;
                    adr+=sens*d_b_s;
                }
                adr=sauv_adr+1;
            }
            adr=sauv_adr+d_e;
        }
        everywhere {
            i = 0;
            pc_restore_context(generic,save);
        }
        return r;
    }
```

A.4 Codes Associés aux Opérateurs

```
#define open_along(axis,sens,adr) {if (sens==-1) adr = pc_scan_adr +=pc_scan_d_b_s*
(pc_scan_taille_l_scan-1);
else adr = pc_scan_adr\
                for(pc_scan_j=0;pc_scan_j<pc_scan_taille_l_scan;pc_scan_j++){

#define close_along(axis,sens,adr) adr = pc_scan_adr +=sens*pc_scan_d_b_s; }}

#define open_hyperplan(generic,axis) \
        int pc_scan_nb_scan1=1, pc_scan_nb_scan2=1, pc_scan_d_b_s=1;\
        int pc_scan_taille_l_scan, pc_scan_i, pc_scan_j, pc_scan_k, pc_scan_d_e;\
        int pc_scan_adr,pc_scan_nb_dim, pc_scan_sauv_adr;\
        pc_scan_nb_dim=(generic->nb_dims)-1;\
        for(pc_scan_j=axis+1;pc_scan_j<=pc_scan_nb_dim;pc_scan_j++){\
                pc_scan_nb_scan1 *=generic->geometry[pc_scan_j].ldim;\
        }\
        for(pc_scan_j=0;pc_scan_j<axis;pc_scan_j++){\
                pc_scan_nb_scan2 *=generic->geometry[pc_scan_j].ldim;\
        }\
        pc_scan_adr=0;\
        pc_scan_d_b_s=pc_scan_nb_scan2;\
        pc_scan_taille_l_scan=generic->geometry[axis].ldim;\
        pc_scan_d_e=pc_scan_nb_scan2*(pc_scan_taille_l_scan-1)+1;\
\
        for (pc_scan_k=1;pc_scan_k<=pc_scan_nb_scan1;pc_scan_k++){\
                for (pc_scan_i=1;pc_scan_i<=pc_scan_nb_scan2;pc_scan_i++){\
                        pc_scan_sauv_adr=pc_scan_adr;\
```

```
#define close_hyperplan() pc_scan_adr=pc_scan_sauv_adr+1; } pc_scan_adr=pc_scan_sauv_adr+pc_scan_d_e; }
```

Bibliographie

[AKLGLS88] Eugene Albert, Kathleen Knobe, Joan D. Lucas, et Jr. Guy L. Steele. Compiling Fortran 8x Array Features for the Connection Machine Computer System. *Symposium on Parallel Programming: Experience with Applications, Langages and Systems.ACM SIGPLAN*, pages 42–56, juillet 1988.

[Ber91] Robert Bernecky. Fortran 90 Arrays. *ACM SIGPLAN Notices*, 26(2):83–97, février 1991.

[Ble89a] Guy E. Blelloch. *Scan Primitives and Parallel Vector Models.* PhD thesis, Laboratory for Computer Science — Massachusetts Institute of Technology, octobre 1989. MIT/LCS/TR-463.

[Ble89b] Guy E. Blelloch. Scans as Primitive Parallel Operation. *IEEE Transactions on Computers*, 38(11):1526–1538, novembre 1989.

[Cal91] D. Callahan. Recognizing and Parallelizing Bounded Recurrences. Dans *Fourth Internationnal Workshop on Languages and Compilers for Parallel Computing*, pages 169–185. Springer-Verlag, août 1991. LNCS 589.

[Fly66] Michael J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, décembre 1966.

[HS86] W. Daniel Hillis et Guy L. Steele Jr.. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, décembre 1986.

[Ker92] Ronan Keryell. *POMP: d'un Petit Ordinateur Massivement Parallèle SIMD à Base de Processeurs RISC — Concepts, Etude et Réalisation.* Nouvelle thèse, Laboratoire d'Informatique de l'Ecole Normale Supérieure — Université Paris XI, A soutenir 1992.

[Kog74] P. M. Kogge. Parallel Solution of Recurrence Problems. *IBM Journal of Research and Development*, 18(3):138–148, mars 1974.

[KRS85] Clyde P. Kruskal, Larry Rudolph, et Marc Snyr. The Power of Parallel Prefix. *IEEE Transactions on Computers*, C-34(10):965–968, octobre 1985.

[KS73] Peter M. Kogge et Harold S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22(8):786–792, août 1973.

[LF80] Richard E. Ladner et Michael J. Fisher. Parallel Prefix Computation. *Journal of the ACM*, 27(4):831–838, octobre 1980.

[Mas91] MasPar Computer Corporation. *MasPar Parallel Application Language (MPL) Reference Manual*, document part number: 9302-000, revision: a4 édition, mars 1991. Software Version 2.0.

[Par92a] Nicolas Paris. Définition de POMPC (Version 1.99). Rapport Technique LIENS-92-5, Laboratoire d'Informatique de l'Ecole Normale Supérieure, mars 1992.

[Par92b] Nicolas Paris. POMPC: A C LANGUAGE FOR DATA PARALLELISM. *International Journal of Modern Physics C*, á paraître 1992.

[San91] Jean-Paul Sansonnet. *Concepts d'Architecture Avancées.* Laboratoire de Recherche en Informatique-URA 410 du CNRS Equipe Architecture des Ordinateurs et Conception des Circuits Intégrés, 1990-1991.

[Thi90] Thinking Machine Corporation. *C* Programming Guide*, novembre 1990. Version 6.0.

[Wav91] Wavetracer Inc. *The multiC Programming Language — User Documentation*, pub-00001-001-1.01 édition, septembre 1991.

[WN91] Haigeng Wang et Alexandru Nicolau. Optimal Schedul for Parallel Prefix Computation with Bounded Resources . Dans *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–9, avril 1991.

Table des matières

Table des figures