Définition de POMPC (version 1.99)

Nicolas PARIS

Laboratoire d'Informatique, URA 1327 du CNRS
Département de Mathématiques et d'Informatique
Ecole Normale Supérieure

LIENS - 92 - 5

Mars 1992

Chapitre 1

Le modèle de programmation de POMPC

1.1 Introduction

Le langage **POMPC** est une extension du langage ANSI **C** destinée à programmer explicitement des machines parallèles suivant un modèle de programmation de parallélisme de données.

POMPC est adapté à la programmation de machines massivement parallèles SIMD[1] ou SPMD[2] à mémoire distribuée telles que la machine POMP, les Connection-Machines CM2 et CM5 et la machine MasPar MP-1. On peut envisager aussi la programmation de machines MIMD[3].

POMPC cherche à prolonger la philosophie de **C**:

- programmation impérative,
- manipulation de variables symboliques typées,
- programmation structurée,
- organisation du programme source en modules et en fichiers d'en-tête,
- accès direct possible aux ressources matérielles, pour l'écriture de *drivers* gérant les ressources physiques (réseau de communication, entrées-sorties, ...).

Le langage **POMPC** contient toute la norme ANSI **C**. Sauf collision sur les nouveaux mots-clés de **POMPC**, tout programme ANSI **C** est un programme **POMPC**. Les modules **POMPC** se caractérisent par leur extension `.pc`. Les fichiers d'en-tête continuent à avoir l'extension `.h` par convention.

L'expression du parallélisme est explicite en **POMPC**:

- les variables parallèles sont déclarées parallèles de manière explicite;
- les mouvements de données entre les processeurs élémentaires sont explicites.

POMPC permet d'exprimer le modèle de programmation *data-parallel*. Le parallélisme est exprimé non pas sur le flot d'instructions mais sur les données.

[1] **S**ingle **I**nstruction **M**ultiple **D**ata
[2] **S**ingle **P**rogramme **M**ultiple **D**ata
[3] **M**ultiple **I**nstruction **M**ultiple **D**ata

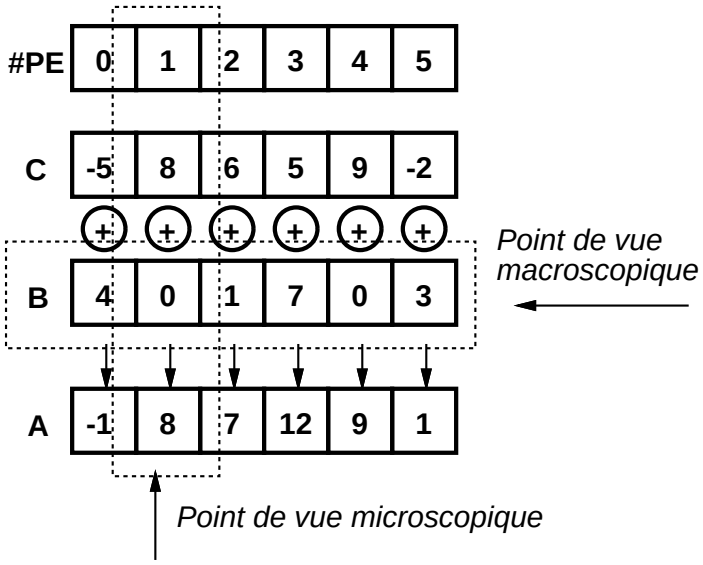

Figure 1.1 - une addition parallèle.

1.2 Programmation synchrone

Chaque processeur exécute le même programme de manière synchrone. Le modèle présente une exécution de type SPMD. Suivant les possibilités de l'architecture cible, l'exécution sera SIMD ou SPMD tout en conservant la sémantique du programme. Du point de vue du programmeur, le modèle de programmation est SIMD, même si l'exécution réelle peut être SPMD. La synchronisation SIMD du modèle de programmation de **POMPC** est assurée par la présence dans le modèle de programmation d'un processeur scalaire qui contrôle le séquencement de la machine.

L'instruction parallèle `A = B + C;` (fig 1.1) réalise l'addition des variables parallèles `A`, `B` et `C`. Chaque processeur physique (en abrégé PP) possède un élément de toutes les variables parallèles et exécute l'addition sur les opérandes qui résident dans sa mémoire locale.

A chaque instruction parallèle, les n PEs exécutent la même instruction sur n éléments de variables parallèles. On peut considérer une instruction parallèle suivant deux points de vue :

- le point de vue microscopique (vertical sur la figure 1.1) : le point de vue de chaque PP avec ses données locales;
- le point de vue macroscopique (horizontal sur la figure) qui considère chaque variable parallèle comme une entité indissociable.

POMPC offre au programmeur le point de vue macroscopique : les variables parallèles sont considérées comme des entités du langage.

Le modèle *data-parallel* est bien adapté à l'exécution de calculs identiques sur tous les éléments d'un tableau. Un tel tableau peut devenir en **POMPC** une variable parallèle. Sa taille est en général une donnée du problème qu'on cherche à résoudre par l'exécution du programme. Elle ne coïncide pas *a priori* avec le nombre de PPs de la machine.

1.3 Les processeurs virtuels (PV)

Le programmeur aimerait faire abstraction de la contrainte du nombre de PPs de la machine cible, car ce n'est pas une donnée de son problème. De plus il serait dommage de ne pas pouvoir

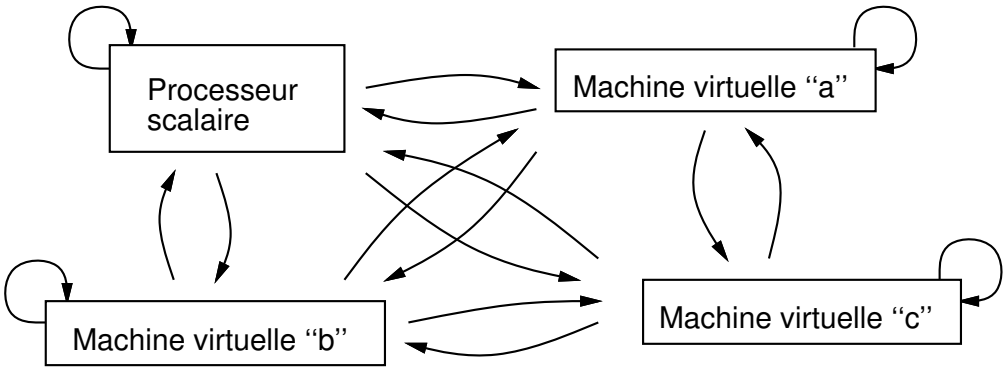

Figure 1.2 - quelques machines virtuelles..

exprimer le même algorithme de la même manière pour des machines de taille différente.

POMPC offre au programmeur un point de vue virtuel de la machine. S'il doit faire des calculs sur des variables parallèles de p éléments alors il dispose d'une machine virtuelle de p processeurs virtuels. Chacun de ces p processeurs virtuels sont distribués équitablement sur les n processeurs physiques.

Si une même application utilise des variables parallèles de tailles différentes, on définit pour elle,autant de machines virtuelles qu'il existe de types différents de variables parallèles.

On découpe ainsi l'ensemble des variables parallèles en classes. Chaque classe contient des variables parallèles ayant le même nombre d'éléments n. Dans le vocabulaire **POMPC**, une classe est une *collection*.

Les n éléments de chaque variable parallèle d'une collection existent pour le traitement en parallèle de n sous-problèmes analogues. Les calculs sur les éléments d'une même classe peuvent être vus comme si on disposait de n processeurs exécutant chacun le même algorithme, pour résoudre n problèmes en parallèle. Ces n processeurs virtuels forment la machine virtuelle associée à la collection.

Les calculs en **POMPC** font interagir des variables appartenant à différents domaines (fig 1.2) :

- les variables scalaires; elles sont manipulées par un processeur spécial appelé processeur scalaire,
- les variables parallèles de chaque collection; elles résident dans la machine virtuelle associée à chaque collection.

Les interactions vont être de deux types :

- directes (entre éléments à l'intérieur d'une même collection),
- indirectes (entre différentes collections ou entre des éléments qui ne sont pas localisés sur les mêmes processeurs virtuels); elles sont vues comme des communications entre différentes machines virtuelles.

1.4 Explicitation des communications

Les machines massivement parallèles présentent jusqu'à maintenant, (probablement pour longtemps) une grande différence entre l'efficacité d'un accès en mémoire locale (interaction directe) et l'efficacité d'une communication pour aller chercher une donnée dans la mémoire d'un autre PV (interaction indirecte). Un à deux ordres de grandeur séparent les durées de

ces 2 accès. Cette disparité impose au programmeur beaucoup d'attention en vue de limiter au minimum le nombre de ces communications si coûteuses. Pour faciliter cette tâche **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 ne sont possibles que par l'utilisation explicite d'opérateurs de communication.

Cette contrainte s'exprime par l'utilisation des collections dans la déclaration des variables parallèles et le typage des expressions parallèles.

Il existe une seule exception à la règle au caractère explicite des communications : la diffusion d'une variable scalaire (normalement réalisée de manière peu coûteuse) dans tous les processeurs physiques n'est pas explicite.

Chapitre 2

Les collections

2.1 Déclaration d'une collection

Définir une collection revient à décrire l'organisation typique d'une variable parallèle de cette classe.

On définit la collection `pixel` par l'instruction :

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

`pixel` devient alors un symbole du langage qui est utilisé :

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

Les variables de cette collection posséderont 1152×900 éléments organisés en une grille de 1152 colonnes et de 900 lignes. De même

```
collection [20,20,20] voxel;
collection [1152] line;
collection vector;
```

définit les trois collections :

- `voxel` dont les variables parallèles sont organisées en une grille cubique de $20 \times 20 \times 20$;
- `line` dont les variables parallèles sont organisées en une ligne de 1152 éléments;
- `vector` dont la taille et l'organisation sont définies dynamiquement.

Les variables parallèles sont déclarées dans les machines virtuelles associées aux collections ou bien dans la mémoire du processeur scalaire (fig. 2.1).

Une variable déclarée comme membre d'une collection est parallèle et est allouée sur tous les processeurs physiques. Une variable déclarée comme n'appartenant pas à une collection est une variable scalaire qui est allouée alors dans la mémoire du processeur scalaire.

```
pixel int n;
pixel double f;
```

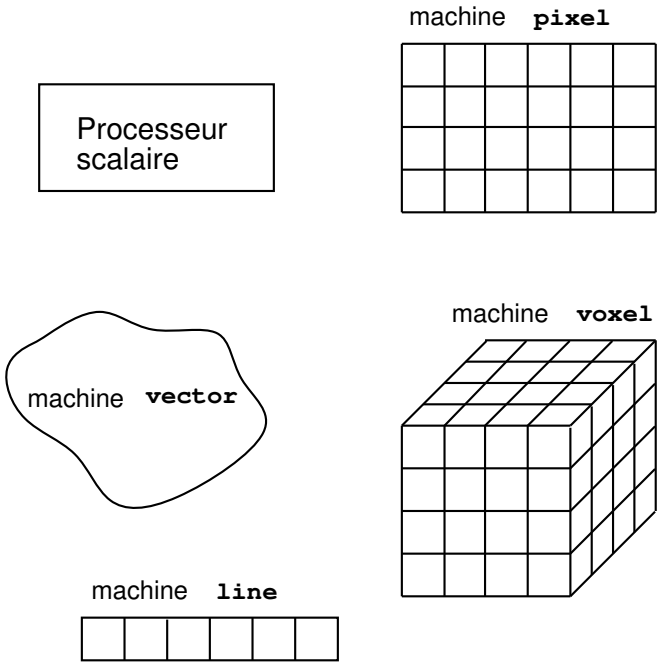

Figure 2.1 - quelques collections....

définit une variable parallèle `n` entière et une variable parallèle `double f` appartenant toutes les deux à la collection `pixel`.

Deux informations essentielles sont attachées à chaque collection :

- la géométrie des variables de la collection. Une variable parallèle est un ensemble de variables de même type distribuées sur les PPs. La taille d'une variable parallèle est son nombre d'éléments, ici $1152 \times 900 = 1036800$. On précise une géométrie : les variables de la collection sont alors organisées suivant un pavé à plusieurs dimensions. Cette organisation est appelée géométrie de la collection. Les variables parallèles d'une même collection partagent non seulement leur taille mais aussi leur géométrie. La collection `pixel` possède une géométrie de 2 dimensions de 1152 par 900. Quelques fonctions permettent d'accéder à la taille et à la géométrie de la collection :
  - `pc_sizeof(pixel)` renvoie la taille de la collection `pixel` (à savoir 1036800 dans notre exemple);
  - `pc_rankof(pixel)` renvoie le nombre de dimensions de la collection `pixel` (ici 2);
  - `pc_dimof(pixel,n)` renvoie la taille de la dimension `n`. La première dimension porte le numéro 0. Ainsi `dimof(pixel, 0)` renvoie 1152 et `dimof (pixel, 1)` renvoie 900.

  La définition d'une géométrie permet la définition de voisinages connus, utiles pour accélérer des communications de voisinages. Cette géométrie intervient également lors de la distribution des variables sur les processeurs physiques.

- une activité parallèle. Chaque processeur virtuel d'une collection peut être actif ou inactif suivant l'élément local de l'activité parallèle. Le constructeur `where`, équivalent au `if` sur une condition parallèle, qui prend en argument un booléen parallèle d'une collection, altère la variable parallèle d'activité de la collection pour la durée d'un bloc. On inhibe ainsi une fraction des processeurs virtuels. Les autres machines virtuelles associées aux autres collections ne voient pas leur activité modifiée et continuent à travailler normalement.

2.2 Collection dynamique

On ne connaît pas nécessairement la taille d'une collection à la compilation. La taille et la forme de la collection est alors une donnée du problème qui peut évoluer au cours de l'exécution du programme. La déclaration de sa géométrie et de sa taille ne se fait alors plus à la compilation mais à l'exécution grâce à un appel de la fonction `pc_start_collection` de la bibliothèque standard (page 57). On aurait pu écrire :

```
collection pixel;
int dim[2];
....
dim[0] = 1152;
dim[1] = 900;
pc_start_collection(pixel,2,dim);
....
pc_kill_collection(pixel);
....
```

Il est bien entendu interdit d'utiliser une collection qui n'a pas encore été démarrée. Une fonction qui démarre une collection ne peut faire des calculs sur des variables de la collection, pour des raisons d'allocation.

La seule manière de changer la taille d'une collection revient à la détruire et à la recrèer. Les variables globales d'une collection dynamique sont allouées au démarrage de la collection et libérées lors de la destruction de la collection.

La déclaration d'une collection n'est légale en **POMPC** que dans le contexte globale : on ne peut déclarer une collection locale à une fonction.

2.3 La déclaration des variables

Lors de la définition d'une variable en **C**, on peut préciser 4 attributs :

1. le type de la variable (`void`, `int`, `char`, `short`, `long`, `float`, `double`, `long double` et `struct/union`),
2. son signe (`signed` et `unsigned`),
3. sa classe (qui précise sa localisation et sa portée) (`static`, `extern`, `auto` et `register`),
4. sa pérennité (`const` et `volatile`).

Ces 4 attributs sont à peu près orthogonaux et possèdent des valeurs par défaut.

Dans **POMPC**, nous allons enrichir la définition de variables parallèles à l'aide d'un $5^{\text{ème}}$ attribut : la collection de la variable. Cet attribut est toujours placé en première position dans la déclaration de la variable. Si aucune collection n'est précisée, il s'agit d'une variable scalaire. Nous pouvons écrire :

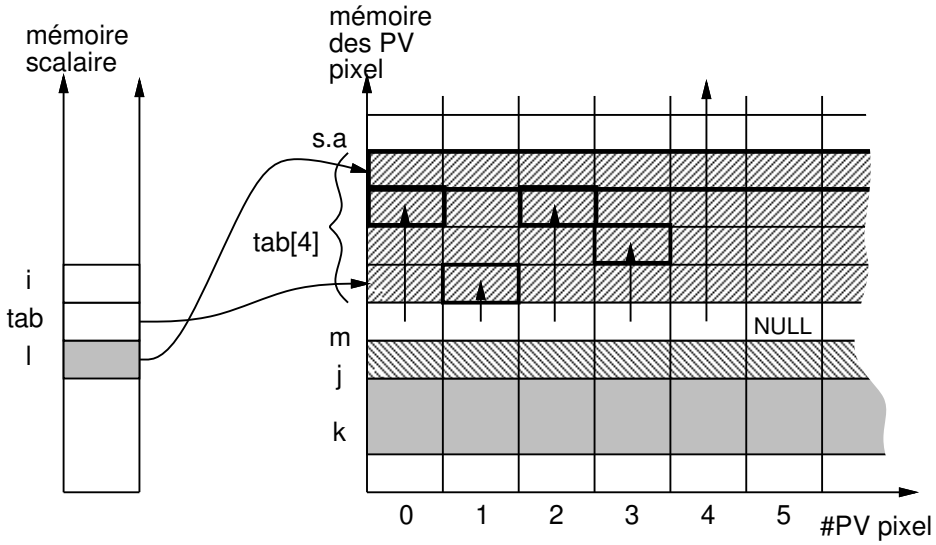

Figure 2.2 - la localisation des variables..

`pixel static double k;`	`k` est une variable statique parallèle formée de réels double précision;
`pixel int j;`	`j` est une variable parallèle d'entiers;
`pixel int unsigned *l;`	`l` est un pointeur scalaire pointant vers une variable parallèle d'entiers non signés de collection `pixel`;
`pixel int * pixel m;`	`m` est une variable parallèle de collection `pixel` constitué de pointeurs sur des éléments de variables parallèles d'entiers de collection `pixel`;
`pixel int tab[4];`	`tab` est un tableau de 4 entiers parallèles;
`pixel struct { int a,b,*c; pixel int *d; } s;`	`s` est une variable parallèle de structures;
`int i;`	`i` est un entier scalaire.

La figure 2.2 montre la localisation des variables entre le domaine scalaire et le domaine virtuel de la collection `pixel`.

2.3.1 Deux nouveaux pointeurs

Deux nouveaux types de pointeurs apparaissent :

- les pointeurs scalaires. Ils correspondent à une adresse d'une variable parallèle (dans la mémoire de chaque PV). Comme l'organisation de l'espace mémoire est la même pour chacun des PVs, cette adresse est la même pour tous les PVs : le pointeur est scalaire et réside en un seul exemplaire dans l'espace mémoire du processeur scalaire.

- les pointeurs parallèles. Chaque PV possède sa mémoire locale et peut prendre des références dans cette mémoire. Un pointeur parallèle est une variable parallèle dont chaque élément est un pointeur sur la mémoire locale du PV. L'arithmétique sur les pointeurs pointant vers des zones parallèles est conservée. L'addition d'un pointeur avec une variable parallèle entière conduit à un pointeur parallèle. Chaque PV dispose alors d'un pointeur dans sa propre mémoire. La valeur de ce pointeur varie d'un PV à l'autre : le

pointeur est donc parallèle appartenant à la collection de l'objet pointé. L'usage de ce type nécessite des facilités d'adressage local pour les PPs.

On retrouve cette classification dans le typage des expressions suivantes (faisant suite aux déclarations précédent) :

```
&k;                  /* pointeur  scalaire  */
l + i, &(l[i]);      /* pointeurs scalaires */
tab;                 /* pointeur  scalaire  */
tab + i, &(tab[i]);  /* pointeurs scalaires */
&(s.b);              /* pointeur  scalaire  */

m + i, &(m[i]);      /* pointeurs paralleles */
l + j, &(l[j]);      /* pointeurs paralleles */
tab + j, &(tab[j]);  /* pointeurs paralleles */
s.c;                 /* pointeur  parallele  */
s.d;                 /* pointeur  parallele  */
```

On remarque que les variables parallèles d'une collection n'ont pas le droit de pointer dans l'espace d'adressage d'une autre collection ni dans l'espace d'adressage scalaire. Le pointeur parallèle est simplement une facilité d'adressage locale. En particulier, déréférencer un pointeur parallèle ne génère aucune communication entre les processeurs. De la même manière, la déclaration :

```
int * pixel p;    /* declaration illegale */
```

est illégale, car le pointeur `p` ferait référence à une donnée dans le domaine d'adressage scalaire.

Les structures doivent contenir des champs appartenant au même espace d'adressage. La structure de base est purement scalaire. Elle peut être ensuite déclarée dans une collection. Les champs pointeurs deviennent des pointeurs parallèles. Une structure scalaire a le droit de contenir un pointeur scalaire vers une variable parallèle. Dans ce cas cette structure ne peut être utilisée que dans le monde scalaire ou dans la collection pointée par la variable parallèle.

Les initialisations des variables parallèles à la déclaration ne sont supportées que pour les variables automatiques.

2.3.2 Allocation des variables parallèles

L'allocation d'un élément d'une variable parallèle dans un processeur virtuel est systématique et ne tient pas compte de l'activité de chaque processeur virtuel. Par processeur physique l'allocation d'une variable parallèle d'une collection de n éléments correspond à l'allocation de $v = \lceil \frac{p}{n} \rceil$ variables. Cette allocation étant systématique, on garantit un schéma d'adressage constant dans tous les processeurs physiques, ce qui est nécessaire pour l'existence de pointeurs scalaires vers une variable parallèle.

2.4 Typage des expressions

La notion de typage d'une expression de **C** est reprise et étendue pour **POMPC** avec un attribut supplémentaire : la collection.

Une expression qui ne contient pas de communications est évaluée localement. Les variables impliquées dans l'expression doivent résider au même endroit :

- sur le processeur scalaire,
- sur une machine virtuelle associée à une collection.

Une entorse est autorisée : une variable scalaire peut participer en tant qu'opérande à une expression évaluée dans une collection. Elle sera automatiquement promue en variable parallèle. La diffusion d'une variable scalaire est le seul cas de communication implicite en **POMPC** car son coût est en général très faible et voisin d'un coût d'accès local.

En aucun cas, sauf communication, des variables parallèles de différentes collections ne peuvent figurer dans la même expression.

En règle générale, une expression `e` de la forme `e1 <op> e2` est scalaire si `e1` et `e2` sont toutes les deux scalaires. Si au moins un des deux membres de l'expression est parallèle, `e` est alors parallèle.

Il reste à préciser quelques cas particuliers :

- les affectations : le membre gauche d'une affectation doit être parallèle dès que le membre droit l'est; Par contre, le membre droit peut être scalaire tandis que le membre gauche est parallèle. Il y a promotion implicite du scalaire en une variable parallèle;
- l'indexation : si l'index est parallèle, le tableau doit être parallèle et le résultat est parallèle. On a alors affaire à une indexation locale dans chaque PV;
- l'indirection : si le pointeur pointe vers une variable parallèle, le résultat est parallèle. On a alors affaire à une indirection locale dans la mémoire de chaque PV;
- l'accès à un champ d'une structure : si `record` est une structure parallèle, et si `field` est un champ de cette structure alors `record.field` est parallèle;
- l'addition ou la soustraction à un pointeur scalaire pointant vers une zone scalaire d'une variable parallèle est illégale;
- l'addition ou la soustraction à un pointeur scalaire pointant vers une zone parallèle d'une variable parallèle renvoie un pointeur parallèle.

Chapitre 3

Appels de fonctions et passages de paramètres

3.1 Passage des variables parallèles en paramètre

Lorsque l'on définit une fonction qui admet comme paramètre ou comme valeur de retour une variable parallèle, on peut désirer deux comportements :

1. la collection de la variable est considérée comme globale, il faudra lors de l'appel de la fonction, que l'argument appartienne effectivement à cette collection. On définit par exemple la fonction `zbuf` :

```
collection pixel;
pixel int screen;
pixel double depth;

void zbuf(pixel int rgb,pixel double z)
{
    where(z < depth) {
        screen = rgb;
        depth = z;
    }
}
```

   Dans la fonction `zbuf` la collection `pixel` est globale : la comparaison `z < depth` et l'affectation `screen = rgb` sont valides car les opérandes appartiennent tous à la même collection globale `pixel` (celle définie à la première ligne de l'exemple).

2. la collection de la variable est également considérée comme un paramètre de la fonction. On peut alors appeler cette fonction avec des variables parallèles de différentes collections. Ce mécanisme est très important pour permettre la définition de bibliothèques de fonctions utilisables avec n'importe quelle collection. On définit par exemple la fonction `abs` :

```
collection col int abs(col int n)
{
    where(n < 0) return -n;
    return n;
}
...
```

```
collection pixel;
...
{
    pixel int p,q;
    ...
    p = abs(q);
}
```

Le mot-clé `collection` dans la déclaration de la fonction `abs` spécifie que la collection `col` est un paramètre de celle-ci. Cette collection sert aussi bien pour la valeur de retour que pour le paramètre formel `n`. On pourra donc se servir de la fonction `abs` pour n'importe quelle collection. Dès qu'une collection est paramètre d'une fonction, celle-ci est alors locale à la fonction et donc différente d'une autre collection de même nom définie dans le contexte global. On ne peut donc avoir des expressions mélangeant des variables de cette collection locale avec des variables globales déclarée dans une collection homonyme.

La définition de la fonction `abs` comporte deux fois la présence de la collection `col`. Le mot clé `collection` qui la déclare paramètre de la fonction doit apparaître sur la première occurrence de `col`, à savoir ici sur la valeur de retour. En **POMPC**, la valeur de retour peut déterminer la collection à passer en paramètre. Lorsqu'une collection paramétrable apparaît plusieurs fois dans les paramètres formels et/ou la valeur de retour, le compilateur cherche lors de chaque appel à la fonction à déterminer la collection à utiliser comme argument. Chaque argument ou l'emploi du résultat de la fonction peuvent éventuellement permettre de déterminer cette collection. Cette détermination n'est pas toujours couronnée de succès :

- un argument scalaire est légal lorsque le paramètre formel associé est une variable parallèle. L'argument sera alors promu en variable parallèle. Cet argument ne permet pas de déterminer la collection passée en paramètre et doit être déterminée par un autre paramètre ou par la valeur de retour.
- la valeur de retour n'est pas utilisée dans une expression où figurent des variables parallèles. On ne peut attribuer une collection à celle-ci.

Pour chaque collection paramètre d'un appel de fonction, le compilateur peut se trouver dans les trois situations suivantes :

1. aucun des arguments ou valeur de retour sur cette collection n'ont permis de déterminer la collection. Il y a alors ambiguïté et le compilateur génère un message d'erreur, le problème est alors contourné par le programmeur avec un *cast* qui lève alors l'ambiguïté;
2. des arguments/valeur de retour ont déterminé une collection unique. Celle-ci devient alors la collection paramètre;
3. plusieurs collections ont été déterminées. Il y a alors conflit entre les arguments et un message d'erreur est généré par le compilateur.

Voici rassemblées dans un même exemple, les différentes possibilités pour le passage d'une collection en paramètre.

```
collection pixel;
collection line;

void  f();

collection col int abs(col int n)
{
    where(n < 0) return -n;
```

```
    elsewhere return n;
}

void g(void)
{
    pixel int p,q,r;
    line int a,b,c;

    ....
    q = abs(p);  /* valable pour la collection pixel */
    a = abs(b);  /* ainsi que pour la collection line */
    q = abs(-1); /* -1 est converti dans la collection pixel */
    f(abs(p));   /* f recoit un vecteur de la collection pixel */
    q = abs(a);  /* erreur : collections differentes */
    f(abs(-1));  /* erreur : collection indeterminee */
    ....
}
```

Le passage de paramètre fonctionne de la manière suivante :

1. tous les paramètres (y compris la valeur de retour) possèdent un type et une éventuelle collection;
2. les vérifications et conversions de types entre paramètres formels et arguments se font suivant la norme ANSI. Il est donc fortement encouragé d'utiliser les mécanismes de prototypage de ANSI **C** et de partager les prototypes à l'aide de fichiers d'en-tête;
3. un argument parallèle ne peut être associé à un paramètre formel scalaire;
4. un argument scalaire est converti en variable parallèle si le paramètre formel associé est parallèle;
5. Si argument et paramètre formel associé sont tous deux parallèles et que la collection du paramètre formel est aussi un paramètre, alors la collection de l'argument est passée en paramètre et devient la collection du paramètre formel.

Une fonction peut voir figurer dans ses paramètres et sa valeur de retour des variables parallèles de différentes collections. Le mécanisme de typage vérifie que chacune de ces collections est associée à une et une seule collection connue de l'appelant.

Dans notre exemple, la collection *col* apparaît 2 fois dans la déclaration de la fonction *abs*. Cela signifie que le paramètre *n* et la valeur de retour doivent appartenir à la même collection. On peut l'utiliser de différentes manières :

- `q = abs(p);` pour la collection *pixel*;
- `a = abs(b);` pour la collection *line*;
- `q = abs(-1);` dans la collection *pixel* imposée par la valeur de retour. -1 est automatiquement promu dans la collection *pixel*.
- `f(abs(p));`. Comme `f` n'a pas de prototype ANSI, on ne connaît pas la collection attendue en premier paramètre. C'est donc `p` qui définit la collection utilisée par *abs* qui est ensuite transmise à `f`.

Certains instances sont invalides, donc rejetées par le compilateur :

- `f(abs(-1))`. On ne peut déduire la collection utilisée par l'instance de `abs` ni par son argument ni par sa valeur de retour. La collection utilisée est alors indéterminée. On peut alors résoudre le problème de deux manières à l'aide d'un *cast* :
  1. `f(abs((pixel int)-1))`

2. `f((pixel int)abs(-1))`

- `a = abs(q)`. Le typage de la fonction *abs* fonctionne correctement. `abs(q)` a pour collection celle de q. Par contre on a une collision de collections lors de l'affectation.

Tous les paramètres sont passés par valeur. Cela implique d'éventuelles lenteurs dues à la recopie de variables parallèles de taille importante. Deux méthodes existent pour contourner ce problème :

- le passage par référence. On ne passe plus les variables parallèles mais les pointeurs vers celles-ci. Les conversions éventuelles de type et les promotions sont alors à la charge du programmeur. Le protypage ANSI des fonctions est alors très précieux.
- si le paramètre formel parallèle est déclaré constant (à l'aide du mot-clé `const`), alors la recopie de la variable parallèle est évitée. On n'échappe pas pour autant aux recopies dues aux conversions de types. Le prototypage est indispensable pour faire connaître au compilateur cette propriété de la fonction appelée.

3.2 Passage des collections en paramètre

On peut également passer une collection en paramètre sans qu'elle soit associée à une variable parallèle :

```
void pc_kill_collection(collection c);
collection pixel;
...
{
    ...
    pc_kill_collection(pixel);
}
```

Le paramètre formel est simplement déclaré `collection c`.

3.3 Surcharge des fonctions

Un mécanisme de surcharge permet de continuer à appeler des fonctions utilitaires aussi bien sur des arguments scalaires que des arguments parallèles.

Par défaut, ce mécanisme est inactifs pour toutes les fonctions. La commande

```
overload sin;
```

déclare que la fonction `sin` peut être surchargée. On peut donc définir différentes fonctions `sin` tant qu'on peut les différencier d'après les types de leurs paramètres formels :

```
double sin(double);
float sin(float);
collection col double sin(col double);
collection col float sin(col float);
```

Par contre la valeur de retour n'intervient pas dans le choix et dans la différenciation de la fonction.

Lors de l'appel d'une fonction surchargée, on cherche une fonction candidate parmi celles déclarées. Quatre cas sont possibles :

1. une fonction possède exactement les bons types de paramètres formels pour chacun des arguments : elle est alors unique et est retenue;

2. une seule fonction est compatible à l'aide des conversions standards d'arguments; elle est alors retenue;

3. plusieurs fonctions sont compatibles; un message d'erreur est alors généré;

4. aucune fonction ne convient; un message d'erreur est aussi généré.

On donne des noms différents aux fonctions surchargées dans le code. Le nom généré pour chaque définition de fonctions surchargées dépend de l'ordre de première définition des différentes surcharges. L'emploi de fichiers en-tête commun entre différents modules est la méthode fiable pour garantir cet ordre. La première définition d'une fonction surchargée conserve le nom originale de la fonction, ce qui permet de surcharger des fonctions existantes de **C**. Dans l'exemple précédent, la première fonction `double sin(double)` continue de s'appeler `sin` dans le code. On récupère donc telle quelle la fonction `sin` déjà existante dans C.

Chapitre 4

Le contrôle de flot parallèle

Le caractère SIMD du modèle de programmation interdit l'existence d'un contrôle de flot parallèle. Le contrôle de flot reste la tâche du processeur scalaire et est basé sur les constructeurs habituels de **C**.

La manipulation des variables parallèles d'activité des machines virtuels permet d'imiter le comportement d'un `if` parallèle avec le `where`.

En première approximation, tout le flot de **POMPC** peut être réduit au contrôle de flot scalaire classique de **C** et à l'opérateur de manipulation d'activité des collections (`where`).

A partir de ces constructeurs de base, un pseudo-contrôle de flot parallèle a été défini. Il est utile pour des raisons de lisibilité et d'optimisations.

A titre d'exemple, pour chaque opérateur du contrôle de flot, nous écrirons la fonction `fact` qui prend en entrée une variable et qui renvoie la variable parallèle qui contient la factorielle de chacun des éléments de la variable parallèle d'entrée.

4.1 Where : le if parallèle

La figure 4.1 montre l'exécution de la division parallèle : `A = C / B;` qui se passe mal à cause des valeurs nulles de la variable parallèle `B`. On constate la nécessité de pouvoir faire des tests pour éviter cette division par zéro. On aimerait en particulier pouvoir empêcher l'opération de s'exécuter là où le diviseur est nul. Plus généralement, on voudrait disposer d'un `if/else` parallèle permettant de choisir l'exécution d'instructions parallèles en fonction d'une expression

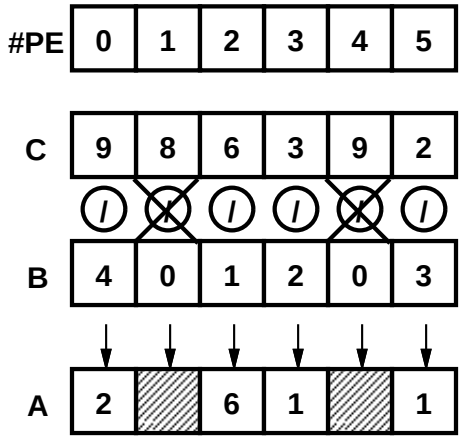

Figure 4.1 - une division parallèle.

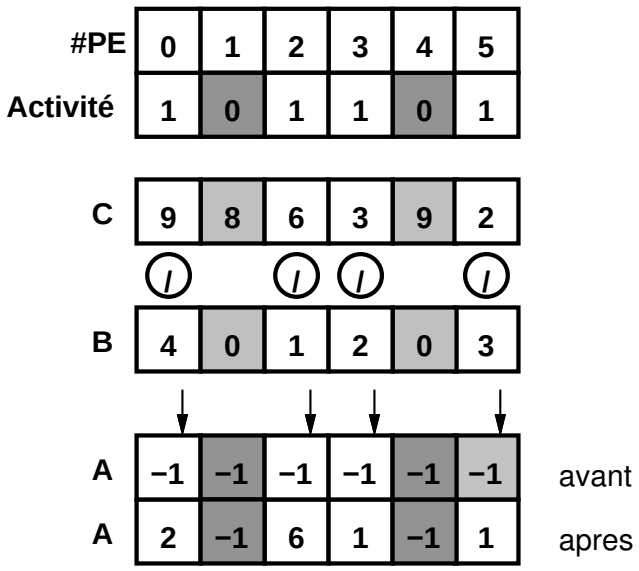

Figure 4.2 - une division parallèle.

booléenne parallèle. Le constructeur `where` joue ce rôle. La figure 4.2 montre l'exécution de la division lors du programme :

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

On suppose qu'au début du programme tous les processeurs virtuels de la collection sont actifs et que `A` est initialisé à -1. Le constructeur `where` se contente simplement de modifier l'activité de chaque processeur virtuel de la collection en fonction d'un booléen de la même collection à savoir local au processeur virtuel. Les PVs sauvegardent la valeur de leur activité courante pour pouvoir la restaurer et ceux dont le booléen est nul deviennent inactifs : leur activité est annulée. La division et l'affectation du résultat dans la variable `A` a lieu uniquement sur les PVs actifs. A la fin du bloc, l'ancienne activité est restaurée.

Comme le `where` n'est qu'une opération de manipulation de l'activité d'une collection (celle du booléen), ce constructeur n'affecte en aucune manière le comportement des calculs sur les autres machines virtuelles. En particulier une instruction scalaire est systématiquement exécutée, puisqu'elle ne fait partie d'aucune collection que pourrait modifier un `where`.

Chaque instruction est exécutée sur le seul critère du masque d'activité de la collection à laquelle elle appartient. Un `where` ne peut servir pour conditionner un appel de fonction puisque la fonction peut contenir des instructions scalaires : même si tous les processeurs virtuels de la collection testée sont inactifs, le corps du `where` est quand même exécuté et la fonction est appelée. Lors de l'exécution de celle-ci, les processeurs virtuels seront toujours inhibés, comme si la fonction n'avait pas eu d'effets sur eux. Par contre, les effets de bords de la fonction se font ressentir :

```
collection col int fact(col int n)
{
    where(n <= 1) return 1;
    elsewhere return n * fact(n-1);   /* BUG! */
}
```

Dans cet exemple, l'appel récursif se trouve à l'intérieur du corps du `elsewhere`. Il est systématiquement exécuté : la récursion est infinie. Il faut utiliser la construction décrite ci-dessous :

```
collection col int fact(col int n)
{
    where(n <= 1) return 1;
    if(|<-(pixel int) 1) return 1; /* return si 0 PV  actif */
    return n * fact(n-1);
}
```

L'opérateur unaire `|<-` correspond au calcul du *ou global* de son unique opérande. Cet opérateur est décrit dans le chapitre sur les communications.

4.2 Whilesomewhere : le while parallèle

Le `while` parallèle appelé `whilesomewhere` est caractérisé par une expression booléenne parallèle (`exp`).

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

La boucle est exécutée tant qu'il existe un processeur virtuel actif vérifiant l'expression booléenne `exp`. A chaque itération de la boucle, on évalue l'expression booléenne sur le jeu de processeurs actifs. Tant qu'il reste un PV actif, on exécute le corps de la boucle en inhibant les PVs dont la valeur du booléen est fausse. `break` et `continue` sont aussi définis suivant le comportement désiré : leur exécution affecte le masque d'activité de telle sorte que les processeurs virtuels actifs sont mis en attente jusqu'à la prochaine itération de la boucle pour `continue` et jusqu'à la sortie de la boucle pour `break`. La boucle est exécutée jusqu'à ce que tous les processeurs virtuels soient inactifs. Si la boucle ne contient pas de `break` ni de `continue` elle est alors équivalente à :

```
pixel int tempo;
while(|<-(tempo = exp)) {
    where(tempo) {
        ...
    }
}
```

Nous pouvons écrire la fonction factorielle de la sorte :

```
collection col int fact(col int n)
{
    col int p,r;
    r = 1;
    p = 2;
    whilesomewhere(p <= n) r *= p++;
    return r;
}
```

4.3 Dowhere : le do parallèle

Il existe pour `dowhere` une construction analogue à celle de `do` :

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

Le `dowhere` fonctionne comme le `whilesomewhere` mis à part que le test de fin de boucle est évaluée à la fin du corps. `break` et `continue` sont aussi autorisés pour `dowhere`. Un `dowhere` sans `break` ni `continue` peut se retraduire en :

```
pixel int tempo;
...
tempo = 1;
do {
    where(tempo) {
        ...
    }
} while(|<-(tempo = exp));
```

On peut réécrire `fact` de la manière suivante :

```
collection col int fact(col int n)
{
    col int n,p;
    p = 2;
    r = 1;
    dowhere {
        where(p > n) break;
        r *= p++;
    } whilesomewhere((col int) 1)
    return r;
}
```

4.4 Forwhere : le for parallèle

Il existe pour `for` une construction analogue à celle de `while` :

```
forwhere(e1;e2;e3) {
    ...
}
```

La collection associée au `forwhere` est celle de `<e2>`. `break` et `continue` sont aussi utilisables avec `forwhere`. Un `forwhere` sans `break` ni `continue` peut être retraduit en :

```
e1;
whilesomwhere(e2) {
    ...
    e3;
}
```

La fonction factorielle peut s'écrire :

```
collection col int fact(col int n)
{
```

```
    col int i,r;
    r=1;
    forwhere(i=2;i<=n;i++) r *=i;
    return r;
}
```

4.5 Switchwhere : le switch parallèle

Le constructeur `switchwhere` permet de réaliser un `switch` parallèle. Il s'emploie presque comme `switch` et admet `break`, `case` et `default`.

```
switchwhere(<e>) {
case c1 :
    ....
case c2 :
    ....
    break;
default :
    ....
}
```

Contrairement à `switch`, l'étiquette `default` doit être en dernière position du `switchwhere`.

On peut écrire `fact` comme suit :

```
collection col int fact(col int n)
{
    switchwhere(n) {
    case 0 :
    case 1 :
        return 1;
    case 2 :
        return 2;
    case 3 :
        return 6;
    case 4 :
        return 24;
    case 5 :
        return 120;
    default :
        where(n < 0) return 1;
        ...
    }
}
```

4.6 Goto

`goto` n'a pas été retraduit en parallèle[1] : il s'exécute de manière systématique. Il n'est valide que s'il ne traverse pas (aussi bien en entrée qu'en sortie) des blocs sous condition parallèle.

On peut également remarquer qu'il en est de même avec les fonctions : on ne peut pas exécuter des pointeurs de fonctions parallèles.

[1] Ce n'est pas un langage MIMD!

4.7 Return

La fonction «valeur absolue» peut être définie par :

```
collection generic int abs(generic int n)
{
    where(n < 0) {
        return -n;
        fprintf(stderr,"-");
    }
    return n;
    fprintf(stderr,"+");
}
```

Lorsqu'une fonction renvoie une variable parallèle, `return` prend un sens particulier. Deux cas sont à envisager :

- le `return` est imbriqué dans un ou plusieurs constructeurs de contrôle de flot parallèle sur la collection de la valeur de retour : la valeur de retour est modifiée pour les PVs actifs qui sont rendus inactifs jusqu'à la sortie de la fonction. Cette fin ne peut avoir lieu à cause d'éventuels PVs qui peuvent redevenir actifs et qui n'ont pas encore exécuté de `return` : l'exécution de la fonction continue. Dans l'exemple précédent, le premier `return` vérifie ce cas. Lors de l'exécution de la fonction, un caractère "`-`" s'affiche.

- le `return` n'est imbriqué dans aucun contrôle de flot concernant la collection de la valeur de retour : la valeur de retour est modifiée dans tous les PVs actifs, il n'y aura plus de PV actif dans la suite de l'exécution de la fonction : celle-ci est alorsinterrompue. Dans l'exemple précédent, le deuxième `return` vérifie ce cas. Lors de l'exécution de la fonction, aucun "`+`" ne s'affiche, car le flot a été interrompu avant. Un message d'avertissement est généré par le compilateur pour indiquer le caractère inutile de l'instruction situé derrière la deuxième instruction `return`.

4.8 Everywhere

Il peut être intéressant dans certains cas, de supprimer le mécanisme d'activité. On peut en avoir besoin pour des raisons de performance, d'initialisations, de gestion d'entrées-sorties...

La syntaxe est la suivante :

```
everywhere {
    ...;
}
```

On ne tient compte d'aucune activité de collection à l'intérieur du `everywhere`. Toutes les instructions parallèles sont exécutées sans tenir compte de leur activité. L'utilisation des constructions de contrôle de flot parallèle à l'intérieur d'un `everywhere` est interdite, car elles ne pourraient pas fonctionner correctement faute de masquage des instructions parallèles. On peut contourner ce problème à l'aide de 3 macros :

1. `pc_save_context` permet la sauvegarde de l'activité d'une collection dans une variable entière de cette collection;

2. `pc_clear_context` rend actif tous les processeurs virtuels en forçant l'activité de la collection;

3. `pc_restore_context` remet en place une activité sauvegardée.

On peut alors écrire :

```
pixel int save,c;
...
everywhere {
    pc_save_context(pixel,save);
    pc_clear_context(pixel);
}
c = ...;
where(c == 0) {
    ...
}
everywhere {
    pc_restore_context(pixel,save);
}
```

L'usage du constructeur **everywhere** peut introduire des effets de bords non portables et est donc aux risques et périls du programmeur et rend invalide la propriété d'invariance de la mémoire des PVs inactifs. Le compilateur envoie un message de mise en garde rappelant ce fait à l'utilisateur.

Chapitre 5

Les communications

Il existe 6 types de communications dans la syntaxe **POMPC** :

- diffusions scalaires,
- émissions scalaires,
- réceptions scalaires,
- réductions scalaires associatives,
- émissions parallèles,
- réceptions parallèles.

Les communications parallèles peuvent également s'effectuer sur grille en exploitant un voisinage connu. Toutes les communications (sauf la diffusion scalaire) sont caractérisées explicitement dans la syntaxe :

- par la présence d'une flèche vers la gauche (`<-`),
- par la présence d'un opérateur de réarrangement (crochets à gauche).

Les communications sont le seul moyen possible de faire interagir des objets appartenant à des machines virtuelles différentes. Pour définir en général ce déplacement de données, il faut spécifier un réarrangement.

5.1 Le réarrangement

Si on considère qu'une variable parallèle est un tableau délocalisé, le réarrangement peut être compris comme un opérateur parallèle d'indexation. Nous le notons donc suivant une syntaxe rappelant l'accès dans un tableau.

5.1.1 L'opérateur de réarrangement [] à gauche

Le réarrangement d'une variable parallèle est caractérisé par une adresse parallèle spécifiant pour chacun de ses éléments le processeur virtuel possédant l'élément auquel on s'intéresse.

```
collection pixel;
collection line;
...
double pixel vector;
int unsigned line address;
...
... [address]vector ...
```

La variable *vector* appartient à la collection *pixel*, tandis que *address* est une variable entière non-signée appartenant à la collection *line*. L'expression `[address]vector` représente une variable parallèle de type `double` (le type de *vector*) appartenant à la collection *line* (la collection de *address*). L'élément correspondant au $k^{\text{ième}}$ PVs de la collection *line* provient de l'élément de la variable *vector* résidant dans le PV de la collection *pixel* ayant pour numéro la valeur de l'élément de la variable *address* du $k^{\text{ième}}$ PVs de la collection *line*.

$$\forall k \in line,\ \texttt{[address]vector}_k = vector_{address_k}$$

L'adresse n'est malheureusement pas aussi simple que le laisse supposer l'exemple précédent. Le codage de l'adresse (en particulier le codage des dimensions et la gestion des processeurs virtuels) n'est pas public. Cette adresse doit donc être construite à partir d'un jeu de variables parallèles correspondant à la géométrie de la collection. La fonction `pc_build_address` de la bibliothèque standard permet de construire une telle adresse :

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

line unsigned int x,y,addr;
double pixel vector;
....
x = ...;
y = ...;
addr = pc_build_address(pixel,x,y);
.... [addr]vector....
```

La fonction `pc_build_address` construit dans la collection *line* la variable parallèle de réarrangement dans la géométrie de la collection *pixel*. Comme cette collection décrit un espace à deux dimensions, l'adresse est construite à partir de 2 coordonnées x et y. Il existe un raccourci pour simplifier la notation.

5.1.2 L'opérateur de réarrangement [[ ]] à gauche

Dans l'exemple précédent les deux dernières lignes peuvent s'écrire directement :

```
.... [[x,y]]vector....
```

L'adresse du réarrangement est également calculée par un appel à `pc_build_address`. Cette deuxième forme est plus lisible donc plus pratique que la première. Pourtant la première garde son intérêt :

- pour factoriser de multiples appels à `pc_build_address` lorsque plusieurs réarrangements semblables sont effectués;
- pour coder en un seul entier un n-uplet de coordonnées servant de référence dans une autre collection. L'adresse peut être vue comme un pointeur vers un PV d'une autre collection.

Le codage de l'adresse dans le système de coordonnées de la géométrie d'une collection fait apparaître le besoin de connaître le système de coordonnées de chaque PV. La fonction `pc_coord` permet de connaître les coordonnées d'une collection :

```
collection [16,16] matrice;
    ....
    matrice float m;
    matrice unsigned i,j;
    ....
```

```
i = pc_coord(0);
j = pc_coord(1);
...[[j,i]]m...;
```

Le réarrangement `[i,j]` correspond à l'identité tandis que le réarrangement `[j,i]` correspond à la transposition de la matrice. Cette fonction est aussi utile pour identifier un voisin dans la géométrie :

```
collection [1152,900] pixel;
....
pixel char b;
pixel unsigned x,y;
....
x = pc_coord(0);
y = pc_coord(1);
....[[x+1,y]]b...
```

Le voisin de droite dans la collection *pixel* est pointé par `[[x+1,y]]b`. On peut écrire cela de manière plus efficace à l'aide de l'opérateur de référence locale ".".

5.1.3 La référence locale "."

Comme beaucoup de machines parallèles possèdent des réseaux d'interconnexions qui privilégient les communications locales et privilégient également les communications régulières, il est important d'offrir au programmeur les moyens d'exprimer cette régularité à l'aide de la référence locale ".".

Cette référence ne peut figurer qu'à l'intérieur d'une énumération de coordonnées dans l'opérateur `[[]]` à gauche. Chaque expression de cette énumération représente la valeur d'une coordonnée pour composer l'adresse de réarrangement. La référence locale dans la $i^{\text{ème}}$ expression représente la valeur locale de la $i^{\text{ème}}$ coordonnée de la collection de l'objet réarrangé. L'exemple précédent peut s'écrire :

```
collection [1152,900] pixel;
....
pixel char b;
....
....[[.+1,.]]b...
```

Le compilateur **POMPC** sait analyser si un réarrangement exprimé à l'aide de références locales correspond à un cas connu de communications régulières rapides de l'architecture cible. Si cela s'avère être le cas, cette communication sera utilisée, sinon, les références locales sont converties en des appels à `pc_coord(0)`, `pc_coord(1)`,...

Les communications rapides exploitées sur les machines parallèles sont habituellement des communications sur grille. Celles-ci se traduisent par la forme du réarrangement :

$$\texttt{...[[.}\pm\Delta x\texttt{,.}\pm\Delta y\texttt{,.}\pm\Delta z\texttt{,...]]b...}$$

Par exemple, dans un espace de dimension 2, pour prendre une métaphore géographique :

- `[[.,.]]` représente le point courant;
- `[[.+1,.]]` représente le voisin Est;
- `[[.-1,.]]` représente le voisin Ouest;
- `[[.,.+1]]` représente le voisin Nord;
- `[[.,.-1]]` représente le voisin Sud;

- `[[.+1,.+1]]` représente le voisin Nord-Est;
- `[[.-1,.+1]]` représente le voisin Nord-Ouest;
- `[[.+1,.-1]]` représente le voisin Sud-Est;
- `[[.-1,.-1]]` représente le voisin Sud-Ouest.

5.1.4 Adresse scalaire

Il est possible d'utiliser les 2 opérateurs de réarrangement avec des adresses scalaires. Dans ce cas, le résultat d'un réarrangement est scalaire. C'est la sélection d'un élément particulier d'une variable parallèle. Pour construire une adresse scalaire à partir d'un jeu de coordonnées scalaires, on utilise la fonction `pc_build_address` surchargée pour le domaine scalaire :

```
int x,y;
int address;
x = ...;
y = ...;

....[[x,y]] vector...
address = pc_build_address(pixel,x,y);
....[address] vector...
```

5.1.5 Validité des adresses et rebouclage circulaire

Dans la composition des adresses, il peut arriver qu'une coordonnée soit en dehors de son domaine de définition. Deux attitudes sont envisageables :

- l'axe est circulaire : on considère alors la coordonnée modulo la taille de la collection suivant la dimension considérée;
- l'axe n'est pas circulaire. L'élément en question ne sera pas sélectionné.

On peut spécifier indépendamment un tel comportement pour chaque axe de chaque collection :

- `pc_set_torus(pixel,mask)` positionne un masque de description pour la collection `pixel`. Si le $n^{\text{ième}}$ bit du masque (en partant des poids faibles) vaut 1, alors la $n^{\text{ième}}$ coordonnée est circulaire. Par défaut tout axe de toute collection est circulaire.
- `pc_get_torus(pixel)` récupère la valeur du masque de la collection `pixel`.

5.2 Description des communications

La figure 5.1 illustre les 6 communications connues par la syntaxe. La plupart des communications vont utiliser des opérateurs de réarrangement. Les exemples suivant utilisent le réarrangement `[]`. On peut bien entendu le remplacer par l'autre forme de réarrangement `[[]]`. Les deux premières communications n'ont pas besoin de cette opérateur car elles régissent les mouvements de données entre le processeurs et tous les processeurs virtuels actifs d'une collection.

5.2.1 Diffusion scalaire

La diffusion scalaire est la seule communication implicite de **POMPC**. Elle consiste à distribuer une même valeur scalaire à tous les PVs actifs d'une collection (figure 5.2) :

```
int i;
```

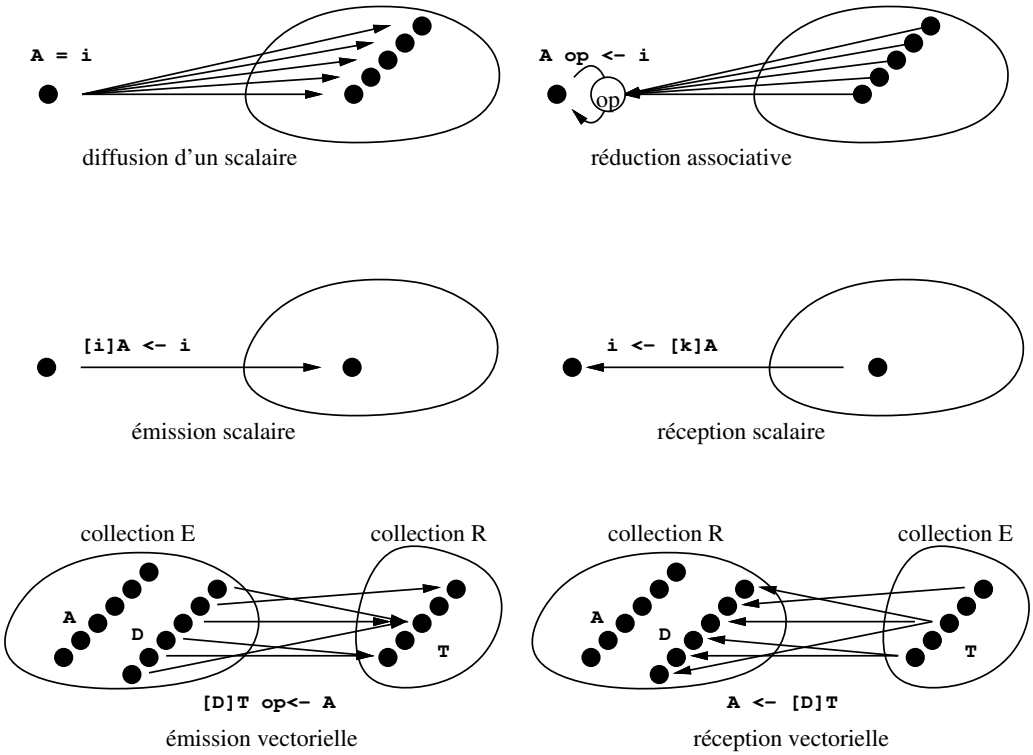

Figure 5.1 - les communications de **POMPC**.

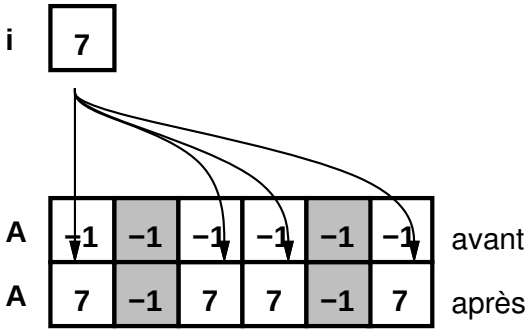

Figure 5.2 - la diffusion scalaire `A = i;`

```
pixel int A;

A = i;
```

5.2.2 Réductions scalaires associatives

Les réductions scalaires associatives servent à récupérer le résultat de la combinaison de tous les éléments actifs d'une variable parallèle à l'aide d'une opération associative.

```
int i,j;
pixel int A;
```

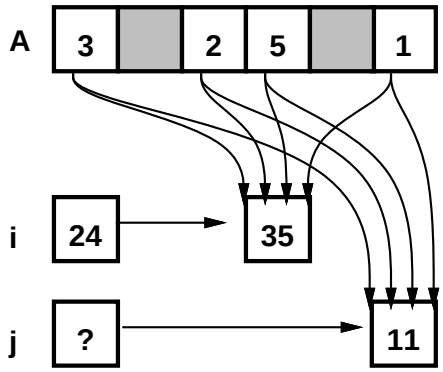

Figure 5.3 - les réductions scalaires associatives `i +<- A;` et `j = +<- A;`

Table 5.1 - les concentrations scalaires associatives.

Opération	*Comportement*	*Nom*
`i +<- a`	`i +=` $\sum_{k \in \texttt{Actif(a)}} \texttt{a}_k$	*plus* global
`i -<- a`	`i -=` $\sum_{k \in \texttt{Actif(a)}} \texttt{a}_k$	*moins* global
`i |<- a`	`i |=` $\bigvee_{k \in \texttt{Actif(a)}} \texttt{a}_k$	*ou* global
`i &<- a`	`i &=` $\bigwedge_{k \in \texttt{Actif(a)}} \texttt{a}_k$	*et* global
`i *<- a`	`i *=` $\prod_{k \in \texttt{Actif(a)}} \texttt{a}_k$	*multiplier* global
`i ^<- a`	`i ^=` $\bigoplus_{k \in \texttt{Actif(a)}} \texttt{a}_k$	*ou exclusif* global
`i >?<- a`	$i = \max(i, \max_{k \in \text{Actif(a)}} \texttt{a}_k)$	*maximum* global
`i <?<- a`	$i = \min(i, \min_{k \in \text{Actif(a)}} \texttt{a}_k)$	*minimum* global
`+<- a`	$\sum_{k \in \texttt{Actif(a)}} \texttt{a}_k$	*plus* global unaire
`-<- a`	$\sum_{k \in \texttt{Actif(a)}} \texttt{a}_k$	*moins* global unaire
`|<- a`	$\bigvee_{k \in \texttt{Actif(a)}} \texttt{a}_k$	*ou* global unaire
`&<- a`	$\bigwedge_{k \in \texttt{Actif(a)}} \texttt{a}_k$	*et* global unaire
`*<- a`	$\prod_{k \in \texttt{Actif(a)}} \texttt{a}_k$	*multiplier* global unaire
`^<- a`	$\bigoplus_{k \in \texttt{Actif(a)}} \texttt{a}_k$	*ou exclusif* global unaire
`>?<- a`	$\max(i, \max_{k \in \text{Actif(a)}} \texttt{a}_k)$	*maximum* global unaire
`<?<- a`	$\min(i, \min_{k \in \text{Actif(a)}} \texttt{a}_k)$	*minimum* global unaire

```
i = 24;
i +<- A;
j = +<- A;
```

ajoute à `i` la somme des éléments actifs d'une variable parallèle `a`; Par contre `j` reçoit la somme des éléments actifs de `A`. Cet opérateur existe donc sous les formes unaire et binaire. les opérations possibles sont récapitulées dans le tableau 5.1.

Par exemple, on peut récupérer le nombre de processeurs virtuels actifs d'une collection de la manière suivante :

```
pixel char A;
...
where (A < seuil) {
     printf("%d pixels verifient A < %d\n",+<- (pixel int) 1,seuil);
}
```

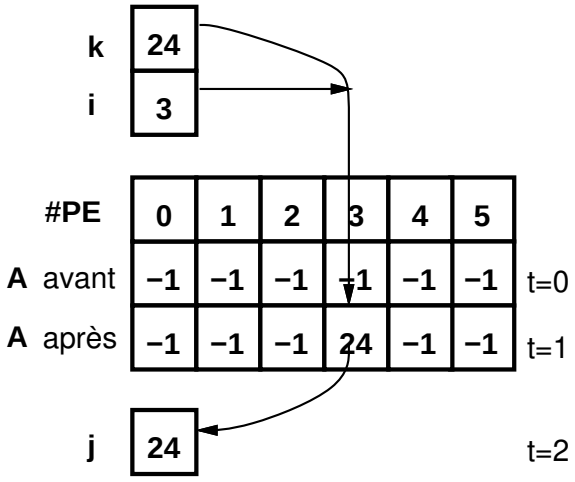

Figure 5.4 - l'émission et la réception scalaire.

Une des concentrations scalaires associatives joue un rôle important :

```
if(|<- flag) where(flag) {
    ...;
}
```

Elle permet de calculer le *ou global* d'un booléen parallèle et est utilisée dans le contrôle de flot parallèle. On peut également s'en servir pour éviter l'exécution d'un bloc `where` lorsqu'aucun processeur n'est actif.

5.2.3 Emissions et réceptions scalaires

Les émissions et réceptions scalaires servent à lire et à écrire un élément particulier d'une variable parallèle (fig. 5.4) :

```
int i,j,k;
pixel int a;

[i]a <- k;
j =  [i]a;
```

On les utilise pour initialiser des variables parallèles et pour les relire. Ces opérations sont indépendantes de l'activité de la collection. En fait, l'indice étant scalaire, l'activité est donc celle du monde scalaire qui est toujours actif.

5.2.4 Emissions parallèles (*Send*)

Les émissions parallèles sont des communications entre deux variables parallèles, suivant un schéma de routage donné par une troisième. L'émission parallèle (figure 5.5) s'écrit :

```
pixel char Destination;
line unsigned int Address;
line char Source;
...
[Address]Destination <- Source;
...
[Address]Destination +<- Source;
```

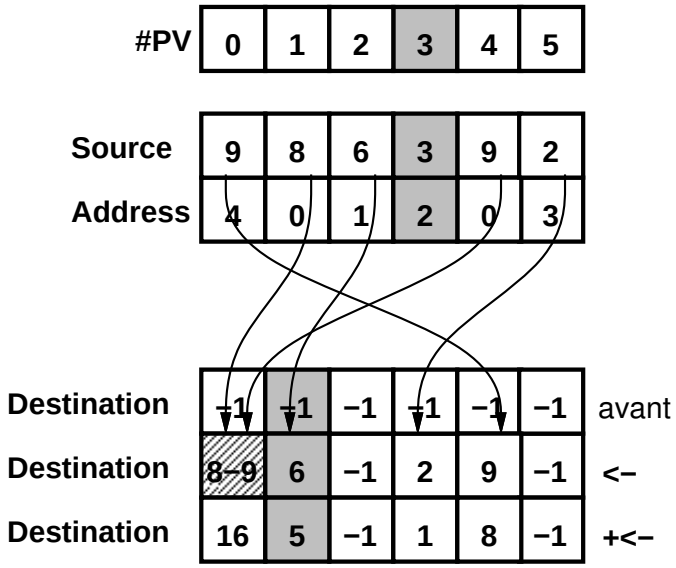

Figure 5.5 - l'émission parallèle.

Table 5.2 - les émissions parallèles.

Opération	*Comportement*
`[i]a <- b`	$\forall k \in \text{Actif(b)}, \texttt{a}_{\texttt{i}_k} = \texttt{b}_k$
`[i]a +<- b`	$\forall k \in \text{Actif(b)}, \texttt{a}_{\texttt{i}_k} \texttt{+=} \texttt{b}_k$
`[i]a -<- b`	$\forall k \in \text{Actif(b)}, \texttt{a}_{\texttt{i}_k} \texttt{-=} \texttt{b}_k$
`[i]a |<- b`	$\forall k \in \text{Actif(b)}, \texttt{a}_{\texttt{i}_k} \texttt{\|=} \texttt{b}_k$
`[i]a &<- b`	$\forall k \in \text{Actif(b)}, \texttt{a}_{\texttt{i}_k} \texttt{\&=} \texttt{b}_k$
`[i]a *<- b`	$\forall k \in \text{Actif(b)}, \texttt{a}_{\texttt{i}_k} \texttt{*=} \texttt{b}_k$
`[i]a ^<- b`	$\forall k \in \text{Actif(b)}, \texttt{a}_{\texttt{i}_k} \texttt{\^{}=} \texttt{b}_k$
`[i]a >?<- b`	$\forall k \in \text{Actif(b)}, \texttt{a}_{\texttt{i}_k} = \max(\texttt{a}_{\texttt{i}_k}, \texttt{b}_k)$
`[i]a <?<- b`	$\forall k \in \text{Actif(b)}, \texttt{a}_{\texttt{i}_k} = \min(\texttt{a}_{\texttt{i}_k}, \texttt{b}_k)$

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 actif 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. Si l'opérateur spécifié est `<-`, alors le rangement est une simple affectation. Si l'opérateur spécifié est *op*`<-` où *op* est un opérateur associatif, alors la valeur du message reçu est accumulée dans l'élément local de la variable de destination à l'aide de l'opérateur *op* (tab. 5.2).

Lorsque l'opérateur `<-` est utilisé, l'accumulation se fait par l'opérateur d'assignation `=` qui n'est pas associatif. Le résultat n'est pas *a priori* déterministe (même s'il est possible qu'il reste constant d'une exécution à une autre sur la même machine).

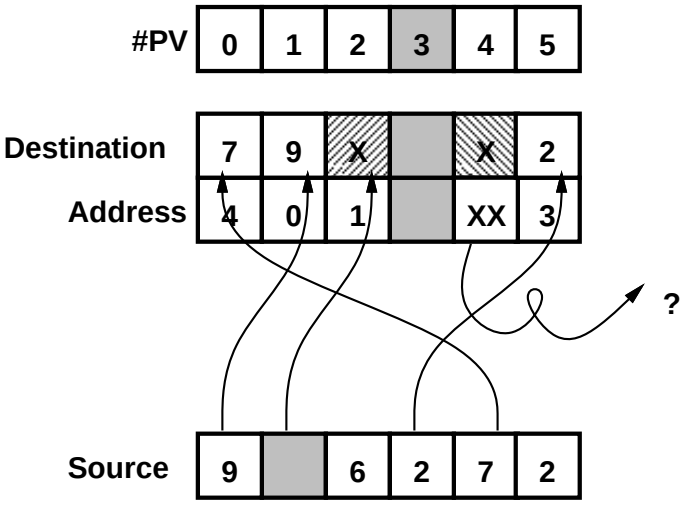

Figure 5.6 - la réception parallèle.

Comme le réarrangement ne possède *a priori* aucune propriété particulière, il n'est en général, ni injectif, ni surjectif. Dans la collection d'arrivée, on distingue donc 3 cas :

- aucun message n'arrive. La destination n'est pas modifiée;
- un seul message arrive. La donnée reçue est rangée dans la destination.
- plusieurs messages arrivent. Chaque fois, la donnée reçue est rangée dans la destination. A la fin la destination contient la valeur de la dernière donnée reçue. Pour l'exemple précédent dans le processeur virtuel $0, 2$ messages sont reçus et suivant leur ordre d'arrivée, le résultat sera 8 ou 9.

Un non déterminisme peut également s'introduire avec des opérateurs associatifs lorsqu'on travaille avec des flottants (`float`, `double` et `long double`) ou que l'on risque de faire des débordements provisoires dans les types entiers (`char`, `short`, `int` et `long`). Ces opérateurs ne sont pas en général associatifs, à cause du codage inexact des flottants ou des débordements. On ne garantit pas en **POMPC** un ordre dans lequel les messages vont être réduits.

Le typage des membres droit et gauche de l'expression s'accordent sur la même collection (ici la collection `line`). C'est cette collection qui définit l'activité de l'instruction.

Dans l'exemple précédent, le processeur virtuel 3 de la collection `line` est inactif (grisé sur la fig. 5.5). Il n'envoie donc aucun message.

Par contre, bien que le processeur virtuel 1 de la collection `pixel` soit inactif, il reçoit un message et le range malgré son inactivité.

L'émission parallèle se caractérise par la présence d'un opérateur de réarrangement parallèle comme membre gauche d'une opération `<-`. C'est la seule possibilité de voir apparaître un réarrangement parallèle dans un membre gauche.

5.2.5 Réceptions parallèles (*Get*)

La figure 5.6 montre un exemple de réception parallèle :

```
line char Destination;
line unsigned int Address;
pixel char Source;
...
Destination = [Address]Source;
```

La réception parallèle fait intervenir deux collections :

- la collection de destination (ici `line`) qui est celle de la variable de destination ainsi que celle de l'adresse du réarrangement,
- la collection de la source (ici `pixel`).

Le fonctionnement de la réception parallèle est le suivant :

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 possède 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.

Contrairement à l'émission parallèle, même si le réarrangement n'est pas *a priori* bijectif, chaque processeur virtuel actif de la collection de destination est assuré de recevoir une et une seule réponse à sa requête, à condition que l'adresse soit valide.

L'activité de la réception parallèle est liée à la collection de destination. Seule les PVs actifs de cette collection soumettent une requête et modifie la valeur de la variable (même si la requête correspond à une adresse invalide, comme le processeur 4 dans l'exemple).

Tous les processeurs virtuels de la collection source répondent aux requêtes reçues indépendamment de leur activité. Ce qu'ils renvoient peut être invalide (si le processeur virtuel est inactif). Dans l'exemple de la figure 5.6 le processeur virtuel 1 de la collection de source renvoie au processeur virtuel 2 de la collection de destination sa valeur locale qui peut être indéterminé.

La réception scalaire, est caractérisé par la présence d'un réarrangement parallèle dans le membre droit d'une affectation. Elle peut également être utilisée dans une expression.

Chapitre 6

Expressions parallèles

Tous les opérateurs de **C** sont disponibles pour la construction d'expression parallèles. Quelques nouveaux opérateurs ont été introduits en **POMPC**. Il convient donc de décrire les nouveaux opérateurs et préciser la sémantique des extensions des opérateurs sous-tendant du contrôle de flot caché.

6.1 Les opérateurs minimum et maximum

En **POMPC**, sont introduits les opérateurs :

- `>?` l'opérateur maximum,
- `<?` l'opérateur minimum,

Les instructions :

```
max = e1 >? e2;
min = e1 <? e2;
```

calculent le maximum et le minimum de `e1` et `e2` et stoquent le résultat dans les variables `max` et `min`. Les opérandes `e1` et `e2` ne sont évalués qu'une seule fois par opérateur. Ces deux opérateurs sont valides pour les paires de types compatibles avec l'opérateur `+`. Le type du résultat est le même type que `e1 + e2`.

Ces opérateurs existent également sous leur forme accumulatrice :

- `>?=` l'opérateur d'accumulation du maximum,
- `<?=` l'opérateur d'accumulation du minimum,

Les instructions :

```
max >?= e;
min <?= e;
```

sont équivalentes à :

```
max = max >? e;
min = min <? e;
```

à la différence près que `max` et `min` ne sont évalués qu'une fois, conformément à la règle de **C**.

Ils existent également en tant qu'opérateurs de réduction globale :

- `>?<-` l'opérateur de réduction en le maximum;
- `<?<-` l'opérateur de réduction en le minimum;

Ces opérateurs existent suivant la forme binaire :

```
int min,max;
pixel int A;

max >?<- A;
min <?<- A;
```

qui accumulent dans `min` et `max` le maximum et le minimum de tous les éléments actifs de la variable parallèle `A`. Ils existent suivant la forme unaire :

```
int min,max;
pixel int A;

max = >?<- A;
min = <?<- A;
```

qui stockent dans `min` et `max` le maximum et le minimum de tous les éléments actifs de la variable parallèle `A`.

On peut également se servir des opérateurs minimum et maximum comme opérateurs associatifs dans la collision des émissions parallèles.

6.2 Les opérateurs court-circuits

Les opérateurs `&&` et `||` sont des opérateurs qui n'évaluent leur opérande droit que lorsque c'est nécessaire. Dans l'expression :

```
a && b
```

`b` n'est évalué que si `a` est vrai donc différent de 0. Cette propriété est conservée pour des opérandes parallèles. Seuls les PVs actifs qui vérifient `a` évaluent `b`. Dans l'expression :

```
a || b
```

`b` n'est évalué que si `a` est faux donc égal à 0. Cette propriété est conservée pour des opérandes parallèles. Seuls les PVs actifs qui ne vérifient pas `a` évaluent `b`.

6.3 L'opérateur du choix

L'opérateur du choix (`?:`) est étendu aux variables parallèles avec la même sémantique. Dans l'expression :

```
a ? b : c
```

`a` est d'abord évalué. Si `a` est vrai, on renvoie le résultat de l'évaluation de `b`, sinon on renvoie le résultat de l'évaluation de `c`. Le choix de l'évaluation est local à chaque PV.

6.4 Précédence des opérateurs

L'introduction de nouveaux opérateurs nécessite la mise à jour des tables de précédences des opérateurs de (tab. 6.1). Le tableau les présente par priorité décroissante de haut en bas.

Table 6.1 - précédence des opérateurs de **POMPC**.

opérateurs	*associativité*
`() [`*droit*`] -> .`	gauche à droite
`! ~ ++ -- + - *  & (`*type*`) sizeof +<- -<- *<- |<- &<- ^<- >?<- <?<-`	droite à gauche
`* / %`	gauche à droite
`+ -`	gauche à droite
`<< >>`	gauche à droite
`< <= > >=`	gauche à droite
`== !=`	gauche à droite
`&`	gauche à droite
`^`	gauche à droite
`|`	gauche à droite
`>? <?`	gauche à droite
`&&`	gauche à droite
`||`	gauche à droite
`?:`	droite à gauche
`[`*gauche*`] [[`*gauche*`]]`	droite à gauche
`<- +<- -<- *<- |<- &<- ^<- >?<- <?<-`	droite à gauche
`= += -= *= /= %= |= <<= >>= >?= <?= &= ^=`	droite à gauche
`,`	gauche à droite

Chapitre 7

Virtualisation et performance

Sauf pour certaines architectures cibles (dont la Connection Machine CM-2), la virtualisation est à la charge du compilateur **POMPC**. Plusieurs stratégies sont envisageables suivant les architectures cibles pour générer le code le plus performant. Même si on peut appliquer des stratégies pour utiliser le pipeline vectoriel de certains processeurs, il reste toujours une boucle externe qui gère la différence entre le rapport de virtualisation de la collection et la taille des vecteurs. Cette boucle (appelée *strip-mining* dans le jargon des processeurs vectoriels) est appelée ici boucle de virtualisation. Elle a pour but d'énumérer tous les processeurs virtuels localisés sur un même processeur physique.

7.1 Collections physiques

La virtualisation concerne aussi l'allocation des variables parallèles. Chaque variable parallèle est distribuée sur les processeurs physiques, qui reçoivent chacun plusieurs éléments de la variable. Une variable parallèle est gérée physiquement comme un tableau par chaque processeur physique. Cet organisation ralentit l'accès aux variables parallèles par l'existence systématique de cette indirection. Dans certains cas, la virtualisation ne présente pas d'intérêt particulier

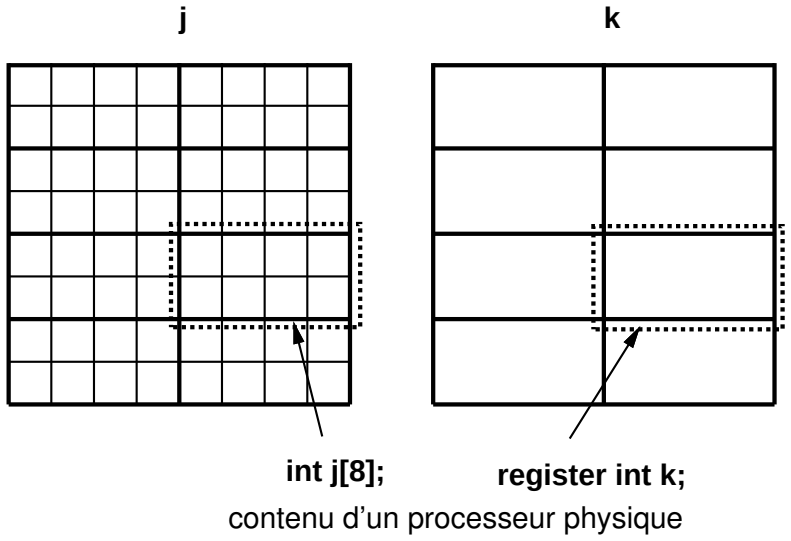

Figure 7.1 - l'allocation physique des variables parallèles.

et l'utilisation d'une collection d'un PV par PP est peu performante à cause du surcoût de la gestion de la virtualisation.

Lorsque la virtualisation n'est pas nécessaire pour une collection, on la peut définir ainsi :

```
collection physical processor;
processor int i;
```

On peut également désirer déclarer dans la projection physique d'une collection normale, une variable parallèle qui possède un seul élément par processeur physique :

```
collection [8,8]image;
image int j;
physical image int k;
```

La variable `j` est une variable parallèle virtuelle tandis que la variable `k` est une variable parallèle physique (fig. 7.1).

7.2 Virtualisation

La boucle de virtualisation représente un certain surcoût qu'il est important de pouvoir minimiser 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 d'une itération 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 (éventuellement vectoriels) de chaque processeur physique. On les alloue effectivement dans la projection physique de la collection concernée suivant le schéma de la figure 7.1. On obtient ainsi une diminution de la ressource mémoire nécessaire, ainsi qu'une réduction substantielle des temps d'accès à ces variables. Tout pousse donc à l'élargissement maximal de cette boucle de virtualisation.

Cet élargissement de la boucle est automatiquement géré par le compilateur en préservant une sémantique unique. Il est limité par 5 restrictions :

1. une boucle de virtualisation ne concerne qu'une seule collection;
2. elle ne contient pas d'affectation de variables d'autres collections;
3. elle ne contient pas de communication autre que la diffusion scalaire;
4. elle ne contient pas d'affectation de variables scalaires;
5. elle ne contient pas d'appels de fonctions susceptibles de cacher les restrictions précédentes.

Les trois premières restrictions peuvent difficilement être contournées, car elles sont en général liées à l'algorithme utilisé. Les deux dernières peuvent être dans certains cas contournées :

- à l'aide de l'utilisation d'un `forwhere`;
- à l'aide de l'utilisation d'un mot-clé précisant qu'une fonction n'a pas d'effets de bords.

7.3 Utilisation du forwhere

La quatrième restriction peut être contournée en faisant une promotion de la variable scalaire modifiée en une variable parallèle. Ce n'est possible que si l'algorithme exprimé le permet.

L'exemple le plus significatif concerne les affectations d'indices de boucle. On a, par exemple, intérêt à réécrire le programme:

```
pixel double tab[200],k;
int i;
...
for(i=0;i<4;i++) {
    k = tab[i];
    k = k * (1 - k);
    tab[i] = k;
}
```

L'intérieur de la boucle peut être géré dans une seule et même boucle de virtualisation. Les affectations `i = 0` et `i++` empêchent l'élargissement de cette boucle au delà de la boucle `for`. Par contre en réécrivant le programme:

```
pixel double tab[200],k;
...
{
    pixel int i;
    forwhere(i=0;i<4;i++) {
        k = tab[i];
        k = k * (1 - k);
        tab[i] = k;
    }
}
```

on obtient cet élargissement de la boucle de virtualisation, qui peut englober alors le bloc dans lequel est défini `i` qui est alors alloué dans la collection physique projetée de `pixel`.

7.4 Virtualisation des appels de fonctions

Lorsque tout le corps d'une fonction ne concerne qu'une seule collection et ne contient pas d'effets de bord en dehors de cette collection, elle est virtualisée au moyen d'une seule boucle. Cette fonction peut s'exécuter en parallèle sur les processeurs sans la moindre interaction entre eux. D'un point de vue purement sémantique, il n'y a plus contraintes de synchronisation entre les différents processeurs physiques, la fonction peut être exécutée de manière asynchrone sans mettre en péril la validité des résultats : la fonction est SPMD.

Un mécanisme permet en **POMPC** de spécifier qu'une fonction possède cette propriété SPMD pour permettre ainsi de rejeter la gestion de la virtualisation sur la fonction appelant et non plus sur la fonction appelée. Une fonction SPMD est toujours associée à une collection. Au lieu de déclarer la fonction en utilisant le mot-clé `collection` pour spécifier la collection paramétrable, on utilise cette fois le mot-clé `spmd` qui déclare la collection comme une collection physique sans effet de bords autre que locaux à chaque PV. La fonction `abs` que nous avons de nombreuses fois définie vérifie cette propriété et peut être redéfinie :

```
spmd generic abs(generic int)
{
    where(n < 0) return -n;
    return n;
}
```

Elle peut être utilisée dans un bloc sans rompre l'élargissement de la boucle virtualisation. Beaucoup de fonctions présentent ce caractère SPMD, en particulier toutes celles qui font des calculs en local sur chaque éléments d'une collection. On trouve entre autres les fonctions

mathématiques classiques. Ainsi l'appel à la fonction `sin` parallèle n'empêche pas l'élargissement de la boucle de virtualisation.

Lors de l'instance d'une fonction SPMD, celle-ci est appelée autant de fois qu'il y a de processeurs virtuels par processeur physique et toutes les variables (parallèles) de la fonction (varaibles locales et paramètres) n'existent qu'en un seul exemplaire par processeur physique.

7.5 Utilisation de ressources physiques

Les deux dernières sections de ce chapitre présentent des techniques assez délicates qui ne sont pas d'un usage courant pour le développement d'application en **POMPC**. Le débutant peut ignorer cette partie en première lecture.

La virtualisation des collections nécessite la mise au point de fonctions de communication capable de gérer des communications entre processeurs virtuels et de les simuler sur un réseau de processeurs physiques. Ces fonctions nécessitent l'interaction du domaine virtuel d'une collection et de son domaine physique projeté.

Il convient de donner les règles de typages qui assurent la sémantique unique d'un programme valide :

- seules les variables physiques et virtuelles ne peuvent interagir que si elles appartiennent à une même collection;
- dans un membre droit d'une affectation d'une variable virtuelle utilisant variables physiques et variables virtuelles d'une même collection, les variables physiques sont promues en variables virtuelles. Par exemple l'indexation d'un tableau physique par un indice virtuel est autorisée et fournit un résultat virtuel :

```
physical pixel double lookup[256];
pixel char i;
pixel double res;

...
res = lookup[i];
```

  Cela permet par exemple de partager des tabulations d'opérateurs entre les processeurs virtuels d'un même processeur physique;
- on ne peut, pas sauf cas particulier détaillé plus bas, avoir une variable physique en membre gauche d'une affectation avec un membre droit virtuel. L'affectation d'une variable physique n'est possible qu'avec une expression physique ou scalaire. Dans ce cas, l'instruction sera exécutée une fois et sera donc en dehors de toute boucle de virtualisation.

7.6 With : la gestion explicite de la virtualisation

On n'autorise l'affectation d'une variable physique par une expression virtuelle qu'à l'intérieur d'une structure explicite de gestion de virtualisation. L'affectation d'une variable physique s'effectue normalement sans virtualisation tandis que le membre droit (une variable virtuelle) n'a de sens qu'à l'intérieur d'une boucle de virtualisation. Si on admet cette opération en la plongeant à l'intérieur d'une boucle de virtualisation, on n'obtient une sémantique précise des opérations effectuées que si la portée de la boucle de virtualisation est définie. Le constructeur `with` permet d'expliciter la localisation de la boucle de virtualisation :

```
with(pixel) {
    ...
}
```

Une telle construction n'est valide que si le bloc du `with` vérifie les contraintes précédamment énoncées pour l'élargissement de la boucle de virtualisation au bloc complet. La fusion de boucles `with` conserve une sémantique unique sauf si une variable physique est modifiée par une des boucles. Par exemple,

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

n'est pas équivalent à

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

Le premier exemple accumule dans `i` la somme de tous les éléments de la variable virtuelle `I` localisés sur le processeur physique et range la somme dans tous les éléments locaux de la variable virtuelle `J`.

On peut illustrer l'emploi d'une telle construction par exemple par le calcul d'une somme globale. L'opérateur `+<-` peut être retranscrit à l'aide de la fonction :

```
int reduction_with_add(collection generic int n)
{
    physical generic int local_sum = 0;
    with(generic) {
        local_sum += n;
    }
    return physical_reduction_with_add(local_sum);
}
```

On a ainsi codé une fonction de communication virtuelle à l'aide de son équivalent physique.

Dans certain cas, lorsqu'on veut manipuler explicitement les variables virtuelles on a besoin de l'indice courant de la boucle qui est représenté par un point "." :

```
generic int b;
generic int c;
physical generic int * physical generic buf;
physical generic int * physical generic p;
physical generic int n,i,size;
...
p = buf = (physical generic int * physical generic) &b;
with(generic) {
    where(c) *p++ = .;
}
```

```
size = p - buf;
p = (physical generic int * physical generic) &c;
forwhere(i=0;i<size;i++) {
    n = buf[i];
    ...
    p[n] = ...;
}
```

Ce petit programme crée un tableau physique `buf` alloué dans l'espace mémoire utilisé par `b` dont la taille est le rapport du nombre de PV par PP. Dans la boucle `with` on accumule dans ce tableau la liste des indices des éléments actifs de la collection `generic` vérifiant la condition `c`. Dans la boucle `forwhere` suivante, on énumère les indices accumulés dans le tableau. On peut alors faire des calculs sur les éléments virtuels de la collection ainsi sélectionnés.

Annexe A

Compilation

Le compilateur, pour fonctionner doit connaître un point d'entrée dans système de répertoires UNIX où il pourra trouver les fichiers dont il a besoin. Suivant les configurations locales, il existe deux possibilités :

1. le répertoire par défaut connu par le compilateur est correct;
2. on indique la position de ce répertoire à l'aide de la variable d'environnement `POMPCLIB`. Si la racine d'installation de **POMPC** est localisée dans le répertoire *dir*, la variable d'environnement doit être positionnée en *dir*`/pompc/lib`.

Tout module **POMPC** doit inclure le fichier `pompc.h`.
La compilation est réalisée à l'aide du programme `pcc`. Les arguments de ce programme sont analogues à ceux de `cc`. On compile un exécutable :

- directement par la commande :

```
pcc -o my_program module1.pc module2.pc module3.pc
```

- de manière séparée par les commandes :

```
pcc -c module1.pc
pcc -c module2.pc
pcc -c module3.pc
pcc -o my_program module1.o module2.o module3.o
```

Le programme `pcc` est un petit programme qui coordonne l'exécution d'autres programmes en 3 temps.

1. Pour chaque module dont l'extension est `.pc` le compilateur `pccom` est appelé. Un fichier est généré dans un langage intermédiaire dépendant de l'architecture cible. Un autre fichier `.glob` est également généré pour chaque module `.pc`. Il contient des informations nécessaires à l'édition de liens.
2. Pour chaque fichier intermédiaire, le compilateur idoine est appelé en vue de générer un fichier `.o`.
3. Lorsque tous les modules `.pc` ont été compilés, le compilateur dispose de tous les fichiers `.glob` qui permettent l'allocation dynamique des variables parallèles globales. Le compilateur génère un dernier module qui assure cette allocation.

4. Ce fichier est ensuite compilé par le compilateur idoine.

5. L'éditeur de liens est enfin appelé sur tous les objets pour former l'exécutable final.

Les options du compilateur se classifient en plusieurs catégories.

A.1 Choix de l'architecture cible

Pour l'instant 3 architectures cibles sont disponibles et appelées par les options exclusives suivantes :

- `-sim` génère un exécutable mono/multi-process pour machine UNIX.
- `-cstar` génère un exécutable pour la Connection Machine CM-2.
- `-mpl` génère un exécutable pour la MasPar MP-1.

A.2 Choix de la sortie

Il s'effectue par les options exclusives suivantes :

- `-o` *nom_de_programme* réclame la génération d'un exécutable *nom_de_programme* par une édition de liens.
- `-c` ne génère que les codes relogeables `.o`

A.3 Choix de l'optimisation

Il s'effectue par les options exclusives suivantes :

- `-O`*optimisation* est passée tel quel au compilateur aval de la machine cible;
- `-g` invoque la génération de tables de symboles pour la mise au point.

Si aucun choix n'est fait, les modules `.pc` sont compilés sans optimisation ni directive de débogue tandis que le compilateur dépendant de la machine cible compile avec les directives de débogue; cela permet de mettre au point la génération du code intermédiaire.

A.4 Directive pour le préprocesseur

Les options passés au préprocesseur `cpp` sont les suivantes :

- `-D`*name*`[=`*value*`]` permet la définition d'une variable du préprocesseur;
- `-I`*dir* ajoute le répertoire *dir* à la fin de la liste des répertoires dans lesquels `cpp` recherche les fichiers inclus.

A.5 Options d'éditions de liens

Les options passées à l'éditeur de liens :

- `-L`*dir* ajoute le répertoire *dir* à la fin de la liste des répertoires dans lesquels l'éditeur de liens recherche les fichiers de bibliothèque déclarés par l'option `-l`
- `-l`*bib* ajoute la bibliothèque `lib`*bib*`.a` pour l'édition de liens.

A.6 Options diverses

- `-dryrun` montre les étapes de la compilation sans les exécuter.
- `-verbose` met le compilateur en mode bavard.
- `-debug` met le compilateur en mode mise au point : il ne supprime pas les fichiers intermédiaires et compile en mode bavard.
- `-warning` empêche le compilateur d'afficher les messages de mise en garde.
- `-pg` fait générer les informations d'étude de performance (voir **gprof(1)**).
- `-help` affiche la liste des options disponibles.

Annexe B

Mise au point

B.1 POMPC et dbx/dbxtool

B.1.1 Configuration

Les environnements de mise au point fonctionnent pour le moment en simulation sur Unix et sur la Connection Machine. Le programme Unix (BSD) `dbx` est l'outil recommandé pour la mise au point de programmes écrits en **POMPC**. Tous les modules du programme compilés avec l'option `-g` peuvent être facilement débogués à l'aide de trois mécanismes :

- l'association entre les instructions et les lignes de source **POMPC** dont elles sont issues. Cela permet de mettre des points d'arrêt dans le source **POMPC**.
- la possibilité de consulter les variables parallèles du programme.
- la possibilité d'afficher une représentation graphique de ces variables parallèles.

L'utilisation correct de `dbx` nécessite la définition d'un certain nombre d'abbréviations (alias), localisés dans le fichier `$POMPCLIB/.pcdbx`. On peut, par exemple définir les abbréviations suivants dans son `.cshrc` :

```
setenv POMPCLIB "/la/ou/il/faut"
alias pcdbx dbx -s $POMPCLIB/.pcdbx
alias pcdbxtool dbxtool -s $POMPCLIB/.pcdbxtool
```

Le contenu du fichier `.pcdbxtool` est présenté dans le tableau B.1. Le fichier `.pcdbx` est analogue mis à part qu'il ne contient pas les directives `button` et `unbutton` de la fin du fichier et qui sont particulières à `dbxtool`.

La bibliothèque graphique communique avec `pompx` ou `pompview` à l'aide du signal `USR2`. Il faut donc faire ignorer à `dbx/dbxtool` le signal `USR2` par la commande :

```
ignore USR2
```

B.1.2 Les commandes de mise au point parallèles

Ce fichier permet la création de nouvelles fonctions pour les variables parallèles :

- affichage du type d'une variable parallèle :
  `pwhatis` *pvar*
  ou
  `pw` *pvar*
  affiche la déclaration de la variable parallèle *pvar*, par exemple :
  `pixel char pvar;`

Table B.1 - contenu de `.pcdbxtool`.

```
alias pwhatis   "call pc_whatis(\"!:1\",!:1)"
alias pw        "call pc_whatis(\"!:1\",!:1)"
alias p print
alias pprint    "call pc_print(\"!:1\",&!:1,!:1)"
alias pp        "call pc_print(\"!:1\",&!:1,!:1)"
alias gprint    "call pc_gprint(\"!:1\",&!:1,!:1)"
alias gp        "call pc_gprint(\"!:1\",&!:1,!:1)"
alias dynamic   "call pc_setdynamicdisplay()"
alias pdisplay  "call pc_pprint(\"!:1\",&!:1,!:1)"
alias pd        "call pc_pprint(\"!:1\",&!:1,!:1)"
alias gdisplay  "display pc_gprint(\"!:1\",&!:1,!:1)"
alias gd        "display pc_gprint(\"!:1\",&!:1,!:1)"
alias zoom      "call pc_setcollectionzoom(\"!:1\")"
alias dynamic   "call pc_setdynamicdisplay()"
alias usr2      "ignore USR2;cont"
ignore USR2
unbutton next
unbutton step
button literal stopvi
button ignore make
button expand list
button expand pwhatis
button expand pprint
button expand gprint
button ignore dynamic
button expand zoom
button ignore usr2
```

- affichage des bornes d'une variable parallèle :

  `pprint` *pvar*

  ou

  `pp` *pvar*

  affiche les bornes inférieures et supérieures de la variable *pvar* ainsi que le nombre d'éléments de *pvar* actifs. On a par exemple :

  ```
  -36 <= pvar <= 42, for 124323 elements
  ```

  ou

  ```
  pvar = 42, for 124323 elements
  ```

  ou

  ```
  pvar = ####, for 0 element
  ```

- affichage graphique de la variable parallèle :

  `gprint` *pvar*

  ou

  `gp` *pvar*

  affiche la variable *pvar* dans une fenêtre graphique. Cette fonction ne marche que lorsqu'on emploie un des gestionnaires de fenêtres `SunView` ou `X11`. S'il s'agit de la première variable parallèle affichée par le debogueur, une fenêtre graphique est ouverte. La variable est ensuite affichée dans la fenêtre suivant une zone rectangulaire. Plusieurs valeurs par défaut

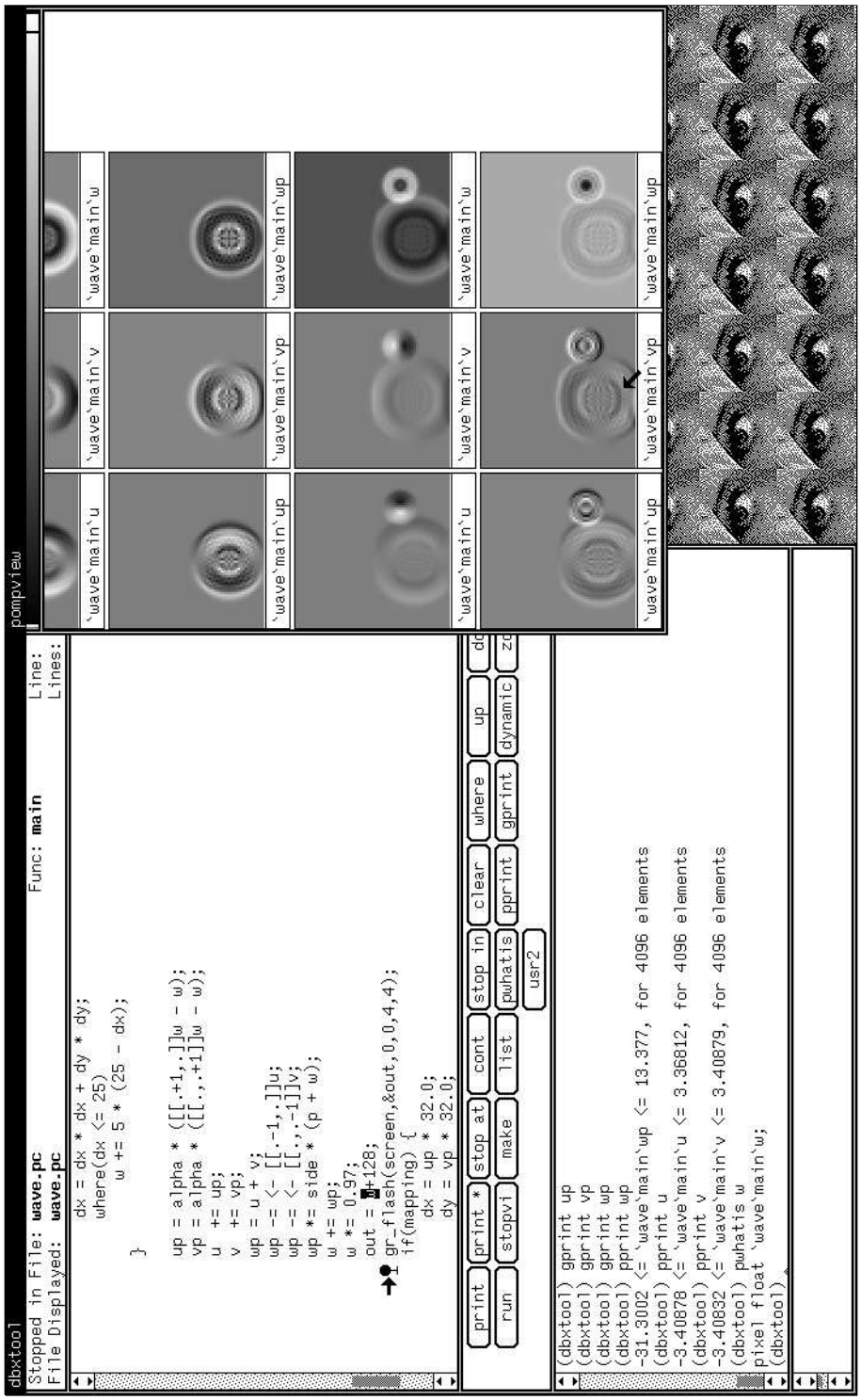

Figure B.1 - un exemple de mise au point graphique (en couleur sur l'écran de l'ordinateur).

sont employées pour réussir cet affichage :

1. le facteur de zoom en X et en Y. Chaque collection possède un tel facteur de zoom. Il peut être modifié sous `dbx` par la commande :

`zoom` *coll*
qui demande les facteurs de zoom en X et en Y.

2. la conversion des variables en une valeur de 25 à 220 pour être affiché sur un écran 8 bits. Plusieurs modes sont disponibles :

   (a) normalisation non signée. C'est le mode par défaut. Dans ce cas on renormalise la variable de telle sorte que la plus petite valeur active soit affichée dans le niveau 25 et la plus grande dans le niveau 220;

   (b) normalisation signée. On choisi cette fois-ci pour borne non plus

   $$[\min \texttt{pvar}, \max \texttt{pvar}]$$

   mais

   $$[\min(\min \texttt{pvar}, -\max \texttt{pvar}), \max(-\min \texttt{pvar}, \max \texttt{pvar})]$$

   (c) normalisation fixe.Les bornes sont définies par l'utilisateur.

   Un mode est associé à chaque type de donnée (`char`, `short`, `int`, `float`, `double`) et peut être modifié par la commande :
   `dynamic`
   qui pose des questions sur les désirs du programmeur.

Les variables inactives sont affichées en rouge.

- mise à jour de l'affichage des bornes d'une variable :
  `pdisplay` *pvar*
  ou
  `pd` *pvar*
  exécute la commande `pprint` à chaque point d'arrêt.

- mise à jour de l'affichage des bornes d'une variable :
  `gdisplay` *pvar*
  ou
  `gd` *pvar*
  exécute la commande `gprint` à chaque point d'arrêt.

B.1.3 Limitations

Expressions parallèles

L'affichage parallèle est limité à l'affichage de variables. On ne peut faire afficher une expression parallèle non atomique car l'évaluation de la fonction est faite par `dbx` qui ne connait rien au caractère parallèle de la variable.

Fonctions surchargées

Les fonctions surchargées peuvent posséder un autre nom dans le code exécutable. Les règles de construction pour la fonction *fonction* sont les suivantes :

- la première instance de la fonction conserve son nom *fonction*,

- la $n^{\text{ème}}$ fonction reçoit le nom `pc_over_`*n*`_`*fonction*.

Table B.2 - les problèmes de configurations du compilateur.

Cible	*Problème/diagnostique*
*	`pcc: Command not found.` La variable d'environnement `PATH` doit contenir le répertoire contenant les exécutables pour la machine courante des programmes `pcc`, `pccom` et `pompx` ou `pompview`.
*	`Please set the envvar POMPCLIB to the directory name of the library of POMPC!` la variable d'environnement `POMPCLIB` doit contenir la position du répertoire contenant les librairies de **POMPC**.
MP	`mpl: No such file or directory` La variable d'environnement `PATH` doit contenir le répertoire contenant les exécutables du compilateur MPL.
CM	`cs: No such file or directory` La variable d'environnement `PATH` doit contenir le répertoire contenant les exécutables du compilateur C*.

B.2 En attendant mieux...

Il existe une différence entre le langage tel qu'il est présenté dans le corps de ce document et la réalité des compilateurs.

Nous décrivons dans cette section un certain nombre de problèmes qu'un débutant peut rencontrer lors de l'utilisation de **POMPC**. Ils sont classés en différentes catégories :

- les problèmes de configuration sur différentes architectures cibles;
- les erreurs classiques du programmeur débutant;
- les limitations du compilateur :
  - en général,
  - pour la simulation sur machine UNIX,
  - pour la Connection Machine,
  - pour la MasPar.

Lorsque c'est possible, nous indiquons comment éviter ces problèmes.

B.2.1 Problèmes de configuration

Le tableau B.2 récapitule les problèmes de configurations du compilateur. Le champs *cible* indique l'architecture cible :

- `*` concerne toutes les architectures cibles;
- `sim` concerne le simulateur, sur machine UNIX;
- `CM` concerne la Connection Machine;
- `MP` concerne la MasPar;

Le tableau B.3 récapitule les problèmes de configuration à l'exécution d'un programme.

Table B.3 - les problèmes de configurations à l'exécution d'un programme.

Cible	*Problème/diagnostic*
`sim`	`Segmentation fault` Eventuellement un dépassement de la taille de la pile. La commande `limit` de `csh` permet de changer la taille maximum de la pile.
`*`	`no window manager is running!` le programme utilise les bibliothèques graphiques et il n'a pu identifier aucun gestionnaire de fenêtre. Peut être faut il positionner correctement la variable d'environnement `DISPLAY` sous `X`.

B.2.2 Les erreurs du débutant

Nous présentons ici les erreurs classiques du débutant sous **POMPC**.

1. Oubli de l'inclusion du fichier `pompc.h` en tête de chaque module **POMPC**.
2. Oubli de l'inclusion des fichiers d'en-tête (principalement `math.h` et `pc_graphic.h`) dans lesquels se trouvent les prototypes de fonctions utilisées (qui sont parfois surchargées).
3. Utilisation d'une collection dynamique sans l'avoir démarrée.
4. Déclaration de variables d'une collection dans la fonction qui démarre cette collection.
5. Calcul dans une collection démarrée dans la même fonction. Les calculs sur les variables parallèles nécessitent parfois des variables temporaires qui sont allouées en tête de la fonction. Si la collection est démarrée ultérieurement, l'allocation des variables est réalisée avec une mauvaise taille.
6. Problème sur la précédence des opérateurs :
   - l'expression `a + b << c` signifie `(a + b) << 3`;
   - l'expression `+<- a * b` signifie `(+<- a) * b`;
   - l'expression `a == b & 2` signifie `(a == b) & 2` soit `0` de toute manière.
7. Confusion entre les réduction associatives unaires et binaires : `s +<- k` est différent de `s = +<- k`, car elle tient compte de la valeur initiale de `s`;
8. Confusion entre les réarrangements d'adresse : `[[x]]a` est différent de `[x]a`. En effet, dans l'expression `[x]a`, x doit avoir été calculé avec la fonction `pc_build_address`.
9. Problème d'activité avec les réceptions parallèles : un PV source sélectionné inactif renvoie une valeur indéterminée.
10. Problème d'activité avec les émissions parallèles : un PV destination inactif peut recevoir des messages et modifier sa mémoire locale.
11. Problème d'activité avec `everywhere`. On peut modifier un PV inactif et lire une valeur d'un PV inactif une valeur incohérente. La recopie des paramètres d'une fonction se fait en tenant compte de l'activité.
12. Problème du `return` parallèle qui arrête ou non l'exécution de la fonction suivant le contexte.

B.2.3 Les limitations du compilateur

Les limitations du compilateur exposées dans le tableau B.4 sont en principe provisoires et doivent disparaître dans les futures versions du compilateur.

Table B.4 - les limitations du compilateur.

Cible	*limitations*
`*`	Le préprocesseur **C** de **POMPC** est celui de la machine UNIX hôte (habituellement `/usr/lib/cpp`) qui n'est pas toujours ANSI.
`*`	Les variables globales ne peuvent être initialisées.
`sim` `MP`	`alloca`, l'usage de la fonction `alloca` est déconseillée dans une fonction qui alloue des variables parallèles.
`sim`, `MP`	Les pointeurs scalaires ne peuvent référencer des éléments de tableaux parallèles ni des champs de structures
`CM`	Les pointeurs parallèles n'existent pas.
`CM` `MP`	Les géométries doivent être exprimées en puissances de 2.
`MP`	Les géométries doivent être multiple de la géométrie de base.
`CM`	Les communications sur grille ignorent le rebouclage circulaire.

Annexe C

Les bibliothèques

Les bibliothèques à ce jour disponibles dans **POMPC** sont les suivantes :

- `libpc_c.a`, la bibliothèque indispensable au fonctionnement de **POMPC**.
- `libpc_gr.a`, la bibliothèque graphique de **POMPC**.
- `libpc_m.a`, la bibliothèque mathématique parallèle de **POMPC**.

Elles sont toutes utilisées systématiquement par lors de l'édition de liens.

C.1 La bibliothèque standard : libpc_c.a

La librairie `libpc_c.a` contient les fonctions vitales de **POMPC**. Elle contient d'une part des fonctions de gestion des collections, et d'autre part les fonctions de base pour les communications.

C.1.1 La gestion des collections

pc_start_collection: on ne peut se servir d'une collection (et de ses variables parallèles) qu'après avoir défini sa taille, à l'aide de la fonction `pc_start_collection`. Cette fonction possède 5 surcharges :

```
overload pc_start_collection;
void pc_start_collection(collection c,int s,int nd,int *d,int *p);
void pc_start_collection(collection c,int s,int nd,int *d);
void pc_start_collection(collection c,int s);
void pc_start_collection(collection c,int nd,int *d,int *p);
void pc_start_collection(collection c,int nd,int *d);
```

Les 5 versions de la fonction ne sont que différentes formes d'appels à la même fonction qui est la première, celle qui contient le plus d'argument. Lorsqu'un argument manque il est remplacé par une valeur adéquate. Nous ne décrirons que la fonction la plus complète, avec cet exemple :

```
collection volume;
....

int dim[3];
int pref[3];
int size;
```

```
int nb_dim;

dim[0] = sizex;
dim[1] = sizey;
dim[2] = sizez;
pref[0] = 3;
pref[1] = 4;
pref[2] = 1;
size = sizex * sizey * sizez;
nb_dim = 3;

pc_start_collection(volume,size,nb_dim,dim,pref);
```

Les arguments de `pc_start_collection` sont :

1. `volume` : le pointeur sur le descriptif de la collection,
2. `size` : le nombre d'éléments de la collection,
3. `nb_dim` : le nombre de dimensions de la collection,
4. `dim` : un tableau de `nb_dim` entiers contenant la taille de chacune des dimensions de la collection
5. `pref` : un tableau de `nb_dim` entiers contenant des préférences pour le placement des processeurs virtuels sur les processeurs physiques.

Le tableau de préférence sert à définir de manière indépendante du nombre de processeurs physiques quelques caractéristiques de placement.

Comme les programmes en **POMPC** sont *a priori* indépendants de la topologie du réseau d'interconnexion de la machine cible, le tableau de préférence sert uniquement à coder des indications du programmeur sur l'importance relative des mouvements de données suivant chaque direction. Le programmeur peut ainsi ordonner les dimensions en classe de besoin aux ressources de communication.

Pour chaque collection on veut pouvoir classer les d dimensions en c classes. On veut ensuite pouvoir donner un ordre entre les classes et un ordre à l'intérieur des classes. Cet ordre peut être noté $\prec$.

Chaque classe contient des dimensions qui ont des besoins de communications à peu près équivalents, mais qu'on peut toutefois ordonner. Deux classes différentes contiennent des dimensions nécessitant des besoins très différents. Cet ordre est noté $\ll$.

Le tableau de préférence permet de coder un classement des dimensions :

$$P_k \ll P_l = P_m \prec P_i \ll P_j$$

Dans l'exemple précédent, on a voulu codé le classement :

$$P_2 \ll P_0 \prec P_1 \text{ soit } 1 \ll 3 \prec 4$$

Ce qui signifie qu'on désire faire beaucoup de communications suivant la dimension 2, et à peu prêt le même volume de communication sur les dimensions 0 et 1 avec un peu plus pour la dimension 1. La relation $\prec$ est définie par :

$$n \prec p \Leftrightarrow n + 1 = p$$

alors que la relation $\ll$ est définie par :

$$n \ll p \Leftrightarrow n + 1 < p$$

Pour illustrer cette expression de la préférence, étudions le placement de l'exemple précédent pour un hypercube : le nombre de processeurs physiques est une puissance de 2 (2^n) et toutes les dimensions physiques sont équivalentes.

La collection est un pavé de plusieurs dimensions dont chaque axe peut être découpé sur un axe physique. On obtient un assemblage de 2^n sous-pavés. Dans notre cas (dimension 3) les 2^n processeurs physiques seront vus par la collection comme une grille 3D dont les cotés sont des puissances de 2 :

$$2^{n_x} \times 2^{n_y} \times 2^{n_z} = 2^n,$$

avec

$$n_x + n_y + n_z = n$$

Le problème du placement d'une collection consiste à distribuer les n bits d'adresse physique sur les *nb_dim* dimensions. Le tableau *pref* va servir à faire cette distribution de manière indépendante la taille physique de la machine : c'est un tableau de priorité.

Si ce tableau ne contient que des 0 ou bien qu'un pointeur nul est passé en paramètre, l'utilisateur indique que la répartition physique ne lui importe peu.

Sinon, on trouve des zones connexes dans l'image du tableau. Dans notre cas, il y a 2 zones connexes :

- 3 et 4
- 1

Pour chaque zone connexe (des plus fortes valeurs vers les plus faibles) on distribue le maximum de bit d'adresse physique (tant que $2^{n_i} \leq dim[i]$) en commençant par les valeurs les plus fortes de chaque zone connexe. On épuise ainsi tous les bits d'adresse physique sur chaque zone connexe une par une. Exemples :

```
sizex = 48
sizey = 48
sizez = 48
```

- si $n = 8$ alors $nx = 4$, $ny = 4$ et $nz = 0$
  il s'agit d'une grille de $16 \times 16 \times 1$ processeurs physiques contenant chacun un pavé de $3 \times 3 \times 48$.
- si $n = 9$ alors $nx = 4$, $ny = 5$ et $nz = 0$
  il s'agit d'une grille de $16 \times 32 \times 1$ processeurs physiques contenant chacun un pavé de $3 \times 2 \times 48$.
- si $n = 12$ alors $nx = 6$, $ny = 6$ et $nz = 0$
  il s'agit d'une grille de $64 \times 64 \times 1$ processeurs physiques contenant chacun un pavé de $1 \times 1 \times 48$.
- si $n = 13$ alors $nx = 6$, $ny = 6$ et $nz = 1$
  il s'agit d'une grille de $64 \times 64 \times 2$ processeurs physiques contenant chacun un pavé de $1 \times 1 \times 24$.

Le tableau de préférence est important lorsqu'on décide de privilégier ou non certaines dimensions. Si on sait que les communications se feront principalement sur un axe donné, on cherche à ce que cet axe soit placé si possible totalement en processeur virtuel : chaque ligne sur cet axe est localisé physiquement sur le même processeur. Dans ce cas on lui donnera un poids très faible dans le tableau de préférences. Donner 2 valeurs de préférence identiques à deux axes indique qu'on désire avoir des répartitions ne privilégiant pas tel ou tel axe. Le placement conduit à des grilles carrées (à un facteur 2 prés). L'incrémentation d'une des préférences permet de forcer la répartition de la dernière puissance de 2.

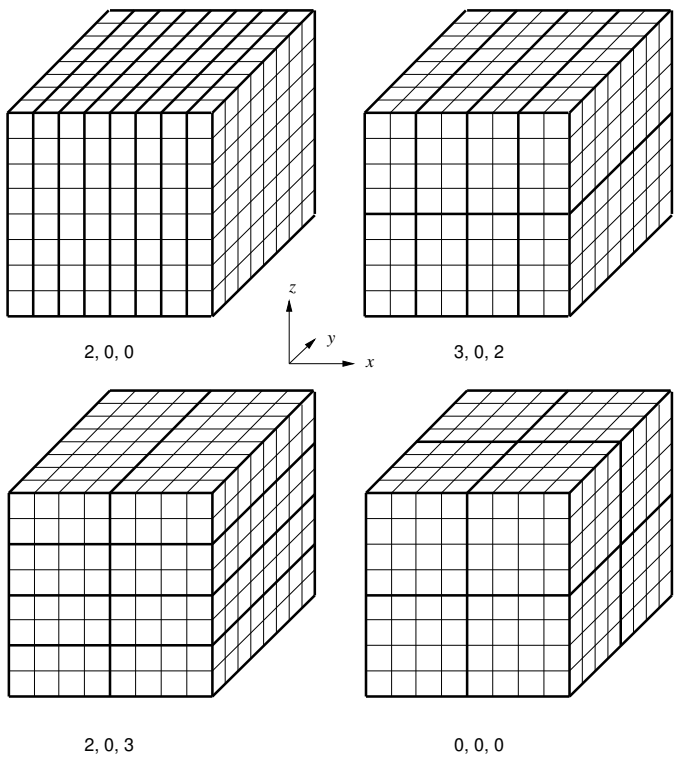

Figure C.1 - le placement d'une grille $8 \times 8 \times 8$ sur 8 processeurs.

Si on désire ne pas s'encombrer de ces problèmes, on peut passer un pointeur nul à la place de *pref*, ou mieux on utilise une des surcharges sans tableau de préférence.

La figure C.1 montre le placement d'une grille $8 \times 8 \times 8$ sur un hypercube de 8 processeurs où à chaque exemple est associé le tableau de préférence.

pc_coord : cette fonction retourne une variable parallèle contenant pour chaque processeur virtuel actif sa position dans la $\texttt{n}^{\texttt{ième}}$ dimension de la géométrie courante de la collection appelant. Elle est surchargée en deux versions :

```
collection generic int pc_coord(int n);
collection generic int pc_coord(collection generic,int n);
```

Cela permet d'écrire :

```
pixel int x;

x = pc_coord(0);
where(pc_coord(pixel,1)) {
    ...
}
```

Dans le premier appel, la collection concernée est déterminée par le typage de l'affectation. Cela ne peut pas se faire dans le deuxième appel, on peut utiliser dans ce cas la deuxième forme, où la collection impliquée est explicitement passée en paramètre.

pc_build_address : cette fonction construit une ou plusieurs adresses dans la géométrie d'une collection. Elle possède 4 surcharges :

```
collection c int pc_build_address(void);
collection c int pc_build_address(collection c);
int pc_build_address(collection d,int x,...);
collection c int pc_build_address(collection d,c int x,...);
```

Les deux premières permettent de récupérer pour chaque processeur virtuel de la collection l'adresse locale du PV. La troisième permet de calculer l'adresse d'un PV particulier de la collection `d` en fonction d'un n-uplet $(x, y, ...)$ représentant les coordonnées du PV sélectionné dans la géométrie de la collection. La dernière, enfin, effectue le même calcul en parallèle dans la collection `c`, le n-uplet appartient alors à cette collection.

pc_set_torus : cette fonction définit pour chaque dimension d'une collection si les communications seront circulaires. Elle admet pour prototype :

```
int pc_set_torus(collection c,int mask);
```

Chaque bit du masque `mask` définit le comportement d'une dimension :

```
pc_set_torus(pixel,5);
```

déclare que seules les dimensions numéro 0 et 2 sont circulaires, car $5 = 2^2 + 2^0$. Par défaut, le masque est à la valeur -1. Cette fonction renvoie l'ancienne valeur du masque.

pc_get_torus : cette fonction renvoie la valeur courante du masque positionné pour une collection par `pc_set_torus`. Elle admet pour prototype :

```
int pc_get_torus(collection c);
```

C.1.2 Les utilitaires

pc_malloc : l'allocation mémoire dans une collection est réalisée par `pc_malloc` :

```
collection generic void * pc_malloc(unsigned n);
```

La valeur de `n` représente le nombre d'octets à allouer sur chaque PV.

pc_free : la libération de la mémoire allouée par `pc_malloc` se fait par `pc_free` :

```
void pc_free(collection generic void *p);
```

pc_fprintf, pc_printf: les sorties textes peuvent être réalisé à l'aide de `pc_fprinf` et de `pc_printf` :

```
void pc_fprintf(FILE *f,char *format,...);
void pc_printf(char *format,...);
```

Ces fonctions acceptent comme argument un mélange de variables scalaires et de variables parallèles. Il faut néanmoins que toutes les variables parallèles appartiennent à la même collection. Pour chaque PV actif de cette collection, on génère une chaîne de caractères à partir du format, des variables scalaires et des éléments locaux des variables parallèles. La syntaxe de la chaîne de format est analogue à celle de `printf`. La distinction entre variable parallèle et variable scalaire se fait à l'aide d'un caractère `p` dans le format situé juste avant le caractère qui spécifie le type de la variable à afficher.

pc_read : la lecture en binaire d'une variable parallèle d'un fichier se fait par :

```
int pc_read(int f,collection generic void *b,unsigned int size);
```

La variable `f` est le numéro du fichier ouvert en lecture duquel on va lire la donnée. On lit `size` octets consécutifs par PV de la collection `generic`. La première série de `size` octets est destiné au PV $x = 0, y = 0, ...$ tandis que la seconde est destinée au PV $x = 1, y = 0, ...$ La fonction `pc_read` renvoie un scalaire égal au nombre total d'octets lus dans le fichier. La fonction `pc_read` ne tient pas compte de l'activité de la collection.

pc_write: l'écriture en binaire d'une variable parallèle dans un fichier se fait par :

```
int pc_write(int f,collection generic void *b,unsigned int size);
```

La variable `f` est le numéro du fichier ouvert en écriture dans lequel on va écrire la donnée. On écrit `size` octets consécutifs par PV de la collection `generic` en commençant par le PV $x = 0, y = 0, ...$, puis le PV $x = 1, y = 0, ...$ La fonction `pc_write` renvoie un scalaire égal au nombre total d'octets écrits dans le fichier. La fonction `pc_write` ne tient pas compte de l'activité de la collection.

pc_random : le tirage aléatoire d'une variable parallèle se fait à l'aide de la fonction `pc_random` qui possède deux surcharges :

```
collection generic int pc_random(void);
collection generic int pc_random(collection generic);
```

Elle renvoie dans chaque PV de la collection `generic` un entier tiré au hasard.

C.2 La bibliothèque graphique

La bibliothèque graphique est chargée en standard et contient les fonctions permettant l'affichage d'une image organisée en une variable parallèle à deux dimensions. Elle contient d'une part des fonctions graphiques non parallèles et d'autre part des fonctions parallèles. Le bon emploi de la bibliothèque graphique nécessite l'inclusion dans les modules utilisant cette bibliothèque du fichier d'en-tête `pc_graphic.h` qui contient tous les prototypes et les surcharges des fonctions. La bibliothèque graphique fonctionne avec deux programmes:

- `pompview` lorsqu'on travaille sur le gestionnaire de fenêtre `SunView`;
- `pompx` lorsqu'on travaille sur le gestionnaire de fenêtre `X11`.

La bibliothèque graphique sait reconnaître automatiquement le gestionnaire de fenêtre sous lequel le programme travaille pour lancer le bon type de programme graphique. Si on ne travaille pas sous un gestionnaire de fenêtre, la bibliothèque signale ce fait par un message d'avertissement, et continue de fonctionner. Cela permet de faire tourner en batch des programmes graphiques sans avoir besoin de les modifier.

C.2.1 Organisation en fenêtres

La fonction `pc_opendisplay` permet l'ouverture d'une fenêtre graphique. Elle possède les deux surcharges suivantes :

```
pc_display pc_opendisplay(void);
pc_display pc_opendisplay(int X,int Y);
```

Cette fonction ouvre une fenêtre graphique (gérée par `pompview` ou par `pompx` de taille `X` par `Y` ou par défaut 512 par 512. Cette fonction renvoie un descriptif de fenêtre de type `pc_display` qui servira à identifier la fenêtre graphique à laquelle on s'adresse.

La sélection d'une fenêtre peut se faire de deux manières :

1. en donnant explicitement le descriptif de fenêtre comme premier argument de l'appel graphique.
2. en utilisant la fenêtre courante. Initialement il n'y a pas de fenêtre courante. Un appel graphique ouvre alors une fenêtre qui devient fenêtre courante, ou bien, la première fenêtre ouverte devient la fenêtre courante. On peut également modifier la fenêtre courante :

   ```
   pc_display pc_setcurrentdisplay(pc_display d);
   ```

   La fenêtre décrite par `d` devient la nouvelle fenêtre courante et la fonction renvoie le descriptif de l'ancienne fenêtre courante. On peut également récupérer le descriptif de fenêtre courante par la fonction :

   ```
   pc_display pc_getcurrentdisplay(void);
   ```

   On peut également gérer la fenêtre courante dans une pile par les fonctions :

   ```
   void pc_pushcurrentdisplay(pc_display d);
   pc_display pc_popcurrentdisplay(void);
   ```

   `pc_pushcurrentdisplay` sauve le descriptif de fenêtre courante dans une pile et `d` devient le nouveau descriptif de fenêtre courante, tandis que la fonction `pc_poptcurrentdisplay` dépile l'ancien descriptif de fenêtre courante.

Toutes les autres fonctions graphiques font référence à une fenêtre graphique courante ou explicite. Elles sont donc toutes surchargées avec le descriptif de fenêtre optionnel. Celui-ci est systématiquement en premier argument s'il est utilisé.

Pour la suite de la librairie, nous ne présentons que les surcharges utilisant ce paramètre.

C.2.2 Les fonctions scalaires

pc_closedisplay : on ferme une fenêtre de descriptif `d` par la commande :

```
pc_closedisplay(d);
```

pc_getdepth : la fonction `pc_getdepth` renvoie le nombre de bits de la fenêtre.

pc_setcmap : permet de modifier la palette graphique de la fenêtre choisie :

```
int pc_setcmap(pc_display d,int type,int cycle,
    int start,int clip);
```

Cette fonction admet 4 paramètres qui permettent la construction d'une palette :

1. `type` choisit le type de la palette parmi :
   (a) `BLACK_AND_WHITE` pour une palette en niveau de gris. La palette de base va du noir (niveau 0) au blanc (niveau 255).
   (b) `RAINBOW` pour une palette arc-en-ciel qui va du rouge au rouge en passant successivement par l'orange, le jaune, le vert, le bleu et le violet.
   (c) `ALTERNATE_RAINBOW` pour une palette arc-en-ciel comme la précédente avec les niveau impairs plus foncés.
   (d) `RED_BLUE` cette palette est faite pour afficher des valeurs signées. Le niveau 0 est bleu, le niveau 128 est blanc et le niveau 255 est rouge.
   (e) `USER_DEFINED` pour la dernière palette qui a été définie par l'appel à la fonction `pc_setcolor`

   On peut modifier (sur les écrans 8 bits) le choix de cette palette de base en frappant au clavier les touches 1, 2, 3, 4 et 0 dans la partie de la fenêtre qui contient la palette.
2. `cycle` permet de modifier cette palette suivant un rapport de périodicité. Si `cycle` vaut 0, les palettes précédemment décrites ne sont pas modifiées. Si `cycle` vaut $n > 0$ la palette est comprimé d'un facteur 2^n. Si `cycle` vaut $n < 0$ la palette est étirée d'un facteur 2^n. D'une manière générale la périodicité de la palette est de 256×2^{-n} niveaux. Pour la palette `RED_BLUE`, `cycle` agit uniquement sur la transition du bleu au blanc puis au rouge au voisinage du niveau 128. Plus `cycle` est élevé plus cette transition est rapide. On peut modifier ce paramètre de manière interactive par les touches `-` et `=` qui décrémente et incrémente la valeur de `cycle`.
3. `start` permet de faire une permutation circulaire de `start` niveaux vers le haut de la palette choisie. De manière interactive, les touches `h` et `l` permettent de décrémenter et d'incrémenter la valeur de `start`.
4. `clip` permet de faire ressortir un niveau particulier de la palette. Si `clip` vaut -1, aucun niveau ne ressort. Si `clip` vaut une valeur de 0 à 255, le niveau correspondant prend alors une couleur qui se détache. De manière interactive, les touches `j` et `k` permettent de décrémenter et d'incrémenter la valeur de `clip`. qui évolue périodiquement dans l'intervalle $[-1, 255]$.

En vue de chercher une compatibilité avec les écrans monochromes. ces palettes sont retraduites en palettes binaires. Les styles `BLACK_AND_WHITE` et `RAINBOW` deviennent équivalents tandis que le style `ALTERNATE_RAINBOW` est la transformation de la palette `RAINBOW` avec un ou-exclusif sur le bit de poids faible de la couleur. Les autres paramètres prennent les significations suivantes :

- *cycle* représente la périodicité de la palette (de 0 a 7) :
  - *cycle* = 0 : le bit de poids fort est significatif.
  - *cycle* = 7 : le bit de poids faible est significatif.
- *offset* représente le déphasage.
- *clip* fait ressortir un seul niveau :
  - *clip* = -1 (inactif)
  - *clip* = 0 - 255. tout est à noir sauf le niveau correspondant.

pc_setcolor : la définition d'une palette totalement définissable se fait par :

```
int pc_setcolor(pc_display d,char red[256],
    char green[256],char blue[256]);
```

Cet appel remplit la palette utilisateur de `pompview/pompx` et appelle la fonction :

```
pc_setcmap(d,USER_DEFINED,0,0,-1);
```

qui utilise cette palette sans modification.

pc_mouse : on peut réclamer une entrée par la souris dans la fenêtre graphique par l'appel :

```
int pc_mouse(pc_display d,int *X,int *Y);
```

Cette fonction attend qu'on presse sur un bouton de la souris. Elle positionne dans `*X` et `*Y` les coordonnées du point sélectionné et renvoie le numéro du bouton pressé :

- 1 pour le bouton gauche;
- 2 pour le bouton du milieu;
- 3 pour le bouton de droite;

pc_ismouse : on peut réclamer une entrée par la souris dans la fenêtre graphique par l'appel :

```
int pc_ismouse(pc_display d,int *X,int *Y);
```

Cette fonction indique si on a pressé sur la souris depuis le dernier appel à cette fonction; elle positionne dans `*X` et `*Y` les coordonnées du point sélectionné et renvoie le numéro du bouton pressé :

- 0 si aucun bouton n'a été enfoncé;
- 1 pour le bouton gauche;
- 2 pour le bouton du milieu;
- 3 pour le bouton de droite;

C.2.3 Les fonctions d'affichages parallèles

On utilise pour l'affichage de variables parallèles la convention classique qui veut que le pixel (0, 0) se trouve en haut à gauche de la fenêtre.

pc_show : l'affichage d'une variable parallèle se fait par la fonction `pc_show` qui possède 40 surcharges. La fonction type est la suivante :

```
void pc_show(pc_display d,collection c const type v,
    int ox,int oy,int zx,int zy);
```

où `type` est à choisir parmi `char`, `short`, `int`, `float` et `double`. La fenêtre `d` est facultative et on utilise la fenêtre par défaut lorsque elle n'est pas précisée. On peut remplacer `zx` et `zy` par `zoom` lorsqu'on veut le même facteur de zoom en x et en y. On peut également omettre ces facteurs de zoom qui sont remplacés par des valeurs par défaut associées à chaque collection, modifiables par la fonction surchargée :

```
void pc_setcollectionzoom(collection col);
void pc_setcollectionzoom(collection col,int Z);
void pc_setcollectionzoom(collection col,int Zx,int Zy);
```

L'appel sans argument pose des questions sur `stderr` et attend des réponses sur `stdin`. Le deuxième appel positionne les zooms en x et en y à la valeur de `Z`. Le dernier appel modifie les deux zooms suivant les valeurs de `ZX` et de `ZY`.

Les arguments `ox` et `oy` précisent un décalage de la position du pixel (0,0) de la collection dans le repère de la fenêtre. Ces arguments sont optionnels, si aucun facteur de zoom n'est donné à `pc_show`. Ils prennent alors la valeur 0. La fonction doit convertir une variable d'un certain `type` en un entier 8 bits compris entre 25 et 220. Pour chaque type, plusieurs modes sont disponibles :

1. normalisation non signée. C'est le mode par défaut. Dans ce cas on renormalise la variable de telle sorte que la plus petite valeur active soit affichée dans le niveau 25 et la plus grande dans le niveau 220.

$$[\min \texttt{pvar}, \max \texttt{pvar}] \rightarrow [20, 220]$$

2. normalisation signée. On choisi cette fois-ci pour borne non plus mais

$$[\min(\min \texttt{pvar}, -\max \texttt{pvar}), \max(-\min \texttt{pvar}, \max \texttt{pvar})] \rightarrow [20, 220]$$

   Ce mode est interessant car 0 est toujours convertit en le niveau 128.

3. normalisation fixe. Les bornes sont définies par l'utilisateur.

Un mode est associé à chaque type de donnée (`char`, `short`, `int`, `float`, `double`) et peut être modifié par la fonction :

```
void pc_setdynamicdisplay(void);
```

qui pose des questions sur les désirs de l'utilisateur. On peut aussi faire ce changement à l'aide des fonctions surchargées :

```
void pc_setdynamicdisplaychar(double min,double max);
void pc_setdynamicdisplaychar(int n);
void pc_setdynamicdisplayshort(double min,double max);
void pc_setdynamicdisplayshort(int n);
void pc_setdynamicdisplayint(double min,double max);
void pc_setdynamicdisplayint(int n);
void pc_setdynamicdisplayfloat(double min,double max);
void pc_setdynamicdisplayfloat(int n);
void pc_setdynamicdisplaydouble(double min,double max);
void pc_setdynamicdisplaydouble(int n);
```

Pour chaque type, il existe une fonction avec deux surcharges :

1. celle au paramètre entier `n` permet de passer au mode automatique non-signé par la valeur 0, au mode automatique signé par la valeur 1 et au mode utilisateur par la valeur 2 (on emploie alors les dernières valeurs spécifiées).
2. celle aux deux paramètres `double min` et `max` qui positionne le type en mode utilisateur avec les bornes `min` et `max`.

Les pixels correspondant aux PVs inactifs sont affichés dans le niveau 2.

pc_flash : l'affichage d'une variable parallèle `char` peut se faire par la fonction `pc_flash` qui possède 8 surcharges. La fonction type est la suivante :

```
void pc_flash(pc_display d,collection c const char v,
    int ox,int oy,int zx,int zy);
```

Elle n'est valable que pour des variables de type caractère. Elle ne fait aucune conversion et a l'avantage d'être rapide. Les pixels inactifs sont quand même affichés.

C.3 La bibliothèque mathématique

La bibliothèque mathématique parallèle est récapitulée dans le tableau C.1.

Les fonctions mathématiques classiques sont surchargées dans leur utilisation parallèle. Il existe la surcharge avec des arguments parallèles flottants simple et double précisions. Les prototypes de ces fonctions sont contenus dans le fichier `math.h`. Ce sont toutes des fonctions `spmd` qui ne brisent pas l'extension de la boucle de virtualisation.

Table C.1 - fonctions mathématiques de `libm.a`.

lignes trigonométriques	`spmd c double sin(c double x)`
	`spmd c double cos(c double x)`
	`spmd c double tan(c double x)`
	`spmd c float sin(c float x)`
	`spmd c float cos(c float x)`
	`spmd c float tan(c float x)`
lignes trigonométriques inverses	`spmd c double acos(c double x)`
	`spmd c double asin(c double x)`
	`spmd c double atan(c double x)`
	`spmd c double atan2(c double y, c double x)`
	`spmd c float acos(c float x)`
	`spmd c float asin(c float x)`
	`spmd c float atan(c float x)`
	`spmd c float atan2(c float y, c float x)`
lignes trigonométriques hyperboliques	`spmd c double cosh(c double x)`
	`spmd c double sinh(c double x)`
	`spmd c double tanh(c double x)`
	`spmd c float cosh(c float x)`
	`spmd c float sinh(c float x)`
	`spmd c float tanh(c float x)`
lignes exponentielles	`spmd c double exp(c double x)`
	`spmd c double log(c double x)`
	`spmd c double log10(c double x)`
	`spmd c double pow(c double x,c float y)`
	`spmd c double sqrt(c double x)`
	`spmd c float exp(c float x)`
	`spmd c float log(c float x)`
	`spmd c float log10(c float x)`
	`spmd c float pow(c float x,c double y)`
	`spmd c float sqrt(c float x)`
gestion des arrondis	`spmd c double floor(c double x)`
	`spmd c double ceil(c double x)`
	`spmd c double fmod(c double x,c double y)`
	`spmd c float floor(c float x)`
	`spmd c float ceil(c float x)`
	`spmd c float fmod(c double x,c float y)`

Table des matières

Liste des tableaux

Table des figures

Index

PUBLICATIONS DU LIENS

Les publications précédées par un '◇' sont disponibles par ftp anonymous sur la machine spi.ens.fr (129.199.104.3) dans le répertoire pub/reports/liens.

Publications mentioned by '◇' are available by anonymous ftp on the machine spi.ens.fr (129.199.104.3) in the directory pub/reports/liens.

90 - 1	M.P. GASCUEL A. VERROUST C. PUECH	Animation with Collisions of Deformable Articulated Bodies
90 - 2	B. VIROT	Parallelization of the Simulated Annealing Algorithm Application to the Placement Problem
90 - 3	J.P.THIRION	Tries : Data Structures Based on Boolean Representation for Ray Tracing
90 - 4	J.P. THIRION	Interval Arithmetic for High Resolution Ray Tracing
90 - 5	F.P. PREPARATA J.S. VITTER M. YVINEC	Output-Sensitive Generation of the Perspective View of Isothetic Parallelepipeds
90 - 6	L. BOUGÉ P. GARDA	Towards a Semantic Approach to SIMD Architectures and their languages
90 - 7	L. PUEL A. SUÁREZ	Compiling Pattern Matching by Term Decomposition
90 - 8	D. DURE	Simulation Multi-Mode de Circuits VLSI (Thèse)
90 - 9	P.L. CURIEN	Substitution up to Isomorphism
90 -10	P.L. CURIEN G. GHELLI	Coherence of Subsumption
90 -11	F. FAGES	A New Fixpoint Semantics for General Logic Programs Compared with the Well-Founded and the Stable Model Semantics
90 -12	A. BOUVEROT	Pliage/Dépliage et Extraction de Programmes Logiques: Présentation Comparée
90 -13	L. BOUGÉ	On the Semantics of Languages for Massively Parallel SIMD Architectures
90 -14	K. BRUCE R. DI COSMO G. LONGO	Provable Isomorphisms of Types
90 -15	F. FAGES	Consistency of Clark's Completion and Existence of Stable Models
90 -16	J.D. BOISSONNAT O. DEVILLERS R. SCHOTT M. TEILLAUD M. YVINEC	Applications of Random Sampling to On-line Algorithms in Computational Geometry

	90 -17	J.P. THIRION	Utilisation de la Cohérence des Rayons Lumineux pour le Lancer de Rayons (Thèse)
	90 -18	M. POCCHIOLA E. KRANAKIS	Camera Placement in Integer Lattices
	90 -19	R. M. AMADIO	Domains in a Realizability Framework
	90 -20	G. LONGO	Notes on the Foundation of Mathematics and of Computer Science
	90 -21	G. LONGO E. MOGGI	Constructive Natural Deduction and its "ω"-set Interpretation
	90 -22	M.P. GASCUEL	Déformations de Surfaces Complexes : Techniques de Haut Niveau pour la Modélisation et l'Animation (Thèse)
	90 -23	M. POCCHIOLA	Trois Thèmes sur la Visibilité : Enumération Optimisation et Graphique 2D (Thèse)
	90 -24	Y. LAFONT A. PROUTÉ	Church-Rosser Property and Homology of Monoids
	90 -25	P.H. CHEONG	Compiling Lazy Narrowing into Prolog
	90 -26	P.H. CHEONG L. FRIBOURG	Efficient Integration of Simplification into Prolog
	90 -27	A. VERROUST	Etude de Problèmes liés à la définition, la Visualisation et l'Animation d'Objets Complexes en Informatique Graphique (Thèse d'Etat)
	90 -28	L. FRIBOURG	Generating Simplification Lemmas Using Extended Prolog Execution and Proof-Extraction
	90-29	L. COLSON	Représentation Intentionnelle d'Algorithmes dans les Systèmes Fonctionnels : une Etude de Cas. (Thèse)
	91 - 1	G. BERNOT	Testing Against Formal Specifiations : a Theoretical View
	91 - 2	A. BOUVEROT	Comparaison entre la Transformation et l'Extraction de Programmes Logiques.(Thèse)
	91 - 3	R. AMADIO	Bifinite Domains : Stable Case
	91 - 4	A. BOUVEROT	Extracting and Transforming Logic Programs
◇	91 - 5	P. HOOGVORST R. KERYELL P. MATHERAT N. PARIS	POMP or How to Design a Massively Parallel Machines with Small Developments
	91 - 6	G. BERNOT M. BIDOIT T. KNAPIK	Observational Approaches in Algebraic Specifications: a Comparative Study
	91 - 7	Y. LAFONT A. PROUTÉ	Church-Rosser Property and Homology of Monoids (revised version)
	91 - 8	G. BERNOT M. BIDOIT	Proving the Correctness of Algebraically Specified Software: Modularity and Observability Issues

	91 - 9	M. BIDOIT	Development of Modular Specifications by Stepwise Refinements Using the PLUSS Specification Language
	91 - 10	R. Di COSMO	Invertibility of Terms and Valid Isomorphisms. A Proof Theoretic Study on Second Order λ-calculus with Surjective Pairing and Terminal Object
	91 - 11	R. Di COSMO P.L. CURIEN	A Confluent Reduction for the λ-calculus with Surjective Pairing and Terminal Object and Terminal Object
	91 - 12	P. CRÉGUT	Machines à Environnement pour la Réduction Symbolique et l' Evaluation Partielle (Thèse)
	91 - 13	L. CHILLAN	Typing with Type Relations and ML-Polymorphism
	91 - 14	P.COUSOT R. COUSOT	Inductive Definitions, Semantics and Abstract Interpretation
	91 - 15	A. ASPERTI	A Linguistic Approach to Deadlock
	91 - 16	P.L. CURIEN T. HARDIN A. RÍOS	Normalisation Forte du Calcul des Substitutions
	91 - 17	N. PARIS	MOD2MAG User's Manual
	91 - 18	A. BUCCIARELLI T. EHRHARD	A Theory of Sequentiality
	91 - 19	P.L. CURIEN G. GHELLI	Subtyping + Extensionality : Confluence of $\beta\eta$top$_{\leq}$Reduction in F$_{\leq}$
	91 - 20	A. BUCCIARELLI T. EHRHARD	Sequentiality and Strong Stability
	91 -21	A.BUCCIARELLI T. EHRHARD	Extensional Embedding of a Strongly Stable Model of PCF
	91 - 22	T. EHRHARD P. MALACARIA	Stone Duality for Stable Functions
	91 -23	L. FRIBOURG	Mixing List Recursion and Arithmetic
	92 - 1	C. LAVATELLI	Cohérence dans les Catégories Fermées Sur un Résultat de G.E. Mints
	92 - 2	G.BERNOT M. BIDOIT T. KNAPIK	Towards an Adequate Notion of Observation
	92 - 3	G.BERNOT M. BIDOIT T. KNAPIK	Observational Specifications and the Indistinguishability Assumption
◇	92 - 4	G. CATAGNA G. GHELLI G. LONGO	A Calculus for Overloaded Functions with Subtyping
◇	92 - 5	N. PARIS	Définition de POMPC (version 1.99)