XIII

Les langages de programmation parallèle et leurs compilateurs *

1- Introduction

Depuis deux décennies, l'évolution des architectures des processeurs et des machines multiprocesseurs a une influence considérable sur les modes de programmation. Face à la profusion des modèles d'exécution, il est en effet difficile de dégager des modèles abstraits qui permettent de proposer des environnements de programmation standardisés et stabilisés, tirant le meilleur parti des performances du matériel.

La complexité des modes d'exécution, depuis les processeurs pipelinés, vectoriels ou superscalaires – à parallélisme interne –, jusqu'aux architectures multiprocesseurs – à mémoire partagée ou massivement parallèles –, est difficile à capturer par un modèle unique, qui devrait intégrer, entre autres, différents modes de contrôle, c'est-à-dire de synchronisation, et différents modes de gestion des espaces mémoire (caches, adressage distant, etc.).

Cette diversité – fondée sur des progrès du matériel – est apparue sur une période très courte, qui n'a pas permis des développements de même ampleur du logiciel. On peut donc observer un décalage extraordinaire entre le rythme de ces évolutions et l'échelle de temps en matière de développement et d'exploitation des codes applicatifs : quelques années – voire quelques mois – dans le premier cas et plusieurs décennies dans le second.

Dans le domaine du logiciel et des environnement de programmation, l'évolution des langages, les modèles de programmation qu'ils expriment, leur adéquation aux savoir-faire des utilisateurs, sont des facteurs aussi importants que l'efficacité

* Ce chapitre a été rédigé par Paul FEAUTRIER et Guy-René PERRIN.

de leur mise en œuvre, leur pérennité ou la portabilité des codes qu'ils tolèrent. Ce chapitre est un panorama des langages de programmation parallèle et des problèmes de compilation qui leur sont liés.

2- Evolution des modes de programmation

2.1- La programmation des machines vectorielles

Commençons ce tour d'horizon par la programmation des machines vectorielles, pour lesquelles nous parlerons de compilateur plutôt que de langage proprement dit. Il s'agit en effet pour le compilateur de *vectoriser* un programme Fortran quasi standard, c'est-à-dire de générer, à partir des boucles de ce programme, un code utilisant autant que possible les unités fonctionnelles du processeur.

Les extensions au langage consistent donc essentiellement en des directives de compilation et des fonctions prédéfinies pour *forcer* la vectorisation, au-delà de ce que peut produire une analyse de dépendances simple : le CFT (Cray Fortran Translator) est un exemple significatif (cf. [Per87]), où l'analyse de dépendances est limitée à un seul niveau de boucle et la vectorisation ne porte que sur un seul indice. En transformant le nid de boucles initial, le programmeur peut alors radicalement transformer l'effet de la compilation. C'est ainsi que ci-dessous, l'inversion de boucles est autorisée : on voit facilement que le programme de gauche est non vectorisable, alors que celui de droite l'est :

```
DO  1  I=1,N                          DO 1 J=2,N
DO  1  J=2,N                          DO 1 I=1,N
       A(I,J)=...A(I,J-1)...             A(I,J)=...A(I,J-1)...
1 CONTINUE                          1 CONTINUE
```

De plus, la vectorisation peut être forcée, sous la responsabilité du programmeur, qui doit s'assurer de la correction sémantique de la transformation, comme dans l'exemple suivant :

```
CDIR$ IVDEP
      DO 1 I=2,N,2
         A(I) =  ... A(I-1) ...
1     CONTINUE
```

Bien que l'on parle de *compilateur-vectoriseur,* ces deux exemples montrent que la programmation vectorielle suppose un minimum d'investissement de la part du programmeur, s'il veut utiliser efficacement les caractéristiques du matériel. Au-delà de ces deux exemples, qui ne sont pas spécifiques de telle ou telle caractéristique d'une architecture particulière, mais simplement révélateurs du concept général de vectorisation, si on veut atteindre une réelle efficacité (voir section 4.2), il suppose en outre de profiter autant que faire se peut, de caractéristiques propres à chaque matériel : par exemple, du chaînage entre les unités fonctionnelles vectorielles ou du parallélisme potentiel entre ces unités, ou de l'optimisation du temps d'amorçage des pipelines, éventuellement en adaptant la taille des vecteurs traités et en restructurant les boucles du programme (voir section 4.2.2). L'implication du programmeur peut donc être plus importante qu'il n'y paraît *a priori.*

Bien que l'exploitation des machines vectorielles soit en marge de l'étude du parallélisme massif, ce mode de programmation et sa marque sont inscrits, en positif et en négatif, dans la suite de l'histoire de la programmation parallèle. Ce mode de programmation a, en effet, pu accréditer l'idée d'un certain style de programmation, relégué à une simple *technique d'optimisation,* pour exploiter au mieux les caractéristiques de l'architecture. Un point très négatif est, lorsqu'elle sera transposée aux machines parallèles, l'idée qu'il n'est donc pas nécessaire, ni même souhaitable, de réellement imaginer de nouveaux modes de programmation, et par conséquent, de nouveaux langages de programmation.

Du point de vue des outils de programmation, un point très positif de cette génération de machines a été le développement des compilateurs vectoriseurs, fondés sur les mêmes techniques que la parallélisation automatique des programmes (cf. section 4 ci-dessous).

2.2- Les premiers modes de programmation parallèle

L'idée selon laquelle la programmation des machines parallèles ne doit pas requérir de savoir-faire nouveau de la part des utilisateurs, à qui on préfère parler de *portage de code,* a cours également avec les machines vectorielles multitâches et les premières générations de machines parallèles. Il est éclairant de relire, à ce sujet, l'introduction humoristique de [Bab88], qui montre comment deux extensions "apparemment innocentes" de Fortran sur la machine HEP de Denelcor (le préfixe `$` pour signifier les conditions de partage d'une variable par deux sous-programmes, et l'énoncé `CREATE` substitué à l'appel de procédure traditionnel), ont répandu le "génie du parallélisme" sur le monde ! Hélas, derrière ces notations se cacheront beaucoup de désillusions pour les utilisateurs... Ces désillusions ont pour noms : tâches, synchronisations, ressources critiques, etc., avec leur cortège de *deadlocks,* de non-déterminisme et d'erreurs dans les programmes. On trouve, toujours dans [Bab88], des *confessions* d'utilisateurs, devenues des classiques : partages involontaires de variables par utilisation du `COMMON`, problème de passage de paramètres par référence lors de la création de tâches, comme ci-dessous, etc. :

```
      DO 1 I=1, N
         CREATE CALCUL(I)
1     CONTINUE
```

Les difficultés intrinsèques de la programmation parallèle sont dès lors évidentes et vont, pour longtemps, freiner le passage des utilisateurs à ce type de machines. D'autant plus que ces utilisateurs sont très démunis face à cette difficulté

avec des environnements et interfaces très pauvres, et des extensions de langage séquentiel – Fortran en l'occurrence – par des primitives diverses d'activation de tâches, de définition de barrière de synchronisation, de section critique, ou d'échange de messages, selon le mode d'exécution sous-jacent (cf. les deux exemples ci-dessous). Ces premiers modes de programmation non portables ont rebuté les utilisateurs.

```
REAL SUM,A()
INTEGER I,N,NB_TASKS
COMMON A,SUM,N,NB_TASKS
EXTERNAL WORKER

SUM=0.0
DO I=1,NB_TASKS
  CALL TSKSTART(...,TASK)
ENDDO

DO I=1,NB_TASKS
  CALL TSKWAIT(...)
ENDDO
END

SUBROUTINE TASK

REAL SUML
INTEGER J,N,NB_TASKS
COMMON A,SUM,N,NB_TASKS

SUML=0.0
DO J=ME,N,NB_TASKS
  SUML=SUML+A(J)
ENDDO

CALL LOCKON(S)
SUM=SUM+SUML
CALL LOCKOFF(S)
RETURN
END
```

Programmation multitâche avec variables partagées (Exemple du Cray X-MP).

```
REAL SUM,TMP,A()
INTEGER I,J,N,NB_NODES,ME

NB_NODES=NUMNODES()
ME=MYNODE()

IF (ME.EQ.0) THEN
  CALL CSEND(N,ALLNDS,...)
ELSE
  CALL CRECV(N,...)
ENDIF

SUM=0.0
DO J=ME+1,N,NB_NODES
  SUM=SUM+A(J)
ENDDO

IF (ME.NE.0) THEN
  CALL CSEND(SUM,0,...)
ELSE
  DO I=1,NB_NODES-1
    CALL CRECV(TMP,...)
    SUM=SUM+TMP
  ENDDO
ENDIF
END
```

Programmation SPMD avec échange de messages (Exemple de l'Intel iPSC).

Il faut se souvenir des recherches nombreuses à l'époque sur les langages de programmation truffés de processus (à variables locales ou globales) et de primitives de synchronisation : CSP [Hoa78], Ada, etc. Le mode de programmation était au niveau du modèle d'exécution, en traitant des problèmes de concurrence, d'exclusion mutuelle, etc. Il était prévisible que ce mode ne pouvait être accepté en l'état par les utilisateurs, parce qu'il révèle, en effet, des difficultés intrinsèques au parallélisme : asynchronisme dû aux flots de contrôle multiples, non-déterminisme de l'exécution et problème de vivacité (interblocage et terminaison). Ces difficultés ont bien été mises à jour par les études sur les systèmes formels de preuve : les extensions de la logique de Hoare aux processus communicants en sont un bon exemple [AFR80].

2.2.1- La mémoire partagée

Ce bas niveau d'expression explicite de la concurrence existe encore dans les langages de programmation parallèle, pour les architectures reposant sur l'emploi de caches secondaires ou sur la notion de mémoire partagée virtuelle : que l'on considère un système de gestion de cohérence des caches locaux, pour les architectures de type serveurs de calcul à mémoire physiquement partagée, ou un système VSM *(Virtual Share Memory)*, pour les architectures à mémoire physiquement distribuée, avec cache ou système d'écriture distante, la gestion des accès concurrents aux variables, et des synchronisations nécessaires, requiert l'écriture de code supplémentaire. D'autre part, de tels systèmes ne dispensent pas d'un travail d'optimisation de la distribution des données et de structuration des programmes, par exemple pour localiser au mieux les données ou pour "aider" le compilateur à générer les opérations de préchargement des caches, afin d'obtenir des performances convenables.

2.2.2- La mémoire distribuée

Ce niveau d'expression par passage de messages demeure dans les systèmes actuels à mémoire physiquement et logiquement distribuée. Il explicite le placement des données et des calculs, dans un style de programmation SPMD, en intégrant des primitives de communications telles que des *send, receive ou multicast* – comme on l'a vu dans l'exemple ci-dessus. Ce mode de programmation suppose une identification de chaque processus et une programation par cas, les communications elles-mêmes pouvant être blocantes ou non. Un extrême dans ce style de programmation fut le langage OCCAM [Occ88] pour transputers, et son fichier de configuration établissant les liens entre les processus et canaux et les supports matériels.

2.3- La programmation des machines SIMD

L'apparition des architectures SIMD massivement parallèles a donné lieu à un nouveau mode de programmation parallèle – le parallélisme de données, *data parallelism* en anglais – et à la définition de nouvelles extensions des langages traditionnels. Ce mode de programmation, introduit dans le dialecte de Fortran pour la machine ILLIAC IV, puis dans les extensions de C pour MasPar (langage MPL) ou Connection Machine (langage C*), est fondé sur l'exécution séquentielle et synchronisée de la même instruction sur toutes les composantes de variables dites *parallèles*.

Plus précisément, les grands principes de ce mode de programmation sont les suivants :

- déclaration de variables indicées, dont les éléments sont répartis sur les processeurs de l'architecture (variables déclarées *plural* en MPL) ou définition de grilles de *processeurs virtuels,* associées à des variables indicées de la même forme (définition d'un VPS *(Virtual Processor Set)* en CM-Fortran, ou déclarations de *shapes* en C*) :

```
shape [256][256] matrice;
int : matrice A, B;
```

- expression de traitements globaux sur les variables parallèles, éventuellement avec un conditionnement définissant un sous-ensemble de processeurs actifs. Ces instructions signifient l'exécution de traitements identiques sur l'ensemble des processeurs virtuels actifs, et donc sur les composantes des variables impliquées,

- expression généralement explicite des communications entre processeurs virtuels, pouvant être associées à des expansions de variables *(diffusion)* ou des *réductions* (section 4.1.2). La définition des communications fait référence à l'indice courant du processeur virtuel dans la grille. Par exemple, en C* :

```
A = [.+1][.+1] B;
```

signifie que les éléments $A_{i,j}$ reçoivent simultanément (instruction SIMD `get`) la valeur de l'élément $B_{i+1,j+1}$ de la matrice *B*, dans un décalage uniforme de vecteur (1,1),

- définition de structures de contrôle itératives globales.

```
shape [N] vecteur;
main ()
    { vecteur int X,S;
      vecteur int I,N_PV;
      N_PV= pc_coord(0);
      S=X;
      I=1;
      whilesomewhere (I<N)
         { where (N_PV>=I) S+=[.-I]S;
           I=2*I;
         };
      }
```


3- Modèle de programmation et modèle d'exécution

3.1- L'époque de la confusion des modèles

Il est curieux qu'une étude sur les langages de programmation soit fondée sur les problèmes d'architecture et les modes d'exécution associés. C'est que pendant longtemps on a confondu le mode de programmation avec le modèle d'exécution : mode fondé sur la concurrence ou la communication pour les architectures MIMD, et mode fondé sur les variables parallèles pour les architectures SIMD. La conséquence est évidemment un manque de portabilité des programmes.

La confusion entre ces deux niveaux de modèles a permis un travail de compilation relativement simple, quasi trivial pour la programmation des architectures SIMD, mais également dans le cas MIMD, dans la mesure où le programme explicite les processus et leur mode d'interaction, voire les liaisons virtuelles avec les composantes de l'architecture. Restent, dans ce dernier cas, définissant ce qu'on appelle maintenant le *parallélisme de contrôle*, les problèmes de conception des programmes sous la forme d'un graphe de tâches (voir, par exemple, [Fos95]) et la difficulté de définition d'une implantation physique optimale, qui doit placer un graphe de tâches sur le graphe de l'architecture. Ces questions sortent du cadre de ce chapitre, mais mettent en cause le *grain* du parallélisme, c'est-à-dire la taille moyenne des tâches (en nombre d'instructions, par exemple), la *dépendance* entre les tâches et par conséquent, le *degré* de parallélisme qu'elles permettent et la *localité* des données traitées.

3.2- Enfin Malherbe vint ...

Il a fallu attendre pratiquement cette décennie pour voir apparaître une certaine "virtualisation", gage de *portabilité,* dans les modes de programmation :

- aussi bien pour une approche de type flots de contrôle multiples, suite logique des approches précédentes, sous la forme de bibliothèques de communications (avec PVM, par exemple) dans un parallélisme à gros grain,
- que pour une approche fondée sur le parallélisme des données, à grain fin, avec des langages de type HPF où apparaissent des alignements entre variables.

Sans rentrer dans le détail des approches de type PVM *(Parallel Virtual Machine)*, ou MPI *(Message Passing Interface)* [MPI93], qui font l'objet du chapitre XVI, on doit noter cependant que cette approche en termes de tâches (par programmation SPMD) et communications, intègre les fonctionnalités des outils précédents : langages à passage de messages, propres à chaque architecture à mémoire distribuée, ou bien bibliothèques de passage de messages (PARMACS, Express, etc.) commercialisées sur diverses architectures. Ces bibliothèques sont devenues dans les années 90 des standards *de facto*, ou agréés, par la volonté des utilisateurs de disposer de modes de programmation portables pour une efficacité acceptable.

Le jugement qu'on peut cependant porter sur cette approche est qu'elle renonce à proposer de nouveaux langages, mais adopte définitivement l'idée de définition de bibliothèques – portables et efficaces – dans le cadre d'un langage existant. On peut faire l'analogie avec l'usage de bibliothèques scientifiques (cf. chapitre XVII) de type ScaLAPACK ou PETSc, qui assurent elles-aussi, dans un autre registre – celui des calculs –, la portabilité et l'efficacité des programmes, en renonçant à une remise en cause de nature plus algorithmique.

3.3- Le parallélisme de données

A partir des notions introduites dans les langages pour machines SIMD massivement parallèles, un véritable paradigme nouveau de programmation s'est dégagé : celui dit du parallélisme de données, qui s'est enrichi de la notion *d'alignement* entre variables.

On trouve quelques idées de base sur ce type de langages dans le CM-Fortran de TMC [TMC93], où une certaine forme de distribution et d'alignement des tableaux peut être définie par la directive de compilation `LAYOUT` – les dimensions déclarées `SERIAL` sont placées sur le même processeur virtuel, tandis que celles déclarées `NEWS` sont distribuées sur un ensemble de processeurs virtuels – et la directive `ALIGN` :

```
      DIMENSION A(N,N)
      DIMENSION X(N)
CMF$  LAYOUT A(:NEWS,:SERIAL)
CMF$  ALIGN X(I) WITH A(I,1)
```

Sur cet exemple, les éléments d'une ligne i de la matrice A sont alloués au même processeur que la i-ème composante du vecteur X.

On trouve par ailleurs dans ce langage les opérations caractéristiques de la programmation des machines SIMD, entre autres les opérations globales sur les tableaux, une forme d'itération parallèle `FORALL` et de conditionnement `WHERE`.

Ces notions, avec les notations de "sections" de tableaux, d'affectation globale et de réduction *(Array Intrinsic Functions)*, se généralisent dans une forme de standard, le Fortran 90 [ANS90]. Pour la première fois, les extensions *data parallel* sortent du contexte des machines SIMD et sont intégrées dans la version du langage désormais disponible sur des architectures quelconques.

```
      REAL X(N,N),Y(N,N)
      Y(2:N-1,2:N-1)=(X(1:N-2,2:N-1)+X(3:N,2:N-1)+
     $                X(2:N-1,1:N-2)+X(2:N-1,3:N))/4
      DIFF=MAXVAL(ABS(Y-X))
```

Utilisation de sections de tableaux et de réduction en Fortran 90.

L'aboutissement actuel est le HPF, High Performance Fortran [HPF93], langage complexe, qui définit des extensions de Fortran 90, notamment en matière de distribution des données, et reprend certaines constructions de nature typiquement SIMD, comme le `FORALL` (voir l'introduction par R. Schreiber [Sch96] et la référence [KLS94]).

Le placement des données est défini en trois étapes : les deux premières peuvent être spécifiées par le programmeur par l'emploi des directives de compilation `ALIGN` et `DISTRIBUTE`. Le niveau de placement obtenu est alors virtuel et donc portable. La troisième étape est à la charge du compilateur, en fonction de ces directives et de l'architecture physique.

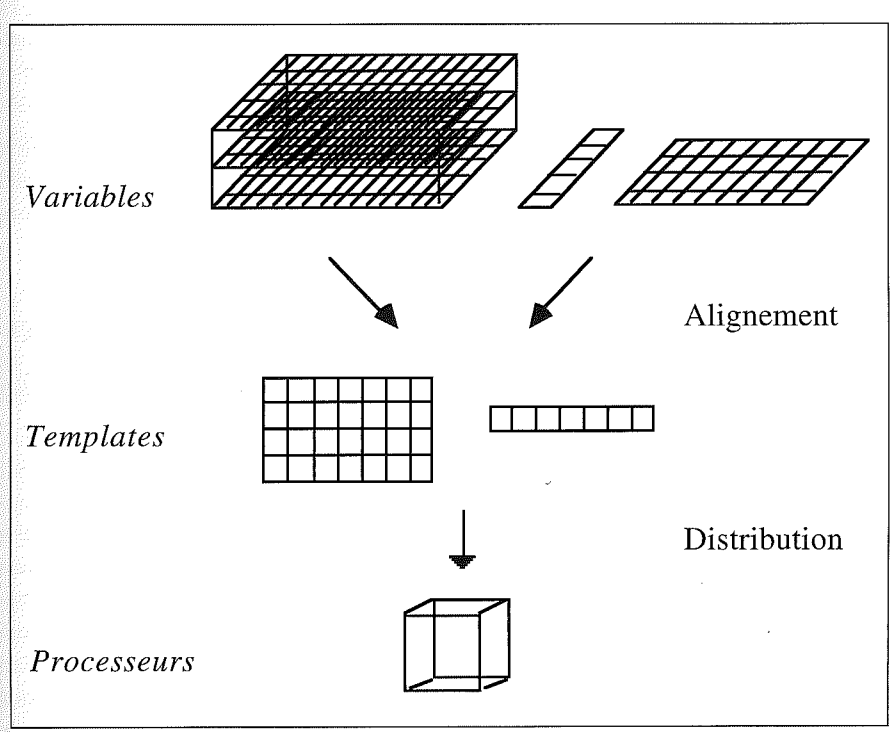
Les directives *d'alignement* de la forme

```
HPF$  ALIGN A(...) WITH T(...)
```

définissent une allocation logique des variables indicées sur des tableaux virtuels, appelés *templates,* ou de variables indicées entre elles. Les directives de distribution, de la forme

```
HPF$  DISTRIBUTE T(...) ONTO P
```

précisent le placement des *templates* sur une grille virtuelle de processeurs. Les arguments `BLOCK` ou `CYCLIC` de la distribution, précisent une allocation des composantes par bloc, ou modulaire, pour chaque dimension des templates.



A noter que ces directives de compilation sont des commentaires au compilateur (on dit parfois des "conseils" au compilateur) et concernent donc l'efficacité des programmes – en localisant les données autant que possible – et non pas leur correction. Il en est de même des directives *runtime* de la forme `REDISTRIBUTE` ou `INDEPENDENT` – cette dernière suppose que l'utilisateur a vérifié l'indépendance des itérations impliquées pour permettre leur exécution éventuelle en parallèle.

Ainsi devenu un véritable paradigme de programmation, le parallélisme de données pose de nombreux problèmes et ouvre des voies de recherche intéressantes [PeD96], que nous évoquerons plus loin :

- compilation efficace pour une architecture MIMD,
- traitement efficace de structures de données dynamiques, irrégulières ou creuses,
- traduction source-à-source Fortran vers HPF : génération des directives et analogie avec les techniques de parallélisation,
- mixité des modes de programmation, par exemple par un couplage HPF-MPI

Il est indéniable que ce mode de programmation présente de grands intérêts :

- étant un mode de programmation essentiellement séquentiel, il offre une transition douce de la programmation séquentielle vers le parallélisme (cf., par exemple, les problèmes de preuve des programmes dans [PeD96]),
- par une approche très macroscopique de la conception des programmes, en reléguant le parallélisme au cœur des instructions, il permet une grande maîtrise de la complexité de la programmation et est insensible, en particulier, à la taille des données : il offre ainsi une sorte *d'extensibilité logicielle,*
- il donne lieu à une véritable étude académique en tant que langage et modèle de programmation, et à la définition de normes garantissant la portabilité des programmes : forum HPF pour Fortran et le projet DPCE, en cours, pour une extension de C.

3.4- Compilation des langages à parallélisme de données

Sans vouloir entrer dans les détails, nous donnons ici une brève description des techniques utilisées par les compilateurs des langages à parallélisme de données.

3.4.1- La résolution à l'exécution

Un programme à parallélisme de données est caractérisé par le fait que le programmeur a spécifié dans le programme la répartition des données entre les divers processeurs. Il est donc possible d'écrire des fonctions calculant le numéro du processeur hébergeant une donnée, et partant, de savoir si un processeur dispose des données nécessaires à un calcul, et à qui il doit les demander, le cas échéant. Pour écrire le programme distribué, il suffit de décider du placement de chaque opération de calcul. On s'appuie en général sur la règle des écritures locales – *owner computes rule* – selon laquelle un calcul est effectué par le processeur qui en héberge le résultat. Le programme compilé prend alors la forme suivante [Ger89] : chaque processeur exécute une réplique du programme original, mais les opérations de calcul sont modifiées comme suit :

- un premier test permet de savoir si le processeur héberge le résultat du calcul courant. S'il n'en est pas ainsi, la suite de l'instruction n'est pas effectuée,
- une série de tests permet de repérer les données non locales, qui sont demandées aux processeurs qui les hébergent,
- quand toutes les données non locales ont été récupérées, le calcul est effectué.

L'intérêt de cette méthode est qu'elle ne fait aucune hypothèse sur la structure du programme original. Par contre, elle est extrêmement inefficace.

3.4.2- Modélisation par polyèdres

La méthode ci-dessus peut être optimisée si le programme original ne comporte que des boucles DO et des tableaux à indices affines. Dans ce cas, les tests peuvent être effectués à la compilation [Fur95, Coe96] et intégrés dans le code de parcours des structures de données. Soit, par exemple, la boucle :

```
DO I = 1,N
    A(I) = ....
ENDDO
```

Supposons que le tableau A soit distribué de façon cyclique sur deux processeurs : l'élément *A(1)* est sur le processeur *1*, l'élément *A(2)* est sur le processeur *2*, l'élément *A(3)* est sur le processeur *1*, etc. Alors le code de chaque processeur s'écrit simplement :

```
DO I = P,N,2
    A(I) = ....
ENDDO
```

où *P* est le numéro du processeur courant. Il n'y a plus de test pour décider si un calcul doit être effectué ou non.

Une autre transformation importante est la vectorisation des messages. Elle consiste à sortir le code de communication de la boucle, ce qui permet de transmettre des messages plus longs et de gagner sur le temps de latence. Elle n'est valide que si la boucle est parallèle : bien que ce ne soit pas indispensable en théorie, les langages à parallélisme de données comportent des directives permettant de signaler de telles boucles parallèles. Ici encore, la construction de longs messages n'est possible que pour les programmes utilisant des boucles et des tableaux à indices affines. Les éléments à transmettre sont alors définis par des systèmes d'inégalités linéaires. On en déduit le code de rassemblement des données à transmettre par des techniques de balayage de polyèdres.

3.4.3- Exemple : état de l'art pour la compilation de HPF

Il existe actuellement quelques compilateurs pour HPF, à la fois chez les industriels et dans le monde académique (cf. [Paz96] pour un survol des environnements actuels). On trouve également des compilateurs pour des langages moins généraux que HPF, comme CRAFT de Cray et CM-Fortran de TMC. Ces compilateurs, en général plus efficaces que les compilateurs HPF parce qu'ils s'attaquent à des langages moins ambitieux, sont en train d'être mis à la norme HPF.

Chacun de ces compilateurs peut prétendre légitimement à l'efficacité. Le malheur est que chacun d'entre eux est efficace dans un domaine différent. Il s'ensuit que bien que tous les programmes HPF se compilent partout, il faut adapter son style de programmation à son compilateur sous peine de n'obtenir que des performances presque séquentielles. Il s'agit là d'une maladie de jeunesse qui disparaîtra avec le temps, à condition que le langage n'évolue pas trop vite. De plus, certains traits de HPF sont et seront toujours difficiles à compiler : il en est ainsi, par exemple, des redistributions inconnues, qui surviennent quand une procédure n'a pas accès aux distributions de ses arguments. Sur ces points particuliers, il faudra sans doute adapter le langage [Coe96].

4- Parallélisation et vectorisation automatiques

Bien que la vectorisation automatique soit antérieure, nous avons choisi de présenter d'abord la parallélisation automatique. En effet, les concepts de base sont les mêmes. On peut voir la vectorisation automatique comme une parallélisation automatique suivie d'une étape de reconnaissance de formes permettant de repérer les *instructions vectorielles.*

4.1- Parallélisation

Qu'est-ce qui permet d'espérer qu'un compilateur puisse jamais engendrer un programme parallèle efficace? Après tout, il n'existe pas encore de méthode de programmation automatique séquentielle réellement opérationnelle. Le problème n'est-il pas encore plus difficile si l'on recherche du parallélisme?

La réponse est qu'il s'agit d'objectifs bien différents. Une des approches de la programmation automatique est de partir d'un théorème affirmant qu'un objet ayant certaines propriétés existe. Si la logique utilisée est constructive, de la preuve du théorème on peut extraire un algorithme qui construit l'objet. Dans le cas de la parallélisation automatique, ce travail est déjà fait : la donnée est un programme séquentiel censé répondre aux spécifications du problème à résoudre. S'il y a à prouver la correction du programme parallèle, c'est en prenant pour hypothèse que le programme séquentiel est correct. Il s'agit donc d'une approche de type "transformation de programmes", dont de nombreux auteurs, à commencer par Jacques Arsac, ont montré qu'elle est bien plus facile qu'une création *ex nihilo.*

L'objectif de la parallélisation automatique est donc de faire subir au programme séquentiel une série de transformations qui n'en change pas les résultats, tout en l'adaptant de mieux en mieux à l'architecture parallèle visée. Naturellement, la validité de chaque pas de transformation doit être prouvée. L'intérêt des méthodes que nous allons présenter est que cette preuve peut être "mise en facteur". Pour chaque transformation, on peut prouver un théorème (nous ne le ferons pas ici) affirmant, sous certaines conditions, que le programme original et le programme transformé sont équivalents. Le travail du paralléliseur est alors de vérifier les conditions de validité de la transformation, c'est-à-dire les hypothèses du théorème, puis de l'appliquer mécaniquement.

4.1.1- Le concept de dépendance

La transformation la plus importante est celle qui permet de remplacer un groupe d'instructions exécutées séquentiellement par le même groupe exécuté en parallèle. On peut se demander ce qui empêche cette transformation, ou, en d'autres termes, qu'est-ce qui fait obstacle au parallélisme. La réponse est que les opérations d'un programme *dépendent* les unes des autres. Le cas le plus intuitif est celui d'une opération qui calcule un résultat intermédiaire, suivie d'une autre qui utilise ce résultat. Il paraît évident que la deuxième opération ne peut commencer que quand la première est terminée; il ne peut donc pas y avoir de parallélisme entre ces deux opérations. On dit que les deux opérations sont en *dépendance de flot.* Ce type de dépendance est liée à la structure de l'algorithme et ne peut pas, en général, être éliminée.

Dans le même ordre d'idées, deux opérations qui écrivent dans la même cellule de mémoire ne peuvent pas être parallélisées, parce que le contenu de cette cellule serait indéfini. On parle alors de dépendance de sortie. Enfin, il y a anti-dépendance quand la première opération lit une cellule et la deuxième la modifie. Les dépendances de sortie et les anti-dépendances ne sont pas liées à l'algorithme, mais à la gestion de la mémoire par le programme source. On peut les éliminer en évitant de réutiliser une même cellule pour héberger des informations différentes. Nous verrons plus loin comment cette transformation peut être réalisée.

Les dépendances ci-dessus sont associées aux données du programme. D'autres dépendances sont associées au contrôle. Le cas le plus simple est celui d'un test qui gouverne deux branches. On ne peut connaître la branche à exécuter qu'une fois le test effectué. Là également, il ne peut y avoir parallélisme. Une situation analogue se présente pour le cas des boucles `while`. Il est possible de contourner ces *dépendances de contrôle* par *spéculation,* mais il s'agit là de recherches récentes qui n'ont pas encore atteint les paralléliseurs du commerce.

La détermination des dépendances est au centre de toute parallélisation. Lorsqu'il s'agit de dépendances entre scalaires, le calcul est très simple et se ramène à des comparaisons de noms. On notera cependant que les langages de programmation usuels permettent de créer des alias, c'est-à-dire de désigner un même objet par des noms différents. En Fortran, cette création d'alias est spécialement facile : il existe une déclaration – `EQUIVALENCE` – dont c'est le seul but. On peut également

fabriquer des alias à l'aide de COMMON et des appels de sous-programmes. Dans d'autres langages comme C, on peut fabriquer des alias dynamiquement à l'aide de pointeurs. La présence d'alias gène considérablement les paralléliseurs, qui en général renoncent à transformer un programme s'ils ont détecté la présence possible d'alias.

La détection des dépendances sur tableaux est beaucoup plus complexe que celle des dépendances scalaires. Il est, par exemple, suicidaire de traiter un tableau comme un tout et de considérer qu'il est modifié dans son ensemble dès qu'une de ses entrées est modifiée : cette convention aurait pour effet de faire disparaître l'essentiel du parallélisme d'un programme. Il faut plutôt replacer les instructions qui agissent sur les tableaux dans leur contexte, celui d'une ou plusieurs boucles emboîtées. Les cellules lues et modifiées varient au cours de l'exécution de la boucle : on doit les considérer comme des fonctions des compte-tours des boucles englobantes. Il y a dépendance si on peut trouver deux itérations différentes qui accèdent à la même cellule d'un tableau, donc qui engendrent des indices égaux. Ceci se traduit en un système d'équations et d'inéquations pour lequel on doit discuter l'existence de solutions entières : les inconnues sont les compte-tours des boucles, les équations expriment l'égalité des indices de tableaux, alors que les inégalités prennent en compte les bornes des boucles et le fait que les itérations doivent être distinctes. Si on peut mettre une solution en évidence, il y a dépendance. De plus, on peut associer chaque dépendance à l'une des boucles englobantes. L'absence de dépendance indique que la boucle correspondante peut être parallélisée.

La discussion des systèmes de dépendances ne peut être automatique que si tout y est linéaire : les indices et les bornes des boucles. Fort heureusement, cette situation se présente très fréquemment dans les programmes de calcul numérique et de traitement du signal. Il existe cependant des exceptions importantes, comme par exemple les programmes appliquant la méthode des éléments finis et, plus généralement, tout ce qui concerne les calculs sur matrices creuses.

4.1.2- Transformations de programmes

Très fréquemment, l'analyse des dépendances montre que le programme objet ne comporte que peu ou pas de parallélisme. Ce n'est pas que les algorithmes usuels en soient dépourvus, mais bien plutôt que lors de leur implémentation, le programmeur a appliqué des optimisations qui détruisent le parallélisme potentiel. Pour réussir une parallélisation, il faut défaire ces optimisations en appliquant des *transformations parallélisantes*, qui ne sont autre que des *pessimisations*.

L'exemple type de ces optimisations qui détruisent le parallélisme, et celui que l'on rencontre le plus fréquemment, est la réutilisation de la mémoire à des fins de réduction de l'encombrement du programme. Il est facile de se convaincre que la taille de la mémoire de travail d'un programme et son degré de parallélisme sont étroitement liés. Si en effet n opérations peuvent être exécutées en parallèle, elles doivent écrire leurs résultats à des adresses différentes, sans quoi elles seraient en dépendance de sortie. La taille de la mémoire de travail est donc au moins de n. A l'extrême limite, un programme qui ne modifie qu'une cellule de mémoire est nécessairement séquentiel. Si donc un programme n'a pas assez de parallélisme, il faut tenter d'en créer en augmentant la taille de la mémoire. Ceci peut se faire en renommant des scalaires, en transformant des scalaires en tableaux ou en augmentant le nombre de dimensions – le *rang* – d'un tableau. Les deux premières transformations sont assez faciles à réaliser, et on les trouve dans presque tous les paralléliseurs. L'expansion de tableau est plus délicate; elle n'est pas encore sortie des laboratoires.

D'autres transformations permettent de réordonner les opérations d'un programme de façon à regrouper celles qui ont les mêmes caractéristiques de parallélisme. La transformation *"éclatement de boucle"*, par exemple, permet dans certains cas de séparer, dans une boucle, les opérations qui doivent être exécutées en séquence de celles qui peuvent être exécutées en parallèle. La transformation *"inversion de boucles"* a pour but d'adapter le parallélisme à l'architecture cible. Placer les boucles parallèles à l'extérieur, par exemple, permet d'augmenter le grain de parallélisme, ce qui convient aux multiprocesseurs; les placer à l'intérieur, au contraire, convient mieux aux machines vectorielles (voir section 2.1). On peut démontrer que cette *intériorisation* des boucles parallèles est toujours possible, alors que leur *extériorisation* ne l'est pas toujours : c'est peut-être la raison pour laquelle les architectures vectorielles semblent plus faciles à programmer que les multiprocesseurs.

Un dernier type de transformation, enfin, s'appuie sur les propriétés mathématiques des opérations effectuées. Considérons, par exemple, un calcul de somme :

```
x = a + b + c + d
```

Presque tous les langages de programmation interprètent ce calcul comme :

```
x = ((a + b) + c) + d
```

ce qui ne comporte aucun parallélisme. Si cependant on se souvient que l'addition est associative, on peut écrire :

```
y = a + b
z = c + d
x = y + z
```

ce qui comporte bien du parallélisme. Il est clair que cet exemple n'est pas réaliste; la méthode ne prend tout son intérêt que si le nombre de valeurs à cumuler est grand, comme dans le cas des calculs d'algèbre linéaire (produit scalaire, produit de matrices) ou de statistique (calculs de moyennes). Il existe beaucoup d'opérateurs associatifs à coté de l'addition. On parle alors de *réduction* par l'opérateur associatif considéré.

Comme il n'y a pas d'algorithme permettant de décider de l'associativité d'un opérateur (ici encore, il faudrait prouver un théorème), on utilise en général une technique de reconnaissance de forme, parfois précédée par une phase de normalisation du programme source. Si l'analyse des réductions sur scalaires est actuellement bien maîtrisée, il n'en est pas de même de celle des réductions sur tableaux, qui font l'objet de recherches actives.

L'utilisation de ce type de transformation pose cependant un problème délicat : l'associativité de certains opérateurs n'est parfois assurée qu'aux erreurs d'arrondi près. C'est le cas, par exemple, des additions en virgule flottante. On peut en fait distinguer trois situations :

- l'algorithme utilisé est numériquement stable. Dans ce cas, l'ordre des calculs dans une chaîne associative est de peu d'importance, et la transformation peut être appliquée sans crainte,
- l'algorithme utilisé est naturellement instable, mais l'ordre des calculs a été soigneusement choisi pour le stabiliser. Il faut alors se garder d'y toucher,
- le dernier cas est celui où la stabilité de l'algorithme n'a pas été étudiée. On peut alors se demander quel est l'intérêt des résultats obtenus, aussi bien en séquentiel qu'en parallèle. C'est dans ce type de situation que l'on voit apparaître des artefacts – non-conservation de l'énergie, brisure de symétrie, influence du degré de parallélisme sur les résultats – qui doivent conduire à une révision de l'algorithme original.

4.1.3- Structure d'un paralléliseur

Nous pouvons maintenant esquisser la structure d'un paralléliseur. La première étape est l'analyse syntaxique du programme source. Il s'agit là d'un traitement standard, qui diffère peu de ce que font les compilateurs séquentiels. Toutefois, les paralléliseurs utilisent en général une représentation interne du résultat de cette analyse qui est de plus haut niveau que celle des compilateurs habituels : on conserve plus d'informations sur la structure du programme, alors que les compilateurs séquentiels perdent la notion de boucle et représentent les expressions par du code "trois adresses" très proche de l'assembleur.

Vient ensuite l'étape de l'*analyse sémantique*, qui, pour l'essentiel, porte sur le calcul des dépendances. Cette étape peut cependant être précédée d'une *analyse préliminaire* : recherche d'invariants, élimination des variables inductives, structuration. L'analyse de dépendance peut être plus ou moins précise, suivant la qualité des tests utilisés, et plus ou moins complète. Beaucoup de paralléliseurs, par exemple, se contentent de rechercher les dépendances nid de boucles par nid de boucles ; d'autres étudient le programme complet. Enfin, de très rares paralléliseurs effectuent une *analyse interprocédurale*, ce qui veut dire qu'ils recherchent les dépendances entre opérations dépendant d'instances de procédures différentes. Le résultat de cette analyse se représente le plus souvent sous la forme d'un *graphe de dépendance*.

L'étape suivante est la plus délicate. Il s'agit de décider des transformations qu'il est utile d'appliquer au programme avant de le paralléliser. Il n'y a pas pour cela de règles précises, et on doit se contenter d'heuristiques. Certaines de ces heuristiques sont de bon sens. Par exemple, le paralléliseur doit passer l'essentiel de son temps à la parallélisation des parties du programme qui consomment le plus de temps de calcul. On détecte le plus souvent ces *noyaux* par analyse d'une exécution. Cependant, certains compilateurs disposent d'un module de prédiction des temps de calcul, en général approximatif.

On peut ensuite se laisser guider par la structure du graphe de dépendance. Par exemple, la présence de dépendances de sortie à l'intérieur d'une boucle suggère l'application d'une transformation d'expansion de scalaire. De même, une dépendance de sortie entre nids de boucles distincts suggère un *renommage*. Cependant, si deux instructions sont liées à la fois par une dépendance de sortie et une dépendance de flot, il est inutile de procéder à une expansion ou un renommage, parce que ces transformations ne feront pas disparaître la dépendance de flot. Il est, en général, peu fructueux de s'intéresser aux anti-dépendances, car celles-ci disparaissent avec les dépendances de sortie.

Certaines transformations, enfin, sont suggérées par l'architecture cible. Si celle-ci est vectorielle, on doit abaisser la complexité des instructions élémentaires pour les adapter à ce que peuvent faire les pipelines usuels. Ceci est obtenu par éclatement de boucle, éclatement d'instructions et inversion de boucles. Par contre, sur un multiprocesseur, les tâches doivent être aussi complexes que possible pour mieux amortir le temps perdu à les lancer. Il faut utiliser des transformations de type *fusion*.

Pour terminer, il faut générer le code parallèle. Cette étape dépend beaucoup de ce dont on dispose sur l'architecture parallèle. Sur certains multiprocesseurs à mémoire globale, on dispose d'un compilateur parallèle. Il suffit d'indiquer à celui-ci les boucles parallèles, soit au moyen d'annotations, soit en remplaçant les boucles `DO` par des boucles `FORALL`. Dans d'autres cas, on peut se ramener à la notation vectorielle de Fortran 90. Par contre, pour un multiprocesseur à mémoire distribuée, la génération du code objet ressemble beaucoup à la compilation d'un langage à parallélisme de données (voir section 3.4).

4.1.4- Modèle polyédrique

La sélection des bonnes transformations est l'étape la plus difficile de la parallélisation automatique. Le modèle polyédrique a été inventé pour simplifier cette sélection, l'objectif ultime étant de déterminer directement le résultat des transformations sans passer par les étapes intermédiaires. Pour cela, il faut regrouper plusieurs transformations simples en une seule transformation plus puissante. On y parvient en observant que presque toutes les transformations employées en parallélisation automatique sont des *changements de base* dans l'espace des itérations. Comme il est bien connu, de tels changement de base peuvent être représentés par des matrices, et l'enchaînement de deux transformations est représenté par le produit des matrices associées. On peut donc rechercher directement la matrice de transformation qui met en évidence "le plus de parallélisme", en un sens à préciser.

Une des techniques de base consiste à rechercher un *ordonnancement* du calcul, c'est-à-dire à trouver la date d'exécution de chaque opération du programme. Tous les ordonnancements ne sont pas légitimes : on doit s'assurer que deux opérations en dépendance sont ordonnancées en succession. A partir d'un ordonnancement légitime, on peut construire un programme parallèle : deux opérations ordonnancées au même instant sont exécutées en parallèle.

On peut de la même façon rechercher un *placement,* c'est-à-dire trouver le nom du processeur qui exécute chaque opération du programme. On doit affecter deux opérations au même processeur si elles échangent de l'information. Si on applique cette règle à toutes les opérations d'un programme, on retrouve, en général, toutes les opérations sur le même processeur, parce que, sauf dans des cas particuliers, toutes les opérations d'un programme sont liées par des chemins d'échange d'information. Pour obtenir un programme parallèle, on doit accepter des *communications résiduelles.* On choisit, en général, de ne conserver que des communications correspondant à un flot de données le plus réduit possible.

Lorsque l'on a déterminé un ordonnancement et un placement, on peut les combiner en une seule transformation d'espace/temps, qui donne à la fois l'heure et le lieu d'exécution de chaque opération du programme. Le programme parallèle s'en déduit par "inversion" : on détermine l'opération que chaque processeur doit exécuter à chaque instant du calcul. Pour que cette détermination soit possible, il faut que la transformation d'espace/temps soit inversible. Il est même préférable que cette transformation soit bijective, ce qui permet d'assurer que chaque processeur a du travail à presque tous les instants du calcul. Il s'agit là de contraintes complexes que l'on doit imposer à la transformation d'espace/temps [Fea96].

A l'heure actuelle, les paralléliseurs basés sur le modèle polyédrique sont encore des outils de laboratoire, et de nombreux progrès – spécialement sur la vitesse de compilation – doivent être faits avant qu'ils puissent être mis entre toutes les mains.

4.1.5- Les limites de la parallélisation automatique

Le principal défaut des paralléliseurs actuels est l'étroitesse de leur domaine d'application. Un programme ne peut être parallélisé que si ses dépendances peuvent être calculées, et ceci n'est en principe possible que si le programme est à contrôle statique : boucles `DO` à bornes affines, tableaux à indices affines. Naturellement, on peut légèrement dépasser les limites de ce modèle. Une technique efficace est d'ignorer les constructions – bornes ou indices – non affines. Cette démarche ne peut que faire perdre du parallélisme; dans certains cas, les éléments restants peuvent suffire à prouver qu'une dépendance n'existe pas. Cependant, cette méthode est en général insuffisante pour paralléliser les programmes contenant des calculs d'indices complexes, comme par exemple les calculs sur matrices creuses et les codes d'éléments finis.

De même, la parallélisation de codes contenant des procédures et des appels de procédures est difficile pour les logiciels actuels. On peut d'ailleurs se demander en quel sens de tels codes doivent être parallélisés. On peut, par exemple, envisager de paralléliser chaque procédure isolément. On se heurte alors au fait que la procédure peut avoir des dépendances cachées, dues, par exemple, à deux paramètres formels représentant le même paramètre effectif. Ce phénomène *d'aliasing* rend difficile une parallélisation indépendante du contexte.

Une autre technique est de considérer les appels de procédures comme des instructions ordinaires, et de tenter de paralléliser ces appels. Il faut alors calculer les ensembles de variables lues et modifiées par la procédure. Ceci nécessite d'avoir

Enfin, les programmes qui manipulent des structures de données dynamiques sont complètement hors de portée des paralléliseurs actuels. Ceci est vrai quel que soit le langage utilisé : manipulations d'indices en Fortran, pointeurs et gestion explicite de la mémoire en C, pointeurs cachés et utilisation d'un ramasse-miettes en Java ou en Caml.

En d'autres termes, seuls les programmes les plus simples et les plus réguliers peuvent être parallélisés à l'heure actuelle. De nombreuses années de recherche sont encore indispensables pour briser cette barrière.

4.2- Vectorisation

La vectorisation d'un programme commence comme sa parallélisation, par une analyse syntaxique suivie d'une recherche des dépendances. Les transformations que l'on applique ensuite sont différentes : leur but est de mettre le programme sous une forme telle que les opérations vectorielles y sont facilement reconnaissables. On termine en appliquant diverses optimisations qui permettent de mieux adapter le programme à la structure de la machine cible.

4.2.1- Forme normale vectorielle

Un programme est en forme normale vectorielle lorsque ses instructions de traitement sont des opérations "à trois adresses" portant sur des vecteurs. On peut, par exemple, les noter sous la forme Fortran 90 :

```
A(IA:JA) = B(IB:JB) + C(IC:JC)
```

où les "étendues" *JA-IA, JB-IB ET JC-IC* sont égales et où le signe + peut être remplacé par n'importe quel autre opérateur binaire. De telles instructions peuvent être implémentées par une ou plusieurs opérations d'un pipeline vectoriel, même si elles utilisent des opérateurs complexes ou si elles sont englobées dans plusieurs niveaux de boucles.

Pour mettre un programme sous forme vectorielle, plusieurs transformations doivent être effectuées. On utilise tout d'abord l'éclatement d'instructions, qui permet d'atteindre la forme à trois adresses. On essaie ensuite d'éclater un maximum de boucles, ce qui n'est pas toujours possible. On sait, en particulier, que si une boucle contenant plus d'une instruction ne peut être éclatée, c'est que cette boucle doit rester séquentielle. L'analyse des dépendances permet ensuite de savoir quelles sont les boucles parallèles. Pour terminer, on intériorise les boucles parallèles ; à ce moment, les boucles vectorielles peuvent être détectées sans difficulté.

L'ensemble de ces transformations a été rassemblé en un élégant algorithme par Allen et Kennedy [Al,K87]. Historiquement, cet algorithme est le premier exemple d'une technique pouvant enchaîner plusieurs transformations sans tâtonnement.

4.2.2- Optimisations particulières

4.2.2.1- Strip mining et utilisation des registres vectoriels

Les calculateurs en pipeline sont équipés de registres pouvant contenir tout ou partie d'un vecteur. La longueur d'un registre vectoriel est fixe ; une valeur courante est de 64 éléments. Pour obtenir les meilleures performances, il faut charger les registres vectoriels avec des "tranches" des vecteurs de l'application, effectuer un maximum d'opérations de registre à registre, ranger les résultats en mémoire et passer à la tranche suivante. Ce découpage des vecteurs en tranches peut être vu comme une transformation de boucle. Appelée *strip mining*, elle est toujours possible.

De plus, les machines vectorielles supportent une forme spéciale d'expansion de scalaire. Si, dans une boucle à laquelle le *strip mining* a déjà été appliqué, un scalaire est utilisé comme variable temporaire, il suffit de l'affecter à un registre vectoriel pour voir disparaître les dépendances dont il est la cause. Il s'agit là d'une optimisation de très bas niveau, qui se fait au moment de la génération du code machine.

4.2.2.2- Reconnaissance des réductions

Les ordinateurs en pipeline peuvent exécuter efficacement les réductions à condition que l'opérateur utilisé soit associatif *et* commutatif. La méthode utilisée est un simple rebouclage d'un pipeline sur lui-même. Les vectoriseurs du commerce reconnaissent un certain nombre de réductions sur scalaires à condition que leur écriture soit standardisée, et que leurs opérateurs fassent partie d'une courte liste : addition, multiplication, calcul de maximum ou de minimum, opérations booléennes.

4.2.2.3- Opérations gardées

Il peut arriver que l'on rencontre des boucles parallèles qui contiennent des tests. En voici un exemple simple :

```
DO I=1,N
   IF (A(I).GT.0.) B(I)=SQRT(A(I))
ENDDO
```

Cette boucle peut être exécutée efficacement sur une machine vectorielle possédant des opérations masquées ou des instructions de *scatter-gather*. Dans tous les cas on construit un masque, c'est-à-dire un vecteur de bits représentant le résultat du test `(A(I).GT.0.)`. Ensuite, on peut, par exemple, extraire les éléments de A qui correspondent au masque, calculer leurs racines carrées, et ranger les résultats à leur place dans B. Ici le parallélisme découle du fait que le vecteur A sur lequel porte le test n'est pas modifié dans la boucle. Ceci peut être vu sur le graphe de dépendance à condition que les prédicats des tests y soient pris en compte comme opérations qui lisent des cellules de mémoire, mais n'en modifient pas.

5- Discussion générale et conclusion

5.1- Qu'offre-t-on à un utilisateur ?

Dans le cas le plus général des machines MIMD à mémoire distribuée, la distribution des calculs, des données et les communications doivent être précisées, par l'utilisateur, ou par le compilateur, à la compilation ou à l'exécution. On peut classer les modes de programmation parallèle en comparant les quantités d'information que doivent respectivement fournir le programmeur et le compilateur. Mais précisons tout d'abord qu'aucune méthode de programmation ne dispense le programmeur de spécifier l'algorithme de son programme. Les recherches sur la déduction de programmes à partir d'une spécification n'en sont, en effet, qu'à leurs premiers balbutiements.

- Au plus bas niveau, lorsqu'il utilise un langage à parallélisme explicite (voir section 3.1), le programmeur doit TOUT spécifier : l'implantation des données, le parallélisme, les communications et la distribution des tâches sur les processeurs.
- A un niveau intermédiaire, l'utilisateur d'un langage à parallélisme de données ne doit spécifier que l'implantation des données et le parallélisme. Le compilateur se charge d'en déduire les communications et la répartition des calculs. On peut même se convaincre (section 3.4) que l'implantation des données à elle seule permet de retrouver le parallélisme, mais l'information supplémentaire permet de simplifier le travail du compilateur. En théorie, ce travail de construction des communications peut être fait pour n'importe quel programme ; en pratique, le résultat n'est efficace que si le programme respecte les contraintes du modèle polyédrique.
- L'utilisateur d'un paralléliseur automatique ne fournit qu'une réalisation séquentielle de son algorithme. Celle-ci joue le rôle, pour le paralléliseur, d'une spécification : c'est à lui d'en extraire le parallélisme et, le cas échéant, de trouver la bonne implantation des données. La construction du code objet se poursuit de la même façon que pour un langage à parallélisme de données. Ici encore, ce travail ne peut être mené à bien que si le programme respecte les contraintes du modèle polyédrique. Il y a également une issue de secours, qui consiste à conserver tout ou partie du code séquentiel s'il s'avère trop difficile à paralléliser. L'étude des méthodes de parallélisation partielles, locales et interprocédurales est un sujet de recherche prometteur.
- Enfin, dans les langages de très haut niveau, le compilateur ne peut pas recourir à la version séquentielle en cas de problème. Son travail n'en est rendu que plus difficile. On peut, par contre, définir des modèles d'exécution parallèle qui peuvent être confiés au logiciel – le *run-time system* – ou au matériel.

5.2- A quand la crise du logiciel en parallélisme ?

L'expérience passée, en programmation séquentielle, montre qu'après une première phase de programmation très bas niveau, expliquée par la pauvreté des environnements et le manque de recul, mais justifiée par des contraintes d'efficacité, on en vient à des modes d'expression de haut niveau qui s'abstraient des spécificités de l'architecture : la performance des processeurs, la multiplication des espaces mémoire, et les progrès des compilateurs font que la différence d'efficacité se comble, au profit de codes plus facilement portables et maintenables.

Cette évolution résulte, en général, d'une "crise du logiciel". Verra-t-on se développer une crise du logiciel en parallélisme ? Peut-être y est-on déjà. Sans plus attendre, nous pourrions concevoir ces hauts niveaux de langage que nous évoquions ci-dessus – les aspects de virtualisation mentionnés sont une voie intéressante à poursuivre – et nous tourner vers les architectes de machines et les écrivains de compilateurs pour les rendre attractifs.

5.3- Echappera-t-on à un travail de conception de nouveaux algorithmes ?

Tout d'abord, il faut bien comprendre que les recherches sur les algorithmes et sur la programmation parallèle sont des sujets indépendants. Un algorithme n'est ni séquentiel ni parallèle ; ce n'est qu'au moment de l'implémenter que l'on peut savoir s'il se parallélise bien ou mal. Ce travail d'implémentation peut être plus ou moins assisté par ordinateur : la parallélisation "à la main" risque d'être toujours meilleure que celle effectuée par un compilateur, de même qu'un programme écrit en langage machine est souvent plus efficace que celui généré par un compilateur. Mais qui programme encore en langage machine de nos jours ?

Qu'il s'agisse de parallélisme de contrôle ou de parallélisme de données, on ne pourra certainement pas à l'avenir se suffire d'utiliser uniquement :

- des bibliothèques scientifiques,
- des bibliothèques de communication ou des systèmes de mémoire partagée virtuelle,
- des paralléliseurs automatiques,

qui sont finalement adaptés à la réutilisation vaille que vaille des codes existants.

De même qu'on ne pouvait pas se contenter dans les années 80 de limiter la programmation parallèle à l'instillation de synchronisations entre processus – il a fallu penser à des notions de plus haut niveau pour porter les codes rationnellement –, il faut maintenant franchir une seconde étape – industriellement délicate – et développer des programmes spécifiques adaptés aux machines massivement parallèles, et certainement liés à des modèles physiques et des méthodes de simulation nouvelles.

Pour en revenir aux langages, pour ce faire, il faudra certainement imaginer de nouveaux formalismes – peut-être plus "déclaratifs" – certainement plus proches du niveau d'expression des problèmes : des langages fonctionnels ? des langages à flots de données ? des langages à équations ? Et pourquoi pas APL ?

6- Références

[ANS90] ANSI X3J3/S8.115, Fortran 90, 1990.

[AFR80] Apt K.R., Francez N., De Roever W.P., A Proof System for Communicating Sequential Processes, ACM TOPLAS, 2, 3, 359-385, 1980.

[AlK87] Allen J.R., Kennedy K., Automatic Translation of Fortran Programs to Vector Form, ACM TOPLAS, 9, 491-542, 1987.

[Bab88] Babb R., Programming Parallel Processors, Addison-Wesley Pub., 1988.

[Coe96] Coelho F., Contribution à la compilation du High Performance Fortran, Thèse, Ecole des Mines de Paris, 1996.

[Fea96] Feautrier P., Automatic Parallelization in the Polytope Model, in [PeD96], 79-103.

[Fos95] Foster I., Designing and Building Parallel Programs, Addison-Wesley Pub., 1995.

[Fur95] Le Fur M., Scanning Parametrized Polyhedron using Fourier-Motzkin Elimination, High Performance Computing Symposium, 130-143, 1995.

[Ger89] Gerndt H.M., Automatic Parallelization for Distributed Memory Multiprocessing Systems, Thèse, Bonn, 1989.

[Hoa78] Hoare C.A.R., Communicating Sequential Processes, Communications of the ACM, 21, 8, 666-677, 1978.

[HPF93] High Performance Fortran Forum, HPF Language Specification, 1993.

[KLS94] Koelbel C., Loveman D., Schreiber R., Steel G., Zosel M., The High Performance Fortran Handbook, The MIT Press, 1994.

[MPI93] Message Passing Interface Forum, MPI: a Message Passing Interface, SuperComputing'93, 878-883, IEEE Computer Society Press, 1993.

[Occ88] Occam2 Reference Manual, Inmos Limited, Prentice Hall International Series in Computer Science, 1988.

[Paz96] Pazat J.L., Tools for High Performance Fortran, a survey, in [PeD96], 134-158.

[PeD96] Perrin G.R., Darte A., The Data Parallel Programming Model: Foundations, HPF Realizations and Scientific Applications, Lecture Notes in Computer Science, Tutorial Series, 1132, Springer Verlag, 1996.

[Per87] Perrot R.H., Parallel Programming, International Computer Science Series, Addison-Wesley Pub., 1987.

[Sch96] Schreiber R., An Introduction to HPF, in [PeD96], 27-44.

[TMC93] Thinking Machines Corporation, CM-Fortran Reference Manual, 1993.