50 376
1992
19

USTL
FLANDRES ARTOIS

LIFL
UA 369 du CNRS

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

Numéro d'ordre : 863

Thèse présentée à

L'UNIVERSTITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE DE L'UNIVERSITÉ DE LILLE

sur le sujet

Langages explicites à parallélisme de données

par

Philippe MARQUET

Thèse soutenue le 6 février 1992
devant la commission d'examen composée de

Président
Vincent CORDONNIER, LIFL, Université de Lille

Rapporteurs
Luc BOUGÉ, LIP, Ecole Normale Supérieure de Lyon
Paul FEAUTRIER, MASI, Université Pierre et Marie Curie, Versailles

Examinateurs
Jean-Luc DEKEYSER LIFL, Université de Lille
Michel MÉRIAUX, LIFL, Université de Lille
Nicolas PARIS, LIENS, Ecole Normale Supérieure, Paris
Serge PETITON, Site Expérimental en Hyperparallélisme, ETCA, Arceuil

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PAREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CONSTANT Eugène	Electronique
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie Appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean-Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilitéss
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BREZINSKI Claude	Analyse Numérique

M. BRIDOUX Michel	Chimie-Physique
M. CELET Paul	Géologie Générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. DAUCHET Max	Informatique
M. DEBOURSE Jean-Pierre	Gestion des Entreprises
M. DHAINAUT André	Biologie Animale
M. DOUKHAN Jean-Claude	Physique du Solide
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
M. FAURE Robert	Mécanique
M. FOCT Jacques	Métallurgie
M. FRONTIER Serge	Ecologie Numérique
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. LABLACHE-COMBIER Alain	Chimie Organique
M. LACOSTE Louis	Biologie Végétale
M. LAVEINE Jean-Pierre	Paléontologie
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique Atomique et Moléculaire
M. LEROY Jean-Marie	Spectrochimie
M. LHOMME Jean	Chimie Organique Biologique
M. LOMBARD Jacques	Sociologie
M. LOUCHEUX Claude	Chimie Physique
M. LUCQUIN Michel	Chimie Physique
M. MACKE Bruno	Physique Moléculaire et Rayonnements Atmosph.
M. MIGEON Michel	E.U.D.I.L.
M. PAQUET Jacques	Géologie Générale
M. PETIT Francis	Chimie Organique
M. POUZET Pierre	Modélisation - calcul Scientifique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. SALMER Georges	Electronique
M. SCHAMPS Joel	Spectroscopie Moléculaire
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. TILLIEU Jacques	Physique Théorique
M. TOULOTTE Jean-Marc	Automatique
M. VIDAL Pierre	Automatique
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne	Composants Electroniques
M. ANDRIES Jean-Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BART André	Biologie animale
M. BASSERY Louis	Génie des Procédés et Réactions Chimiques

Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique Atomique et Moléculaire
M. BERTRAND Hugues	Sciences Economiques et Sociales
M. BERZIN Robert	Analyse
M.BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie Végétale
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie Civil
M. BOIVIN Jean-Claude	Spectroscopie
M. BOUQUELET Stéphane	Biologie Appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BRASSELET Jean-Paul	Géométrie et Topologie
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie Animale
M. CATTEAU Jean-Pierre	Chimie Organique
M. CAYATTE Jean-Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie Structurale
M. CHIVE Maurice	Composants Electroniques Optiques
M. COMYN Gérard	Informatique Théorique
M. COQUERY Jean-Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques et Sociales
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique Nucléaire et Corpusculaire
M. COUTURIER Daniel	Chimie Organique
M. CRAMPON Norbert	Tectolique Géodynamique
M. CROSNIER Yves	Electronique
M. CURGY Jean-Jacques	Biologie
Melle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	Géologie Appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean-Paul	Informatique
M. DELORME Pierre	Physioliogie Animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
M. DENEL Jacques	Informatique
M. DE PARIS Jean Claude	Analyse
M. DEPREZ Gilbert	Physique du Solide - Cristallographie
M. DERIEUX Jean-Claude	Microbiologie
Melle DESSAUX Odile	Spectroscopie de la réactivité Chimique
M. DEVRAINNE Pierre	Chimie Minérale
Mme DHAINAUT Nicole	Biologie Animale
M. DHAMELINCOURT Paul	Chimie Physique
M. DORMARD Serge	Sciences Economiques
M. DUBOIS Henri	Spectroscopie Hertzienne
M. DUBRULLE Alain	Spectroscopie Hertzienne
M. DUBUS Jean-Paul	Spectrométrie des Solides
M. DUPONT Christophe	Vie de la firme (I.A.E.)
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FAUQUAMBERGUE Renaud	Composants électroniques

M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie Sturcturale
M. GAMBLIN André	Géographie urbaine, industrielle et démog.
M. GLORIEUX Pierre	Physique moléculaire et rayonnements Atmos.
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie Physique
M. GOURIEROUX Christian	Probabilités et Statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean-Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie Organique
M. GUILBAULT Pierre	Physiologie animale
M. HENRY Jean-Pierre	Génie Mécanique
M. HERMAN Maurice	Physique spatiale
M. HOUDART René	Physique atomique
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et Statistiques
M. Jean Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme (I.A.E.)
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KREMBEL Jean	Biochimie
M. LANGRAND Claude	Probabilités et statistiques
M. LATTEUX Michel	Informatique
Mme LECLERCQ Ginette	Catalyse
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
Melle LEGRAND Solange	Algèbre
M. LEGRAND Pierre	Chimie
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne
M. LE MAROIS Henri	Vie de la firme (I.A.E.)
M. LEROY Yves	Composants électroniques
M. LESENNE Jacques	Systèmes électroniques
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique théorique
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU Jean-Marie	Optique-Physique atomique
M. MAIZIERES Christian	Automatique
M. MAURISSON Patrick	Sciences Economiques et Sociales
M. MESMACQUE Gérard	Génie Mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie Organique
M. MORTREUX André	Chimie Organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
Mme MOUYART-TASSIN Annie Françoise	Informatique
M. NICOLE Jacques	Spectrochimie
M. NOTELET Françis	Systèmes électroniques
M. PARSY Fernand	Mécanique

M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. STEEN Jean-Pierre	Informatique

A Ariane et ses "mecs"

Je tiens à remercier

Vincent Cordonnier d'avoir accepté la présidence de ce jury,

Luc Bougé et Paul Feautrier d'avoir rapporté cette thèse; je leur suis reconnaissant de leur lecture attentive et de leurs remarques constructives, je leur suis également reconnaissant de leur accueil au sein du groupe de travail "C3simd",

Michel Mériaux pour son soutien en tant que directeur du laboratoire,

Nicolas Paris de l'intérêt qu'il a manifesté pour ce travail en acceptant de faire partie du jury,

Serge Petiton pour son attention particulière portée à nos travaux,

Les travaux présentés sont aussi ceux d'une équipe, je tiens particulièrement à remercier Jean-Luc Dekeyser et Philippe Preux avec qui les travaux présentés ici furent menés. Je remercie Jean-Luc pour le style de sa direction de l'équipe et pour sa confiance. Je salue l'enthousiasme et l'ouverture d'esprit de Philippe. Je remercie aussi vivement les autres membres de l'équipe, Akram, Dominique et Tahar.

Je tiens également à remercier tous ceux qui font la vie et l'ambiance du laboratoire.

Table des matières

Introduction

Dans de nombreux domaines, des applications ont des besoins importants de calcul. Derrière la banalité de cette affirmation, nous découvrons des scientifiques dont la résolution des problèmes passe par un nombre important de traitements numériques. Typiquement, ces traitements sont effectués sur des structures de données telles des vecteurs ou des matrices. Ces traitements identifient en eux-mêmes le *parallélisme de données*.

Historiquement, le codage de ces traitements est réalisé en FORTRAN scalaire (FORTRAN 77). Les avantages de FORTRAN sont clairs; ils sont liés à la bonne qualité du code obtenu et à la portabilité sur un nombre important de machines. Cependant, FORTRAN 77 n'autorise pas la manipulation de données parallèles en tant que telles. Deux conséquences en sont, 1—la traduction imposée au programmeur manipulant des données parallèles pour exprimer les opérations sur ces données en FORTRAN, et 2—la nécessité d'une phase de vectorisation, traitement préliminaire à la compilation et chargé de reconnaître automatiquement les constructions parallèles dans le code scalaire.

Bien que débattue depuis plus de 10 ans, récemment, une nouvelle norme FORTRAN (FORTRAN 90) était adoptée. Les vecteurs sont manipulés en tant que tels en FORTRAN 90. Cette nouvelle norme a été précédée de nombreux langages et propositions de langages à parallélisme de données. Il nous semble que les travaux dans cette voie doivent être poursuivis. En particulier, la notion de *langage explicite à parallélisme de données* ne recouvre pas la seule manipulation explicite de données parallèles, mais intègre aussi les moyens pour le programmeur d'exprimer les informations connues de lui sur ces données, et pouvant être prises en compte par les compilateurs pour la production de code de bonne qualité. C'est dans cette perspective que nous proposons nos travaux.

Les nombreux calculs des applications précédemment nommées demandent à être exécutés rapidement. Actuellement, deux types d'architectures sont capables de fournir des performances de l'ordre de celles attendues. Il s'agit de celle des machines pipelines vectorielles et de celle des machines massivement parallèles. Ces deux classes de machines sont regroupées sous le terme de *machines à parallélisme de données*. Jusqu'alors, peu de travaux les ont considérées sous un angle unique. Cependant, cette approche est facilitée par nombre de points communs entre les modes de fonctionnement de ces deux types d'architectures. C'est dans cet esprit que nous développons nos travaux.

Le document regroupe trois chapitres. Le premier chapitre contient un état de l'art de la programmation à parallélisme de données; le chapitre 2 constitue une présentation de nos travaux de conception et réalisation d'un langage vectoriel, EVA; le chapitre 3 traite de nos travaux autour de LSD2, langage à parallélisme de données pour les machines pipelines vectorielles et les machines massivement parallèles. Les grandes lignes de nos propositions de langages sont basées sur des structures de données vecteurs originales.

Nos travaux sont délibérément orientés vers les aspects de programmation des machines à parallélisme de données. Le premier chapitre introduit l'existant en ce domaine. Nous commençons par une présentation des machines visées par nos travaux, machines pipelines vectorielles et machines massivement parallèles. Nous comparons ensuite les deux types de machines et mettons en évidence des analogies. Ces analogies autorisent une approche unique pour la programmation des machines à parallélisme de données. Nous présentons ensuite les différentes approches de programmation de ces machines. Nous insistons en particulier sur les langages explicites, c'est-à-dire les langages autorisant 1—l'expression par le programmeur des connaissances qu'il détient sur son algorithme et, 2—la manipulation directe des données traitées. Enfin, nous traitons de l'implantation de ces langages sur les machines.

Deux parties composent le chapitre 2. Chacune des parties présente un aspect de nos travaux dans le cadre du langage EVA. En effet, deux aspects ont motivé ces travaux. Nous désirons proposer des outils explicites d'expressions des algorithmes vectoriels, la première partie présente donc les constructions du langage EVA qui vont dans cette direction; ces constructions sont basées sur une notion puissante de vecteur et sur la manipulation des objets vecteurs. La seconde partie justifie la proposition d'une telle structure de données vecteur, et des autres constructions et outils disponibles en EVA, par des considérations d'efficacité de l'implantation. Cette partie traite de la projection de EVA sur machines pipelines vectorielles via le langage vectoriel intermédiaire DEVIL.

Le dernier chapitre est la présentation d'une deuxième phase de nos travaux, autour de la proposition du langage à parallélisme de données LSD2. Le langage LSD2 vise en effet les deux classes de machines, pipelines vectorielles et massivement parallèles. Une première partie décrit le langage LSD2 et présente la notion de vecteur introduite dans ce langage. Ce vecteur est une évolution du vecteur EVA. Il encapsule les spécifications fournies par le programmeur, dont la distribution, l'alignement, et les dépendances entre les différents accès aux éléments du vecteur. A partir des ces dépendances, il est possible de générer du code efficace. L'étude de ces dépendances et la génération de code associée sont l'objet de la seconde partie de ce chapitre.

Chapitre 1

Langages et expression du parallélisme de données

1-1 Introduction

Une demande de puissance de calcul toujours plus grande existe dans de nombreux domaines d'application. Les diverses avancées satisfaisant cette demande ne font que justifier la pression toujours plus forte en matière de performances. Si les progrès les plus spectaculaires sont enregistrés par les avancées technologiques, bien d'autres aspects contribuent à l'augmentation des performances : architectures, compilateurs et logiciels de base, langages et environnements de programmation. Nos travaux sont orientés vers des propositions de langages pour la programmation à parallélisme de données et l'implantation de ces langages sur les machines à parallélisme de données.

Les machines sont classiquement partagées suivant les flux de données et de contrôle mis en œuvre lors de leur fonctionnement. Les machines à parallélisme de données sont des machines SIMD (Single Instruction stream — Multiple Data streams). Les machines à parallélisme de données recouvrent les machines vectorielles pipelines bien que celles-ci soient selon les auteurs présentées comme SISD, SIMD ou MIMD.

Ce chapitre présente les divers aspects du domaine dans lequel nous travaillons. Un rappel examine rapidement les caractéristiques des machines visées par nos études. Nous passons ensuite en revue les langages qui ont été proposés pour la programmation vectorielle. Enfin, nous nous intéressons aux techniques de compilation mises en œuvre pour implanter ces langages.

Ce chapitre ne se veut pas une présentation exhaustive du traitement vectoriel. Pour une approche plus générale du domaine, citons les ouvrages de Hockney et Jesshope [HoJe81, HoJe88], Hwang et Briggs [HwBr84], Kuck [Kuck78], Stone [Ston87] comme des références sérieuses et beaucoup plus complètes du traitement vectoriel et parallèle.

1–2 Les machines vectorielles, influences sur la programmation

Nos travaux sont orientés vers la conception de langages pour la programmation vectorielle. Deux types d'architecture mettent en œuvre le parallélisme au niveau des données : les machines vectorielles pipelines et les machines vectorielles tableaux (machines tableaux "traditionnelles" et machines massivement parallèles SIMD). Dans les sections suivantes, nous ne présentons pas l'architecture des machines en tant que telle, mais nous nous intéressons aux caractéristiques de l'architecture de ces machines qui sont importantes pour nous et qui devraient être prises en compte lors de la programmation de ces machines (soit au niveau du langage, soit au niveau du programmeur). Nous nous efforcerons aussi de présenter les analogies entre ces deux types de machines. Ces analogies font qu'il est possible de concevoir des outils de programmation communs aux deux types d'architecture et d'utiliser des techniques de compilation similaires pour toutes ces machines.

1–2.1 Les machines vectorielles pipelines

Les machines vectorielles pipelines sont commercialement disponibles depuis le milieu des années 70. Suite au CDC 7700, les premières machines vectorielles pipelines furent les Cray 1 et Cyber 205 (1976); les machines les plus récentes sont les Cray Y-MP (voire Cray C-90), NEC SX-3, etc... Nous listons quelques points remarquables des machines vectorielles pipelines. La figure 1–1 représente l'architecture type d'une machine vectorielle pipeline. Cette figure ne représente pas l'architecture d'une machine précise; on y retrouve les grands traits de l'architecture de base d'une machine vectorielle pipeline.

On distingue la mémoire et les processeurs. Un processeur est constitué d'un contrôleur déclenchant l'activité d'unités fonctionnelles diverses s'alimentant dans des registres scalaires et vectoriels et y produisant leurs résultats. Les registres masque et VL (Vector Length) sont utilisés comme registres d'état pour contrôler l'activité des unités vectorielles. Les registres sont chargés depuis la mémoire via un ou plusieurs ports mémoire. Les processeurs communiquent entre eux via la mémoire et/ou des registres (mécanisme de communication inter-processeurs).

1–2.1.1 Registres vectoriels

Les machines vectorielles pipelines sont des machines à registres. Les opérandes (scalaires et/ou vectorielles) sont chargés dans les registres (scalaires et/ou vectoriels) depuis la mémoire; l'exécution produit son résultat dans un registre; le contenu du registre résultat est rangé en mémoire.

Notons tout de même l'exception que furent les machines Control Data Cyber 205 et ETA 10. Les instructions vectorielles de ces machines opéraient à partir d'opérandes vectoriels détenus en mémoire.

Typiquement un registre vectoriel peut contenir 64 éléments (par exemple sur les machines Cray actuelles). Le traitement de vecteurs plus longs est réalisé par découpage des vecteurs en "morceaux" de 64 éléments. Ce découpage est nommé *strip-mining*. Il est en général réalisé par le compilateur.

1–2.1.2 Lecture et écriture mémoire

Les opérations de lecture et écriture mémoire effectuent les transferts des éléments des vecteurs dans les registres vectoriels depuis et vers la mémoire. Ces éléments peuvent être contigus en mémoire (transferts

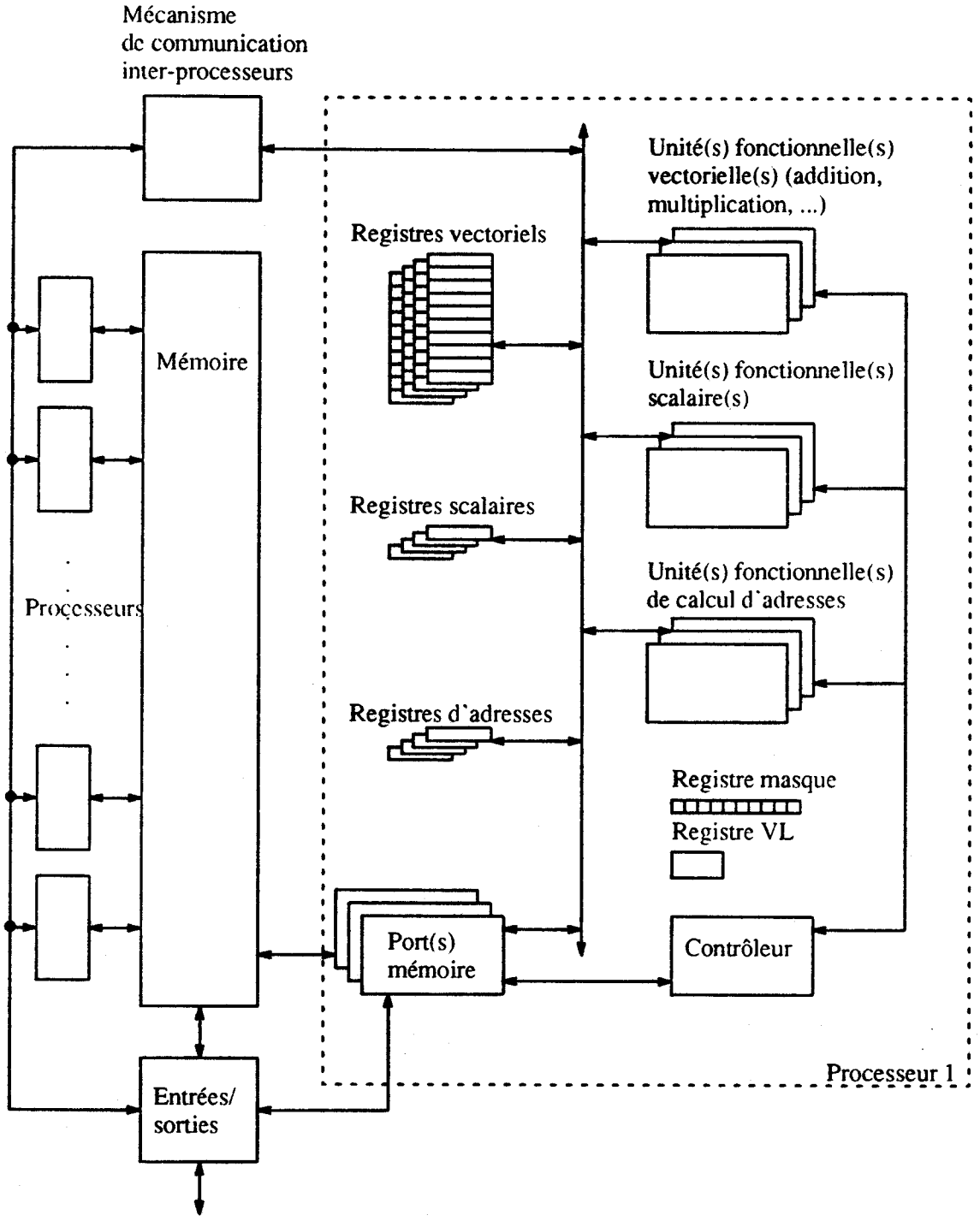


Figure 1–1. Architecture type d'une machine vectorielle pipeline.

traditionnels), ou peuvent être espacés d'un pas constant. Ces transferts peuvent être contrôlés par un masque de bits ou par une liste d'index. Une lecture contrôlée par un masque de bits est appelée *compress*, une écriture contrôlée par un masque est un *decompress* ou *extend*. Une lecture contrôlée par une liste d'index est un *gather*, une écriture est un *scatter*. Les machines ne disposent pas toutes de l'ensemble des instructions de lecture/écriture. Par exemple, les Cray 1 et les premiers modèles de Cray X-MP ne disposent ni d'instructions de *gather* et *scatter* ni de *compress/extend*.

Les opérations inexistantes doivent être simulées au niveau logiciel. Deux approches sont possibles. 1—Le langage de haut niveau inclut ces opérations et le compilateur est à charge de les implanter. 2—Le langage de haut niveau reflète l'architecture de la machine et ne propose pas les outils au programmeur qui doit soit changer son algorithme (si c'est possible), soit implanter les opérations à son niveau. Une telle implantation de haut niveau est bien souvent inefficace.

1-2.1.3 Mémoire entrelacée

Les nouvelles générations de machines vectorielles pipelines sont multi-processeurs. Les processeurs peuvent être multi-ports (Cray X-MP et Y-MP par exemple). La mémoire est partagée entre les processeurs. Ces machines ont donc besoin d'une mémoire rapide et si possible parallèle (multi-accès simultané). Les mémoires sont en général des mémoires découpées en bancs; les bancs sont regroupés en sections (parfois en sous-sections qui sont elles-mêmes regroupées en sections : Cray Y-MP). Si ces caractéristiques augmentent la bande passante de la mémoire, elles sont aussi à l'origine de conflits d'accès qui entraînent des dégradations importantes des performances. Par exemple lors de l'accès à un vecteur avec un pas égal au nombre de bancs, tous les éléments du vecteur se trouvent dans le même banc; aucun parallélisme entre les différents accès ne peut donc être obtenu.

Le programmeur prend cette caractéristique en compte par exemple en évitant, par une allocation judicieuse, que tous les éléments d'une colonne de matrice soient alloués dans un même banc mémoire :

```
DIMENSION TAB_CONFL (LIGN, NB_BANC_MEMOIRE)
```

alloue tous les éléments d'une colonne dans un même banc. La déclaration d'une colonne supplémentaire inutilisée par le programme évite tout conflit mémoire lors d'accès à une colonne :

```
DIMENSION TAB_NO_CONFL (LIGN, NB_BANC_MEMOIRE + 1)
```

1-2.1.4 Mémoire hiérarchisée

Relativement peu de calculateurs vectoriels pipelines possèdent une mémoire hiérarchisée. Toutefois, les processeurs des machines IMB 3090 VF et ETA-10 sont munis d'un cache local. Notons aussi que chaque processeur du Cray 2 possède une mémoire locale (très rapide par rapport à la mémoire centrale) dont il est difficile de tirer parti efficacement. Si l'on ne s'en sert que comme zone de sauvegarde temporaire des registres, elle risque d'être sous-utilisée; si l'on veut y allouer des objets, il est difficile de déterminer les objets dont l'allocation dans cette mémoire sera la plus profitable.

Remarquons l'organisation de la machine Cedar développée au CSRD de l'université d'Urbana-Champaign (Illinois). Une machine Cedar est la réunion de *clusters* partageant une mémoire

commune. Chaque *cluster* est un ensemble de processeurs vectoriels disposant d'une mémoire commune propre au *cluster*. Enfin chaque processeur vectoriel est associé à une mémoire locale. L'utilisation optimale d'une telle organisation de mémoires n'est bien évidemment pas sans difficulté. Cedar FORTRAN définit trois niveaux d'allocation des variables et de gestion des boucles DO associés aux trois niveaux de mémoire de la machine. De plus, une reconnaissance automatique des différents degrés de parallélisme à partir d'un code FORTRAN scalaire produit du code Cedar FORTRAN. La programmation d'une telle machine (tant au point de vue programmeur que compilateur vectorisant/parallélisant) semble devoir se faire tout d'abord en pensant vectoriel puis en parallélisant le code obtenu ou du moins en privilégiant l'approche vectorielle par rapport à l'approche parallèle [Peti88].

1–2.1.5 Multiplicité des unités fonctionnelles

Les machines vectorielles pipelines sont constituées de plusieurs unités fonctionnelles (UFs). Cette caractéristique est commune à la majorité des machines; citons le cas extrême du NEX SX-3 dont chaque processeur peut contenir jusqu'à 16 pipelines vectoriels arithmétiques (divisés en 4 ensembles de pipelines). Ces unités fonctionnelles peuvent travailler en parallèle. Les dépendances entre les traitements à réaliser et les différences entre les temps de traitement de chacune des unités peuvent amener des inactivités de certaines UFs, par exemple en attente de données depuis une autre UFs.

Cette multiplicité des unités fonctionnelles est prise en compte au sein des compilateurs par l'ordonnancement du code lors de la phase de génération de code. D'autres outils restructurent le code généré en fonction des caractéristiques des UFs; citons les travaux de Babb autour de SARA sur Cray X-MP et Y-MP [Babb89].

1–2.1.6 Chaînage des unités fonctionnelles

Une des caractéristiques importantes des machines vectorielles pipelines est la possibilité de chaînage des unités fonctionnelles pipelinées. Le chaînage des pipelines est caractérisé par le fait qu'une instruction vectorielle utilise un résultat produit par une instruction antérieure avant que cette instruction antérieure n'aie terminé son exécution. Cette fonctionnalité est disponible sur la plupart des machines actuelles (Cray, Fujitsu, NEC, etc...). Une exception notable parmi les machines récentes est le Cray 2.

Ce chaînage des unités fonctionnelles est en général géré par un dispositif matériel de l'architecture de la machine et n'apparaît donc pas au niveau du code assembleur. Notons toutefois l'instruction `link` de l'assembleur de l'ETA 10 qui indiquait que les deux instructions suivantes devaient être chaînées. Ce chaînage des unités fonctionnelles doit être pris en compte lors de la génération de code, et plus particulièrement lors de la phase d'ordonnancement du code afin que le dispositif matériel puisse déclencher le chaînage.

1–2.1.7 Masquage des opérations

Une caractéristique intéressante des machines est la possibilité de contrôler l'exécution d'une opération vectorielle par un vecteur de bits. Les machines possèdent alors un (ou plusieurs) registre(s) contenant un masque de la longueur d'un registre vectoriel (registres *MR0* à *MR255* sur Fujitsu, registre *VM* sur Cray).

Sur Fujitsu, une opération est exécutée sur toutes les composantes des vecteurs sources, mais seules les composantes dont le bit d'un vecteur masque est à 1 sont rangées dans le vecteur cible; les autres composantes gardent leur valeur d'avant l'opération.

Ce masquage de certaines composantes est utilisé pour l'exécution de bloc **WHERE**, extension vectorielle du `IF` dans lequel le traitement sur chaque composante est lié à la valeur d'un bit d'un vecteur de contrôle.

1-2.1.8 Multi-processeurs

Les premières machines vectorielles pipelines étaient mono-processeurs (Cray 1, Cyber 205). Control Data furent les premiers à introduire plusieurs processeurs : l'ETA 10 pouvait avoir jusqu'à 8 processeurs. Les machines Cray plus récentes sont également multi-processeurs (de 4 processeurs maximum sur les Cray X-MP et Cray 2, jusqu'à 8 processeurs sur le Cray Y-MP actuel, et même 16 sur les modèles de Cray C-90 à venir). Les constructeurs japonais proposent également des machines multi-processeurs (jusqu'à 4 processeurs sur le NEC SX-3 par exemple).

Deux approches tentent de bénéficier de cette multiplicité des processeurs; le *micro-tasking* et le *macro-tasking*. Elles se distinguent par le niveau de granularité du parallélisme obtenu.

Le *macro-tasking* consiste en un découpage d'un programme en tâches importantes. Le parallélisme est obtenu par des appels à des routines de création de tâches et de communications entre tâches entraînant des commutations de contexte.

Le *micro-tasking* consiste lui en un découpage à un niveau plus fin : celui des instructions FORTRAN par exemple. Typiquement, le *micro-tasking* partage les différentes itérations d'une boucle entre différents processeurs. Pour communiquer entre eux et se synchroniser, les différents processeurs partagent un ensemble de registres dédiés aux communications inter-processeurs (sur les Cray X-MP et Y-MP ou NEC SX-3 par exemple) ou communiquent via la mémoire et un ensemble de sémaphores (Cray 2). L'absence de registres de communications est bien entendu pénalisant.

Que ce soit au niveau *macro-* ou *micro-tâches*, la prise en compte de ce parallélisme est réalisée par le logiciel. Il n'est pas simple de confier la gestion d'un tel parallélisme au programmeur. Aussi des outils dit d'*auto-tasking* sont aujourd'hui proposés; ils sont chargés de la découverte automatique de portions de code pouvant être découpés en tâches [Nage90].

1-2.1.9 Notes bibliographiques

Les livres de Hwang et Briggs [HwBr84] et de Stone [Ston87] décrivent les principes généraux implantés dans les machines discutées ici. Les informations sur les architectures et les langages machines des supercalculateurs commercialisés sont disponibles dans les documentations des constructeurs : Cray [Cray87, Cray88a, Cray88b, Cray88c], Siemens et Fujitsu [Siem88a, Siem88b], NEC [NEC89], IBM [IBM88], Cyber et ETA [CDC87]. De nombreuses informations sur ces machines sont également disponibles dans les ouvrages de Hockney et Jesshope [HoJe81, HoJe88], de Hwang et Briggs [HwBr84] et de Schönauer [Schö87a]. L'architecture des machines Hitachi S-810 est présentée dans [LMM85, ONK86]. Une description de l'ETA 10 peut être trouvée dans [Schö87b]. La mémoire de l'Alliant FX/8 est détaillée dans [GGJ+90]. La machine Cedar est décrite dans [KDLS86] et [Jalb87], le langage FORTRAN

de la machine dans [Guzz87] et [GPHL90], et son compilateur dans [EHJD90]. Des études sur les conflits mémoire sont présentées dans [Cala89], [OeLa85], [RaHa90], et [TaMe89]. Des comparaisons entre des accès sur un ou plusieurs ports mémoire et les conflits résultants sont étudiées dans [Rein88]. Les parallélismes micro- et macro-tâches sont présentés dans [Nage88] et [Rein85]; l'autotasking sur Cray, découverte automatique de macro-tâches, est décrit dans [Nage90]. A propos des techniques d'ordonnancement de code dans le cadre des machines vectorielles, on se référera par exemple à [Arya85] ou [NPA88].

1–2.2 Les machines tableaux et massivement parallèles

La seconde grande famille des machines vectorielles est constituée des machines tableaux. L'architecture de ces machines date de la fin des années 1960 avec l'Illiac IV. Un nouvel attrait pour ce type d'architecture est aujourd'hui dû aux progrès réalisés en matière de technologie autorisant une implémentation plus performante de concepts relativement anciens. Le principe des machines tableaux repose sur la multiplication des unités de calcul (ou PEs—Processeurs Elémentaires). Chaque PE détient une partie des données à traiter. Le traitement des données est donc réalisé en parallèle par tous les PEs sous le contrôle d'un processeur maître (principe du SIMD). Un réseau d'interconnexion relie les PEs entre-eux et/ou les relie à la mémoire.

Les machines tableaux classiques sont l'Illiac IV de Burroughs, la machine MPP de Goodyear Aerospace et l'ICL, maintenant AMT, DAP. Les machines tableaux récentes, souvent dénommées machines massivement parallèles sont constituées d'une à quelques dizaines de milliers de processeurs élémentaires. Les machines représentatives sont les Connection Machine développées par Thinking Machines Corporation (CM-1, CM-2, et maintenant CM-200). La société MasPar Computer Corporation construit également des machines massivement parallèles qui sont maintenant distribuées par Digital Equipment (MasPar MP-1 ou DEC mpp). Un projet de machine massivement parallèle de la société Cray devrait aboutir d'ici peu.

La figure 1–2 est un petit historique des machines tableaux. Y apparaissent les influences des machines plus anciennes sur les nouvelles générations et les caractéristiques propres à chaque machine (basé sur document de MasPar Computer Corp.).

Nous présentons les grandes lignes de l'architecture d'une machine tableau. Ensuite, comme dans la section précédente concernant les machines vectorielles pipelines, nous listerons les caractéristiques essentielles des machines devant être prises en compte lors de la programmation. Cette prise en compte des caractéristiques est réalisée par le programmeur ou par le système de compilation.

1–2.2.1 Une machine tableau

Nous définissons ici les grandes lignes de l'architecture d'une machine tableau. Une machine tableau est constituée d'un grand nombre de PEs, d'un ou plusieurs séquenceurs et d'une machine hôte (figure 1–3). La machine hôte peut ne pas être unique, elle supporte le développement et la compilation des programmes; c'est typiquement une station de travail. Un séquenceur est chargé du contrôle d'une série de PEs, il communique les instructions et les données de la machine hôte vers tous les PEs et rassemble les résultats provenant des PEs. On peut imaginer que le contrôleur et la machine hôte soient regroupés en une

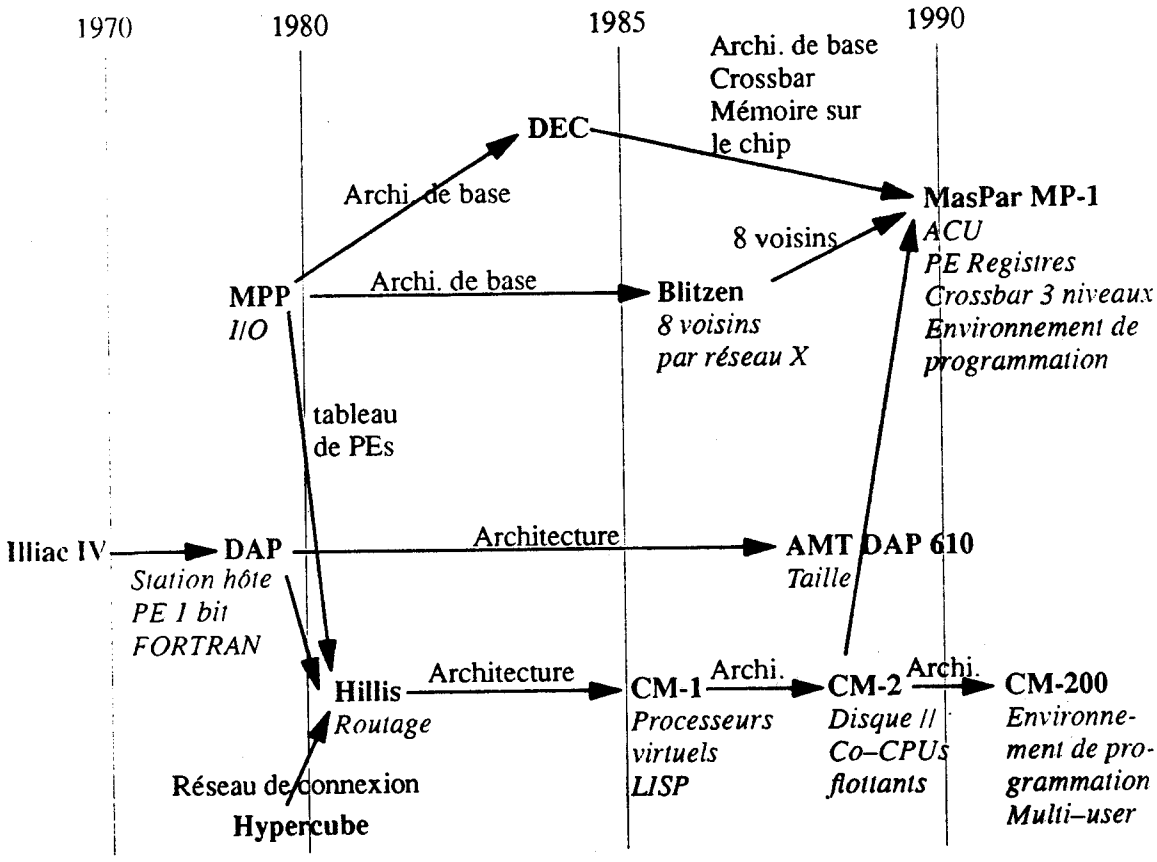


Figure 1–2. Généalogie des machines tableaux.

même unité. Sur les Connection Machine, ce contrôleur est nommé séquenceur, sur les machines MasPar, il est nommé ACU (Array Control Unit).

La simplicité des processeurs élémentaires autorise une multiplication de l'ordre des dizaines de milliers; les PEs de la CM sont des processeurs 1 bit, ceux de la MasPar traitent 4 bits. Notons que sur la CM, un processeur flottant de 32 bits est en option associé à chaque couple de nœuds de 32 PEs élémentaires. Une mémoire locale est attachée à chaque PE. Sur la MasPar et les processeurs flottants de la CM, des registres sont associés à chaque PE.

Les processeurs sont reliés par un réseau de communications. Ce réseau est souvent constitué de plusieurs niveaux. Un premier niveau connecte chaque PE à ses voisins (4 voisins par le réseau NEWS de la CM, 8 voisins par le XNet sur MasPar). Un autre niveau autorise des connexions arbitraires entre deux PEs. Ces communications sont réalisées par trois niveaux de crossbars formant le Global Router sur MasPar, et par un hypercube de dimension 12 reliant des nœuds de 16 PEs sur Connection Machine. Le contrôleur est directement relié à chaque PEs, cette liaison est utilisée pour la diffusion d'informations (généralement via un bus) et la collecte d'états (via le OR-tree sur MasPar, via le Combine et le Global Result Bus sur CM).

Deux niveaux d'entrées/sorties sont distingués; les entrées/sorties scalaires traditionnelles et les entrées/sorties parallèles (au travers les Parallel Disk Arrays de la MasPar ou le Data Vault de la Connection Machine).

1-2.2.2 Un processeur scalaire ?

L'architecture des machines tableaux est basée sur la multiplication du nombre de PEs. Cette nécessité de disposer d'un grand nombre de PEs limite la complexité de chacun d'entre eux. Chaque PE pris individuellement est donc relativement peu performant. Si un ensemble de PEs est bien adapté aux traitements de données telles les tableaux, la réalisation d'une opération sur une donnée scalaire ne peut tirer parti de la multiplication des PEs. Le processeur de la machine hôte ou le séquenceur est généralement d'une technologie plus complexe et ainsi plus à même de réaliser efficacement le traitement d'une donnée scalaire. (Ce traitement est réalisé sur la machine hôte des Connection Machine, sur l'ACU des machines MasPar).

Si une donnée scalaire doit être utilisée en conjonction avec une donnée tableau, la donnée scalaire sera transférée depuis la machine scalaire vers les PEs. Il est inutile en général de copier les données scalaires (variables ou constantes) dans la mémoire de chacun des PEs, ces données pouvant être diffusées à peu de frais à tous les PEs.

1-2.2.3 Accès mémoire — Communications

La mémoire est répartie entre les PEs. L'adressage de cette mémoire est faite de manière SIMD : chaque PEs accède un même emplacement dans sa mémoire locale. Toutefois, un adressage indirect local est aussi possible dans cette mémoire sur MasPar MP-1 par exemple; cela augmente la souplesse d'utilisation de la machine.

Le traitement de données réparties dans ces mémoires locales (typiquement un tableau dont chaque PE détient un élément) entraîne des communications entre les PEs pour accéder à des emplacements mémoire détenus par les autres PEs. On distingue 4 types de communications (voir par exemple [LiCh91]) :

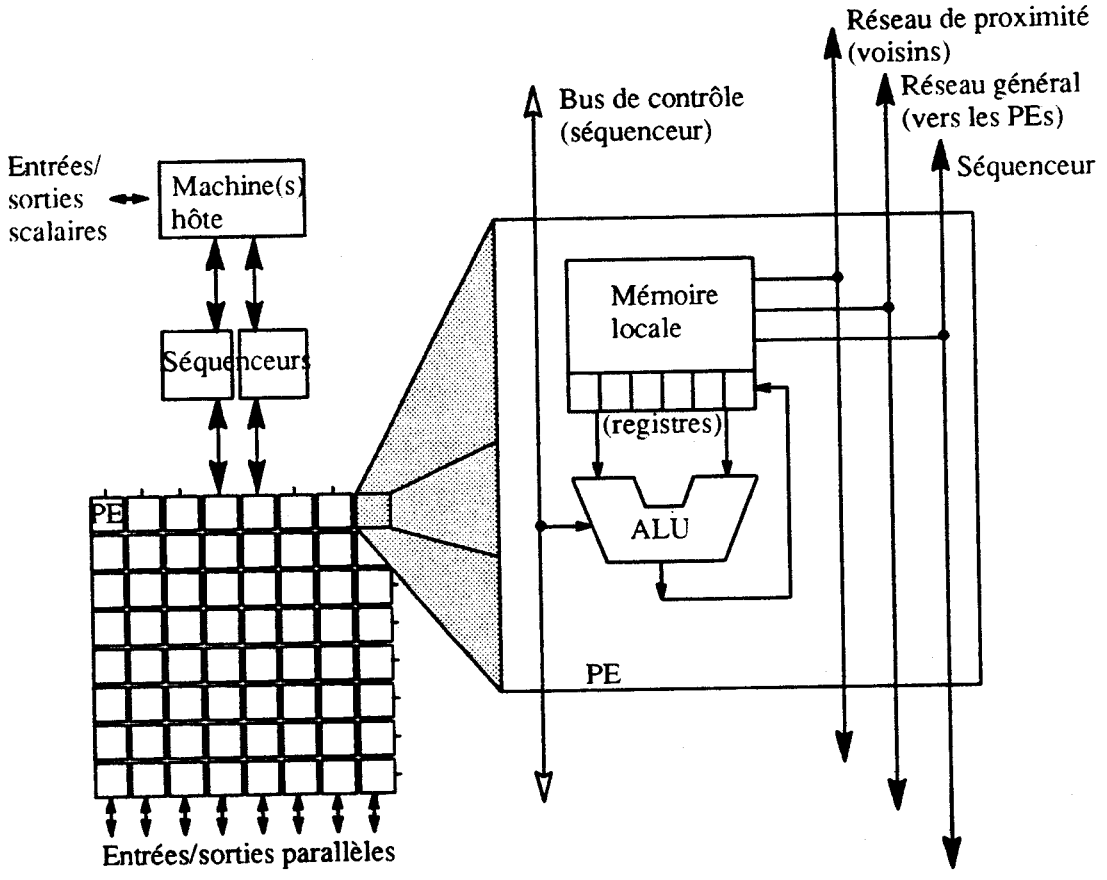


Figure 1-3. Architecture d'une machine tableau.

1) *Communication uniforme* Chaque processeur communique avec un processeur unique et distinct; de plus les couples émetteur/récepteur sont caractérisés par un même schéma spatial. Par exemple, chaque PE envoie une donnée à son voisin de gauche (figure 1–4.a).

2) *Communication point à point* ou *Permutation* Une communication point à point est une communication uniforme pour laquelle les communications ne sont pas toutes construites sur le même modèle. On assure cependant que chaque processeur qui communique le fait vers un processeur unique et distinct (figure 1–4.b).

3) *Communication générale* C'est une communication entre PEs que rien ne caractérise particulièrement. Un PE peut être destinataire de plusieurs messages (figure 1–4.c).

4) *Communication de réduction/diffusion* Un PE unique communique avec tous les autres. On distingue la diffusion—un PE émet un message à tous les autres, et la réduction—un PE reçoit un message de tous les autres (figure 1–4.d).

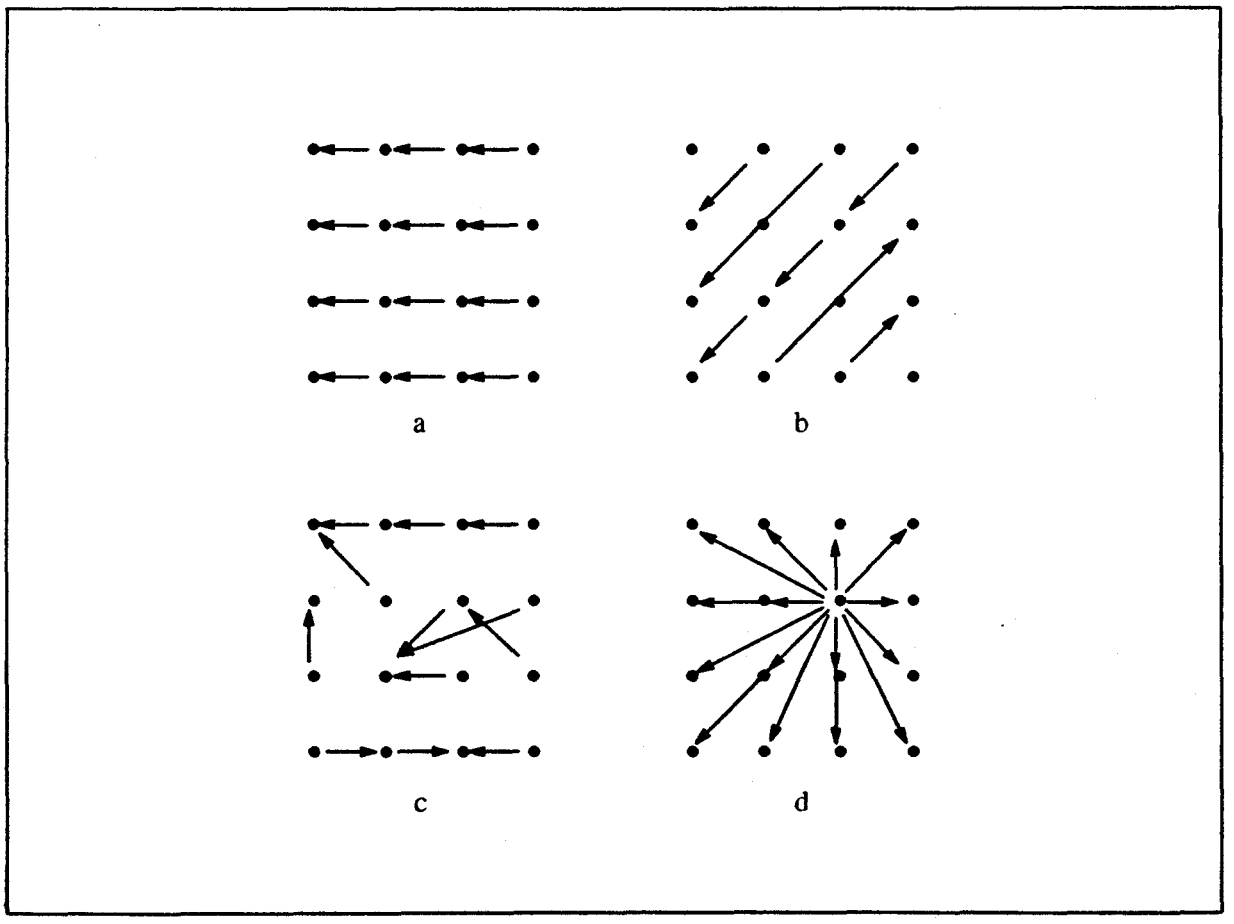


Figure 1–4. Quatre types de communications inter-PEs.

Une opération de réduction des valeurs est associée aux communications pouvant amener un PE à recevoir plusieurs valeurs. La valeur obtenue pour un processeur est calculée localement par composition par cete opération des valeurs reçues. Ces opérations sont préférablement associatives et commutatives. Cependant, l'opération par défaut retient généralement la dernière valeur reçue.

Ces communications sont assurées par deux niveaux de réseaux inter-PEs. Un réseau de proximité lie chaque PE à ses voisins. Il est généralement utilisé pour les communications uniformes. Son architecture

conditionne le type de communications point à point réalisable sur la machine, selon par exemple le nombre de voisins connectés à un PE et les directions selon lesquelles ces voisins sont connectés. Un réseau général est chargé des autres communications. Sur certaines machines, ce réseau peut autoriser des communications point à point plus efficaces que les communications générales; on est assuré qu'il n'y aura pas de conflit d'accès d'un PE.

Il est donc important de distinguer les cas pouvant amener des communications plus performantes au sein des programmes. Cette distinction est nette si les différentes opérations de communications apparaissent en tant que telles au niveau du langage. Les langages dans lesquels les communications sont exprimées par des opérations de description des vecteurs proposent parfois une syntaxe particulière pour atteindre les éléments "proches" les uns des autres. Citons les opérateurs de décalage **`shift`** et **`rotate`** du langage ACTUS :

```
tabl [#] = tab2 [# shift 1]
```

produit à l'évidence une communication uniforme.

1-2.2.4 Un grand nombre de PEs

Les machines tableaux sont constituées d'un grand nombre de PEs physiques. Afin de cacher la singularité de ce nombre à l'utilisateur, la Connection Machine CM-1 introduit le concept de *processeurs virtuels*. Cette abstraction, implantée au niveau microcode, rend les programmes indépendants du nombre de processeurs physiques de la machine. Le *VP ratio* (VPR) est défini comme le quotient du nombre de processeurs virtuels par le nombre de processeurs physiques. L'utilisation de 256.000 PEs virtuels sur une machine 64.000 processeurs fixe VPR à 4. Trois aspects sont concernés par l'implantation de cette virtualité : la mémoire, les traitements et les communications.

VPR processeurs virtuels sont donc "alloués" sur un processeur physique. L'espace mémoire de chaque processeur physique est donc divisé en VPR espaces mémoire et ainsi réduit d'autant.

L'exécution d'une instruction sur un ensemble de processeurs virtuels est multiplexée dans le temps pour être réalisée sur chaque processeur virtuel alloué sur un processeur physique. La simplicité relative des processeurs élémentaires rend les commutations de contexte induites par ce mécanisme peu onéreuses.

L'implantation des communications inter-PEs prend, elle aussi, en compte l'allocation des processeurs virtuels sur les processeurs physiques. Les communications de voisinage partagent le réseau de voisinage entre les PEs virtuels. Les communications générales sont gérées par le matériel selon des adresses virtuelles prenant en compte le VP ratio.

Toutefois, cette facilité d'abstraction des caractéristiques de la machine au niveau matériel limite les optimisations réalisables par les compilateurs. Cette limitation est en grande partie due à l'implantation au niveau microcode qui fixe une fois pour toutes (lors de l'écriture du microcode) la réalisation des opérations sur les processeurs virtuels. L'approche de MasPar et du nouveau compilateur FORTRAN de la Connection Machine est à l'inverse de tenir compte de toutes les caractéristiques de l'architecture (par exemple les registres des PEs) dans la phase de compilation; se référer à [Chri91] et à la section 1-5.4.

1-2.2.5 La mémoire des PEs

Une mémoire est associée à chaque processeur élémentaire. La taille des objets alloués dans cette mémoire locale ne doit pas dépasser la taille de la mémoire; en effet cette mémoire n'est pas étendue par une

quelconque unité de disque ou autre. Cette caractéristique limite par exemple la taille maximale des objets manipulés ou le nombre d'objets alloués à un instant donné. Les conséquences d'optimisations pouvant limiter l'espace mémoire utilisé par un programme ne sont donc pas négligeables.

1–2.2.6 Bit de contexte

Le principe du SIMD est défini par l'exécution simultanée d'une même opération par tous les PEs. Ce principe est modéré par la possible inactivité de certains processeurs lors de la réalisation d'une instruction. Chaque PE possède un bit de contexte qui indique si le PE est actif ou non. En fait lors de l'exécution d'une instruction mémoire à mémoire, l'opération peut avoir lieu sur tous les PEs, mais seuls les PEs actifs rangent le résultat en mémoire.

Des structures de contrôles SIMD sont le reflet au niveau des langages de ce bit de contexte.

1–2.2.7 Notes bibliographiques

Chacun des ouvrages [HwBr84], [HoJe81], et [HoJe88] consacre un chapitre aux machines tableaux et détaille l'architecture de quelques-unes d'entre elles. Le livre de Hord [Hord90] contient une description précise de nombreuses machines tableaux. L'architecture de l'Illiac IV est présentée dans [BBK+68], le processeur de la machine dans [Davi69], et son logiciel dans [Kuck68]. La machine MPP est décrite dans [Batc80]. L'architecture de BLITZEN est décrite dans [BDHR90]. La section 3.4.2. de [HoJe88] est consacrée à l'ICL DAP. La Connection Machine fut conçue par Hillis alors qu'il rédigeait sa thèse. Celle-ci fut publiée [Hill85]. L'article de Trucker et Robertson présente l'architecture des CM-1 et CM-2 [TrRo88]. Les publications [KaHi89] et [DKV88] sont d'autres descriptions de l'architecture de ces machines, et présentent en particulier l'implantation des processeurs virtuels sur la CM-1. Les publications [TMC90b] et [TMC91a] contiennent respectivement une description plus générale de la Connection Machine 2, et de la Connection Machine 200. L'architecture de la machine MasPar MP-1 est présentée dans [Blan90], son implantation dans [Nick90], le logiciel et la programmation de la MP-1 sont développés dans [Chri90]. D'autres informations sur l'architecture sont disponibles dans [Chri91].

1–2.3 Comparaison entre les machines vectorielles pipelines et massivement parallèles

Nous avons présenté les caractéristiques principales des deux grandes familles de machines vectorielles, à savoir les machines vectorielles pipelines et les machines tableaux (dont la tendance actuelle est au parallélisme massif). Les spécificités de ces machines à prendre en compte par le logiciel (par le programmeur dans le langage ou par le compilateur) ont été exposées. Les différences entre les deux types de machines sont basées sur leur architecture fondamentale. Toutefois, ces deux familles de machines peuvent être regroupées sous le vocable de machines vectorielles non seulement parce qu'elles traitent toutes deux des vecteurs, mais aussi parce que de nombreuses similarités existent entre elles.La présente section compare quelques caractéristiques des deux familles et met en évidence des analogies.

1–2.3.1 Registres vectoriels sur machines massivement parallèles ?

L'architecture des machines vectorielles pipelines est basée sur un jeu de registres vectoriels dans lesquels les unités fonctionnelles prennent leur(s) opérande(s) et délivrent leur résultat. Le nombre de registres et la

taille de ces registres sont limités. D'un certain point de vue, une machine vectorielle tableau peut être considérée comme une machine à registres. La taille des registres est le nombre de processeurs élémentaires. La machine est composée d'autant de registres vectoriels que chaque PE contient de registres. Cette analogie autorise à transférer les techniques de compilation pour machines à registres vectoriels aux machines tableaux. Citons par exemple le strip-mining consistant à découper les vecteurs en tronçons de taille raisonnable pouvant être rangés dans un registre, et les opérations de fusions de boucles évitant de trop nombreux chargements des registres depuis la mémoire. Ces opérations seront détaillées dans la dernière partie de ce chapitre (1–5. La projection des langages à parallélisme de données).

1–2.3.2 Registre masque sur machines massivement parallèles ?

Les machines vectorielles disposent d'un registre masque de même longueur que les registres vectoriels (section 1–2.1.7). Ce registre contient une chaîne de bits indiquant si la composante correspondante des registres doit être traitée ou non. De manière analogue, l'activité de chaque processeur élémentaire d'une machine tableau est contrôlée par un bit de contexte (section 1–2.2.6). L'utilisation de ces bits de contexte est identique à celle faite des registres masques des machines vectorielles. Pour l'une comme pour l'autre, le taux de bits à 1 n'influence pas les performances des opérations réalisées; il est donc intéressant de garder un taux maximal de bit à 1.

1–2.3.3 Adressage et communication

Les machines vectorielles pipelines proposent des adressages indirects des vecteurs via les opérations de gather/scatter. Les machines tableaux proposent des primitives de communications inter-PEs. Ces deux opérations ne sont pas sans similitude. Elles mettent en œuvre toutes les deux la même fonctionnalité : l'adressage de vecteurs via une liste d'index. Elles sont toutes les deux pénalisantes par rapport à un adressage linéaire des vecteurs. Sur une machine vectorielle pipeline, une telle opération est réalisée par 1—le chargement d'une liste d'index dans un registre (*V1*), 2— le chargement d'un registre depuis la mémoire (ou l'écriture en mémoire (*@VECTEUR*) d'un registre (*V2*)) contrôlé par cette liste d'index :

```
VLOAD       V1,   @INDEX
VSTORE      V2,   @VECTEUR [ V1 ]
```

Sur une machine tableau, l'opération consiste en 1— le chargement d'un index (numéro de PE) pour chaque PE (par exemple dans *R1*), 2— l'émission ou la réception vers/de la valeur située à l'adresse du vecteur dans ce PE :

```
LOAD        R1,   @INDEX
SEND        R2,   @VECTEUR, R1
```

La similitude entre les deux opérations est flagrante; toutefois le surcoût engendré par une communication (machines tableau) est plus élevé que celui engendré par une adressage indirect (machines pipelines). De même, il existe une similitude entre les communications locales sur une machine massivement parallèle et les accès à des tranches de vecteurs sur les machines pipelines vectorielles.

1–2.3.4 Conclusion sur cette comparaison

Les similarités entre les deux types de machines vectorielles ont été présentées dans cette section. Ces similitudes font que d'une part il est possible de concevoir des langages de programmation communs aux

deux types de machines, et que d'autre part certaines techniques employées par les compilateurs pour un type de machines sont applicables avec un minimum d'adaptations à l'autre type de machine.

1–3 Les langages vectoriels

Cette section présente les approches habituelles et/ou envisageables pour la programmation vectorielle. Nous définissons les langages vectoriels et présentons une classification des langages vectoriels due à Hockney et Jesshope. Les différentes structures proposées pour la programmation vectorielle seront ensuite discutées.

Il n'est pas simple de définir le terme *langage vectoriel*. Considérons la définition : un langage vectoriel est un langage dans lequel il existe une structure exprimant un certain parallélisme de données. Cette définition exclut l'approche, très utilisée pourtant, consistant en l'association d'un langage scalaire et d'un vectoriseur automatique. Nous proposons donc de considérer non plus les langages vectoriels mais les *systèmes de développement de programmes vectoriels* dont la définition est : un système de développement de programmes vectoriels est un ensemble d'outils permettant à un programmeur d'exprimer un algorithme et de le projeter sur une machine vectorielle.

Hockney et Jesshope considèrent un système de développement de programmes vectoriels/parallèles depuis le problème jusqu'à son exécution sur une machine cible. Quatres phases sont distinguées; à chacune est associé un degré de parallélisme défini comme le nombre d'opérations indépendantes réalisables simultanément :

1) Le choix d'un algorithme résolvant le problème (degré de parallélisme P),

2) L'expression de cet algorithme dans un langage de haut niveau (degré de parallélisme L),

3) La compilation de ce langage en langage objet (degré de parallélisme O),

4) L'exécution de ce code objet sur la machine cible (degré de parallélisme M).

Dans la situation idéale, le degré de parallélisme ne doit pas croître lors des phases successives. C'est le *Principe de conservation du parallélisme* :

$$P \geq L \geq O \geq M.$$

Il semble en effet idéal que le programmeur exprime son algorithme en terme de constructions syntaxiques explicites reflétant effectivement le parallélisme des données de son algorithme, sans être contraint par la nature ou même les spécificités de la machine cible. Cependant d'autres approches existent :

- Un système de développement de programmes vectoriels tel que $P \geq L \leq O \geq M$. L'algorithme est exprimé dans un langage scalaire. La phase 3 de compilation de ce langage en langage machine doit découvrir le parallélisme non exprimé. Cette phase est réalisée par les vectoriseurs et paralléliseurs automatiques.

- Un système de développement de programmes vectoriels tel que $P \geq L = O = M$. Les constructions offertes au niveau du langage expriment le parallélisme de la machine. Ces langages sont dédiés à la machine cible.

Ces différentes approches sont discutées dans les sections suivantes.

1-3.1 Langages scalaires et vectorisation automatique

Une approche classique de la programmation vectorielle utilise un langage scalaire associé à un vectoriseur. La production de code vectoriel depuis le langage scalaire est assurée en deux étapes. Le programmeur exprime son algorithme dans un langage scalaire. Une première phase du compilateur retrouve dans le source scalaire les séquences de code pouvant produire du code vectoriel. La seconde est commune à tous les types de langage; elle assure la génération de code vectoriel cible.

Cette approche est principalement motivée par la portabilité des sources et par la compatibilité avec les applications existantes et développées dans des langages scalaires. Ces deux arguments sont de poids à première vue. Nous en discutons dans les sous sections suivantes. Cette section n'a pas pour but de déterminer s'il faut ou non utiliser les vectoriseurs; ils sont de fait très utilisés. Elle n'a pas non plus pour but de déterminer si les travaux sur les vectoriseurs sont utiles; ils le sont de fait. Nous nous attacherons plutôt à déterminer l'adéquation de la "démarche vectoriseur" à la programmation vectorielle.

1-3.1.1 Adéquation des vectoriseurs à la programmation à parallélisme de données

Considérons le programmeur qui écrit du code scalaire qui sera ensuite vectorisé. Il lui faut extraire le parallélisme de son algorithme. Ensuite il lui faut exprimer ce parallélisme dans un langage scalaire en respectant certaines contraintes induites par le vectoriseur. Le vectoriseur aura ensuite pour charge de retrouver le parallélisme dans le code vectoriel. Cette situation n'est pas totalement satisfaisante. La solution qui semble être idéale serait que le programmeur puisse exprimer clairement le parallélisme dans un codage adéquat et facilement reconnaissable par le compilateur.

La portabilité sur machines scalaires et vectorielles d'une même application est, au sens strict, impossible si l'algorithme est exprimé à un niveau trop bas (au niveau scalaire pour des opérations vectorielles par exemple). En effet il existe des constructions qui sont spécifiques au traitement vectoriel et dont la réalisation n'a aucun sens sur une machine scalaire.

Les instructions de compression/extension sont de telles constructions. Les composantes d'un tableau sélectionnées par une certaine condition sont regroupées dans un tableau annexe. Une fois ce tableau annexe traité, les composantes sont redistribuées dans le tableau de départ. Sur une machine vectorielle, cette démarche permet de réaliser de manière vectorielle le traitement sur le vecteur annexe. Sur une machine scalaire, elle n'a pas raison d'être et entraîne même une perte d'efficacité, le traitement pouvant être réalisé au fur et à mesure de la sélection.

Une solution semble être que la programmation se fasse à un niveau assez élevé permettant de prendre en compte toute la séquence d'opérations de telles constructions dans le langage. La génération des compressions/extensions ne se ferait ensuite que sur une machine sur laquelle cette approche serait justifiée.

Un langage scalaire est de trop bas niveau pour exprimer des algorithmes vectoriels. Par exemple, une opération de somme de deux tableaux exprimée dans un langage scalaire impose un ordre de traitement des

composantes alors qu'aucun ordre n'est nécessaire et que les opérations sur les composantes peuvent être réalisées dans un ordre quelconque.

1–3.1.2 Portabilité des applications existantes

Grâce à l'emploi des vectoriseurs, les milliers de lignes des applications existantes n'ont pas à être réécrites pour que les applications soient portées sur les machines vectorielles. Cet a priori tombe lorsqu'on analyse le gain obtenu en vectorisant simplement une application existante, i.e. en passant les programmes sources existants au travers d'un vectoriseur. Ce gain est en général minime par rapport à celui que l'on obtient en "vectorisant" l'application à la main. Cette "vectorisation à la main" est double. Elle peut être locale et consister à rendre vectorisable des boucles existantes mais écrites de manière non vectorisable. Elle recouvre également une restructuration plus en profondeur de l'application pour que les structures de données soient adaptées à la vectorisation automatique. (Rappelons que la plupart des vectoriseurs ne génèrent pas de code vectoriel en dehors des boucles et des manipulations de tableaux.) Le nombre de travaux de "vectorisation à la main" d'applications existantes est la preuve que la portabilité sur des machines vectorielles des applications existantes n'est pas une tâche triviale.

1–3.1.3 L'utilisation des vectoriseurs

L'argumentation précédente ne vise pas à dénaturer les vectoriseurs en tant qu'outils de programmation vectorielle mais à énoncer clairement que les voies à suivre pour le développement de nouveaux outils de programmation ne passent pas nécessairement par les vectoriseurs. Cependant il est clair que les vectoriseurs sont largement utilisés et que les nombreux travaux visant à améliorer leurs performances sont des plus utiles pour que la prise en compte immédiate d'applications existantes écrites en langage scalaire soit des plus efficaces.

Les vectoriseurs permettent le passage en douceur d'un langage scalaire à un langage vectoriel. Une application existante écrite dans un langage scalaire est portée sur une machine vectorielle. Le vectoriseur autorise cette portabilité immédiate. La "réécriture vectorielle" progressive de l'application est ensuite possible.

1–3.2 Langages vectoriels dédiés à une machine cible

Une autre approche classique de la programmation vectorielle utilise un langage dédié à une machine cible. Le développement de tels langages peut trouver une explication dans les rapports entre les évolutions du matériel et du logiciel de base. Les nouvelles générations de machines présentent de nouvelles caractéristiques architecturales. Les premiers langages (de haut niveau) développés sur ces architectures sont des extensions de langages existants prenant en compte les nouvelles spécificités. Il n'est en effet pas toujours facile de compiler efficacement les langages existants pour tirer parti de la nouvelle architecture (par exemple un nouvel adressage vectoriel dans la mémoire ou la présence de mémoires hiérarchisées)

Ces langages sont, par définition, non portables. On peut les comparer à des langages machines de plus ou moins haut niveau.

Des spécificités de l'architecture de la machine sont apparentes dans le langage. Citons par exemple l'extension du langage C pour la machine Cray-2 [Giss86]. Le mot-clé **cache** spécifie qu'une donnée doit être allouée dans la mémoire cache.

Le langage SL/1 développé sur CDC Star-100 reflète délibérément les caractéristiques de la machine [BaKn75]. Il propose des structures de données directement manipulables par le STAR-100 (réels, demi réels, et chaînes de bits). Les vecteurs sont mono-dimensionnels sur le STAR-100, les vecteurs SL/1 aussi. Les adressages de vecteurs sur le STAR-100 se font par un couple (adresse de base, longueur); les vecteurs SL/1 sont représentés par de tels couples. Les lectures/écritures mémoire peuvent être contrôlées par une chaîne de bits sur le STAR-100; l'association d'un vecteur et d'une chaîne de bits est manipulable directement en SL/1.

L'architecture de la machine limite les constructions autorisées au niveau du langage. Citons les langages CFD [Stev75] et Glypnir [LLBR75] développés tous deux sur Illiac IV et dont les constructions sont limitées à une taille de 64, le nombre de processeurs de l'Illiac IV !

1-3.3 Langages vectoriels explicites

Les langages à parallélisme explicite sont qualifiés par Hockney et Jesshope comme les outils idéaux pour la programmation parallèle. Ces langages proposent au programmeur des structures lui laissant exprimer le parallélisme de son algorithme. Cette catégorie recouvre les langages présentés dans la section précédente. Mais les constructions des langages vectoriels (tels que nous les qualifions maintenant) se veulent indépendantes, sinon d'une architecture, du moins d'une machine spécifique.

Cette section présente rapidement les constructions générales qui ont été développées dans ces langages vectoriels. La section suivante présente quelques-uns des langages les plus significatifs qui ont été proposés depuis l'apparition des machines à parallélisme de données. On pourra se reporter au petit historique des extensions vectorielles du langage FORTRAN qui a été établi par Guzzi & al. [GPHL90]. On se reportera sinon aux différents articles et manuels de références des langages.

1-3.3.1 Les structures de données

La structure de données de base des langages vectoriels est le tableau. Un tableau est un ensemble ordonné d'éléments. Cette structure est à la rencontre des besoins des problèmes traités et des machines sur lesquelles ils sont implantés. Les problèmes traités sont relatifs au calcul scientifique et, de ce fait, font une utilisation répétée de tableaux. La structure dont l'implantation sur les machines vectorielles est la plus efficace est sans aucun doute le tableau. Différentes informations sont attachées aux tableaux. Bien qu'il n'y ait pas de terminologie commune entre les différents langages, nous listons ici ces informations et leur attribuons un terme généralement utilisé.

- La ***longueur*** Aussi dénommée taille du tableau. C'est le nombre d'éléments composant le tableau. Elle peut être fixée à la compilation (le plus généralement) ou non (les tableaux sont alors dynamiques).

- Le ***rang*** et la ***dimension*** Le rang (*rank*) d'un vecteur est le nombre de dimensions de ce vecteur. Par exemple, les tableaux de rang 2 sont souvent nommés matrices. Le rang d'un vecteur est généralement fixé à la compilation.

- Les *bornes* et la *forme* Les bornes précisent, pour une dimension donnée, la variation des indices des éléments du vecteur. Elles sont formées d'un couple *borne inférieure*, *borne supérieure*. La valeur par défaut de la borne inférieure est traditionnellement 1. La forme d'un vecteur est la réunion, pour toutes les dimensions, des informations de bornes.

1–3.3.2 Expressions et opérateurs

La sémantique des opérateurs habituels des langages scalaires tels l'addition, la multiplication, etc. est généralement étendue aux tableaux. Sémantiquement, l'opération est réalisée sur les éléments correspondants des vecteurs impliqués dans une opération. Les valeurs scalaires utilisées dans une opération avec un vecteur sont fonctionnellement équivalentes à un vecteur de taille convenable dont tous les éléments ont la valeur du scalaire.

Notons tout de suite qu'une instruction d'affectation vectorielle n'est pas équivalente à la boucle scalaire correspondante. La sémantique de l'affectation vectorielle est telle que toute la partie droite est évaluée avant le rangement des valeurs dans la partie gauche. Une dépendance entre les parties droite et gauche nécessite donc le passage par une zone temporaire.

1–3.3.3 Notion de conformance

Un point important du traitement de vecteurs est de connaître la longueur (nombre d'éléments) sur laquelle traiter ces vecteurs, notamment lorsque tous les vecteurs impliqués dans une opération n'ont pas la même longueur. Notons que bien souvent les langages n'autorisent pas les opérations vectorielles à être réalisées sur des opérandes de longueurs différentes. De même, la majorité des langages ne considère une opération entre deux tableaux valide que si ces tableaux ont le même rang et la même taille dans chaque dimension. Les opérations ne sont alors valides que sur des tableaux *conformes*.

1–3.3.4 Opérations de descriptions

Une opération de description est une opération réalisant un accès (en lecture ou en écriture) à un sous-tableau. Dans les langages scalaires, ces opérations résultent en une référence sur un élément d'un tableau; la description est alors réalisée par une valeur scalaire entière : l'indice de l'élément dans le tableau. Les différentes descriptions résultant en une liste de valeurs (un sous-tableau) sont listées ici. Hext compare les opérations de description de plusieurs langages dans [Hext75].

Description par valeurs booléennes

Un des premiers langages à proposer la description par des valeurs booléennes est le Burroughs FORTRAN de l'Illiac IV (1971). Il utilise des *control vectors* (vecteurs de contrôle) comme descripteurs de tableaux. Un élément d'un vecteur de contrôle prend une des deux valeurs `.true.` et `.false.`. Les éléments du tableau sont associés à l'élément correspondant du vecteur de contrôle. Une valeur `.true.` pour un élément du vecteur de contrôle indique que l'élément correspondant du tableau est à accéder pour former le sous-tableau. Un astérisque `*` dénote un vecteur de contrôle de longueur quelconque dont tous les éléments sont `.true.`. Par exemple [GPHL90], le code

```
      REAL A (100), B (100), C (100)
      DO 10 i = 1, 100, 2
            M (i) = .true.
            M (i+1) = .false.
10    CONTINUE
      A (*) = B (*) + A (*)
      C (M(*)) = B (M(*)) + A (M(*))
```

ajoute l'élément correspondant de *B* à chaque élément de *A*. Mais dans la dernière instruction, du fait de l'utilisation du vecteur de contrôle *M*, seuls les éléments impairs de *A* et de *B* sont ajoutés et rangés dans les éléments impairs de *C*.

Cette description est maintenant réalisée par des vecteurs ou chaînes de bits, chaque élément ayant la valeur 1 (vrai) ou zéro (faux). Ces différents termes recouvrent la même fonctionnalité. Le nombre d'éléments sélectionnés par la description est égal au nombre d'éléments du vecteur de contrôle de valeur vraie, et ne peut donc généralement pas être déterminée à la compilation.

Description par triplet

Le BSP FORTRAN, une extension de FORTRAN proposée pour la machine BSP, introduit un autre type de description. Il autorise la description d'un vecteur par un triplet de valeurs entières scalaires et ce dans chaque dimension d'un tableau. Par exemple, dans le source [KPSW82]

```
      REAL A (M, N), P (K)
      P (*) = 0
      DO 10 j = 1, N
            P (*) = P (*) + A (1:M-1:2, j) * A (2:M:2, j)
10    CONTINUE
```

la notation *A (2:M:2, j)* désigne les éléments de rang d'indice pair de 2 à *M* de la colonne *j*. Cette notation est maintenant reprise dans la plupart des langages vectoriels. Les trois valeurs sont dénommées *borne inférieure* : *borne supérieure* : *pas* ou *lower* : *upper* : *step*. Le pas est souvent optionnel; sa valeur par défaut est alors 1. Les bornes inférieure et supérieure (lower et upper) sont parfois optionnelles; elles prennent alors les valeurs des bornes inférieure et supérieure de la dimension décrite.

Description par liste d'entiers

VECTRAN introduit la description par un vecteur (tableau à une dimension) d'entiers [PaWi78]. Cette description est une extension de la description habituelle par une valeur entière scalaire. Supposons que *Z* soit un vecteur de taille 7, et *U* et *V* deux vecteurs de taille 3 et 4 respectivement tels que

$U = 1\ 3\ 2$
$V = 2\ 1\ 1\ 3$

alors, *Z (U)* est formé des éléments de *Z* dans l'ordre :

Z (1) Z (3) Z (2)

et *Z (V)* est formé des éléments :

Z (2) Z (1) Z (1) Z (3)

Il est à remarquer qu'un même élément d'un tableau peut être référencé plusieurs fois dans une description par un vecteur d'entiers. Ce fait ne pose pas de problème en lecture. Par contre, en écriture le résultat dépend de l'ordre de traitement des éléments. Par exemple, supposons une valeur initiale nulle pour les 7 éléments de Z, une opération d'affectation de Z décrit par le vecteur d'entier précédent *V* :

Z (*V*) = 1 2 3 4

produit le résultat suivant pour un traitement des éléments dans l'ordre croissant des indices

Z = 3 1 4 0 0 0 0

et un résultat différent si les éléments sont traités dans l'ordre décroissant des indices :

Z = 2 1 4 0 0 0 0

De ce fait, bien souvent, la sémantique des langages ne garantit pas le résultat d'une telle opération. FORTRAN 90 interdit même les opérations de description par un vecteur d'entiers comportant des doublets en partie gauche d'une affectation. Il n'est cependant pas envisageable que les comnpilateurs vérifient le respect de cette règle.

Ces opérations de descriptions sous souvent dénommées *gather/scatter*. Un gather est une description par une liste d'index *I* en lecture :

```
Pour tout i ∈ [ Borne_inf .. Borne_sup ]
      T (i) = TAB (I (i))
```

un scatter est une description par liste d'index en écriture :

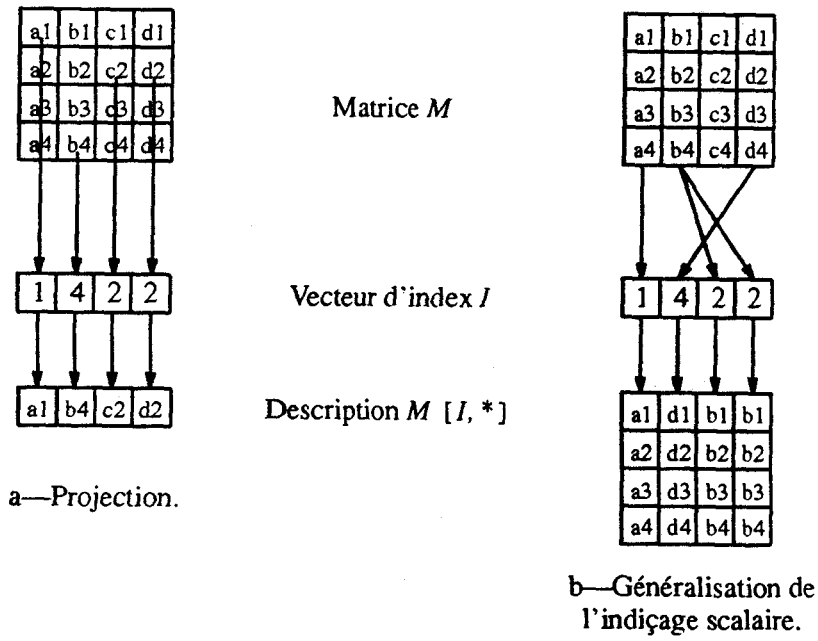
```
Pour tout i ∈ [ Borne_inf .. Borne_sup ]
      TAB (I (i)) = T (i)
```

Le sous-tableau résultant de la description contient autant d'éléments que le vecteur de description. Notons que si une valeur d'une liste d'index n'est pas comprise entre la borne inférieure et la borne supérieure de la dimension décrite, un élément est sélectionné soit en dehors de cette dimension soit même en dehors du vecteur. Pour des raisons évidentes de performances, quand les valeurs du vecteur de description ne peuvent être connues à la compilation, la majorité des langages ne détecte pas d'éventuels dépassements.

Deux interprétations de la description des tableaux multi–dimensionnels par une liste d'index

Il existe au moins deux interprétations de la description d'un tableau multi-dimensionnel par une liste d'index [HoJe88, pp. 389].

1 La première interprétation est une projection sur une (ou plusieurs dimensions d'un vecteur) (figure 1–5.a). Une description réduit donc le rank (nombre de dimensions) du vecteur. Le bound (taille de la dimension) du vecteur d'index doit correspondre à celui de la dimension décrite. Chaque valeur du vecteur d'index est l'indice dans la dimension de l'élément qui sera projeté. Cette interprétation est, par exemple, celle du langage ACTUS.



a—Projection.

b—Généralisation de l'indiçage scalaire.

Figure 1–5. Deux interprétations de la description par une liste d'entiers.

2 La seconde interprétation est une généralisation de l'indiçage par une valeur scalaire entière (figure 1–5.b). Le rang du vecteur résultat est identique à celui du vecteur original. La réduction de rang est obtenue par une description par un scalaire. Les informations bornes inférieure et supérieure du résultat sont celles du vecteur d'index; elles peuvent être différentes de celles de la dimension décrite du vecteur initial. La valeur de chaque élément du vecteur d'indices sélectionne la colonne (ou le plan...) du vecteur initial qui formera celle du vecteur résultant. Cette interprétation est celle de la plupart des langages vectoriels : VECTRAN, VECTOR C, FORTRAN 90, etc...

1–3.3.5 Une structure de contrôle vectorielle : la construction WHERE

Le **WHERE** est une construction courante des langages vectoriels. Elle autorise la réalisation d'une opération sur une partie des éléments des vecteurs d'une instruction. Cette construction fut introduite pour coder en vectoriel les opérations telles que

```
      DO 10 i = 1, 100
            IF (A (i) .GT. B (i))
                  A (i) = A (i) + B (i)
10    CONTINUE
```

On écrit alors

```
WHERE (A .GT. B) A = A + B
```

Cette même instruction peut être considérée comme une mise en facteur de la description par une liste de valeurs booléennes. L'opération est, selon les langages, équivalente à

```
A (A. GT. B) = A (A .GT. B) + B (A .GT. B)
```

ou à

```
A (A. GT. B) = (A + B) (A .GT. B)
```

Cette mise en facteur peut être réalisée sur une suite d'instructions, par exemple en FORTRAN 90 :

```
WHERE (A .GT. B)
      A = A + B
      C = A - B
END WHERE
```

L'alternative **ELSE** d'un **IF** imbriqué dans une boucle de traitements de vecteurs est remplacée par une alternative **ELSE WHERE** (ou **OTHERWISE** selon les langages) :

```
      DO 10 i = 1, 100
            IF (A (i) .GT. B (i))
                  A (i) = A (i) + B (i)
                  C (i) = A (i) - B (i)
            ELSE
                  A (i) = A (i) - B (i)
            END IF
10    CONTINUE
```

est équivalent à

```
WHERE (A .GT. B)
      A = A + B
      C = A - B
ELSE WHERE
      A = A - B
END WHERE
```

Cette construction n'indique pas que le bloc d'instructions **WHERE** et le bloc **ELSEWHERE** peuvent être exécutées indépendamment. Mais, la sémantique du **WHERE** indique généralement que les instructions du bloc **WHERE** sont effectuées avant celles du bloc **ELSEWHERE**. Les instructions apparaissant à l'intérieur d'une structure **WHERE** sont bien souvent limitées à des affectations de vecteurs et à des appels de fonctions prédéfinies retournant des vecteurs. D'autres subtilités concernent la présence ou non d'un test avant l'exécution d'un bloc **WHERE** ou **ELSEWHERE**. MPL assure par exemple qu'un bloc **WHERE** (resp. **ELSEWHERE**) n'est exécuté que si au moins un bit du vecteur de contrôle est vrai (resp. faux); ce n'est pas le cas en POMPC. Les sémantiques de ces constructions sont étudiées par Bougé et Levaire [BoLe91].

1–4 Les propositions de langages pour la programmation à parallélisme de données

Cette section présente les principaux langages qui ont été proposés depuis l'apparition des premières machines vectorielles jusqu'à FORTRAN 90, la plus récente norme du langage FORTRAN, et FORTRAN D, une proposition de langage visant toutes les machines à parallélisme de données. Nous nous efforcerons de présenter les points originaux et limitations de chacun des langages, les caractéristiques communes et/ou reprises de langages plus anciens étant parfois passées sous silence.

1–4.1 Un pionnier des langages vectoriels : VECTRAN

VECTRAN est une extension vectorielle de FORTRAN qui fut développée par Paul et Wilson chez IBM à partir des années 1975. Trois documents peuvent être référencés sur VECTRAN; ce sont : le manuel de référence du langage VECTRAN [PaWi75], l'article introductif à VECTRAN [PaWi78], et une comparaison de VECTRAN avec le projet d'une nouvelle norme FORTRAN de l'époque [Paul84].

Les caractéristiques essentielles et les originalités de VECTRAN se résument ainsi : introduction de la manipulation de tableaux en tant que tels en FORTRAN, dynamicité des *range* (informations relatives aux bornes et à la forme d'un tableau), description des tableaux par des vecteurs (tableaux mono-dimensionnels dans la terminologie VECTRAN), les *virtual arrays* (tableau auquel est associée une description par triplet) et l'instruction **IDENTIFY**, des affectations vectorielles **WHEN**, **AT**, **PACK** et **UNPACK**, et des extensions des mécanismes d'appel et de retour de fonctions.

Un langage vectoriel explicite

VECTRAN est un langage vectoriel explicite. VECTRAN propose non seulement la manipulation des tableaux en tant que tels dans les expressions du langage et l'extension de la sémantique des opérateurs

scalaires aux tableaux, mais Paul et Wilson estiment qu'en tant que langage vectoriel, le langage doit aussi fournir :

1) les moyens de spécifier un sous-tableau rectangulaire et diverses sections d'un tableau,

2) la possibilité de manipuler les tableaux et les sous-tableaux comme des entités aussi bien en partie droite que gauche d'une affectation,

3) la possibilité de définir et d'utiliser des fonctions dont le résultat est un tableau et de passer des expressions tableaux comme paramètres à des fonctions,

4) un ensemble d'opérateurs incluant des opérateurs de traitement de matrices et de réductions, et

5) des facilités pour supporter la manipulation de vecteurs dont les éléments sont référencés soit par une liste de valeurs booléennes, soit par une liste d'index entiers.

Les auteurs du langage se sont également attachés à ce que la compilation des extensions proposées puisse produire un code performant. De plus, ils ont eu souci d'appliquer la règle de "contention". Cette règle stipule que la non-utilisation d'une nouvelle construction offerte par le langage ne doit pas affecter les performances d'un programme.

Le range et l'instruction RANGE

VECTRAN introduit la notion de *range* suivante : le range d'un tableau est une liste de N éléments, où N est le rang du tableau. Chaque élément de cette liste est soit un littéral entier positif, soit le nom d'une variable scalaire. Un tableau est de range fixe si tous les éléments de cette liste sont des littéraux entiers; cette alternative est généralement la seule proposée par les autres langages. Un range caractérisé par une variable lie l'*active range* du tableau à cette variable. Cet active range peut donc varier lors de l'exécution, si la valeur de la variable change.

L'instruction **RANGE** permet de déclarer un tableau et d'en spécifier le range. Exemple :

```
RANGE /N,M/ A (10 , 10), B (15 , 25)
...
N = 5
M = N + 2
...
A = 2.5 * A + B
...
```

A la suite de l'instruction **RANGE**, les tableaux A et B sont déclarés et alloués. A de taille 10*10 et B de taille 15*25. Les scalaires N et M sont déclarés et initialisés (N à 10, et M à 10 : le range du premier tableau de la déclaration). Les tableaux A et B ont un range de $N*M$. Les affectations de N et M changent les ranges actifs de A et B en 5*7. Donc dans l'instruction A = ..., A et B référencent des matrices 5*7, cette instruction est donc fonctionnellement équivalente à

```
      DO 10 i = 1, N
      DO 10 j = 1, M
            A (i, j) = 2.5 * A (i, j) + B (i, j)
10    CONTINUE
```

La *size* (taille) est le nombre d'éléments d'un tableau. Elle est connue par la combinaison (multiplication) des tailles des éléments du range du tableau. Elle doit être connue à la compilation pour tous les tableaux dits primaires (par opposition aux tableaux secondaires que nous détaillons ensuite); l'allocation de ces tableaux étant donc statique.

Description d'un tableau

Un tableau peut être décrit par une liste de *N subscripts* (descripteurs), *N* est le rang du tableau. Comme en FORTRAN, un descripteur peut être une valeur scalaire. De plus, un tableau peut être décrit par une expression entière correspondant à :

- Un *vector valuated*. C'est un vecteur d'index, cela permet de faire des opérations de gather/scatter.
- Une *section selector*. Un astérisque * indique que l'active range est entièrement sélectionné. De plus il est possible de "shifter" une section; la syntaxe est

  <signe> * <expression>

  la valeur de l'expression est la taille du shift; un signe + indique que les éléments sont sélectionnés dans l'ordre des indices croissants, un signe - qu'ils le sont dans l'ordre décroissant. Il est donc possible d'accéder des valeurs en dehors de l'*active-range*, par exemple

  ```
  M (*+1)
  ```

  Les figures 1–6 et 1–7 illustrent des exemples d'utilisation de ces descriptions.
- Un *replication selector*. Cette opération augmente le rank d'un tableau en le dupliquant suivant une dimension donnée. Par exemple, *X* étant un vecteur de dimension 1 de taille 4, et *N* un scalaire de valeur 3,

  ```
  X (/N/, *)          et          X (*, /N/)
  ```

  représentent respectivement les matrices,

  ```
  X (1) X (2) X (3) X (4)                X (1) X (1) X (1)
  X (1) X (2) X (3) X (4)     et         X (2) X (2) X (2)
  X (1) X (2) X (3) X (4)                X (3) X (3) X (3)
                                         X (4) X (4) X (4)
  ```

  La construction est intéressante; on peut regretter la syntaxe laissant croire que le vecteur *X* est à un vecteur à deux dimensions.

Tableaux secondaires – Instruction `IDENTIFY`

Dans la terminologie VECTRAN, *secondary array* (tableau secondaire) et *virtual array* (tableau virtuel) sont synonymes; ils référencent des tableaux pour lesquels aucune allocation n'est faite, mais qui partagent l'espace mémoire alloué à un autre tableau.

Les tableaux virtuels permettent de manipuler directement (via un nom) des éléments de tableaux qui ne sont pas nécessairement contigus, mais régulièrement espacés dans la mémoire. L'instruction `IDENTIFY`

```
RANGE /N,M/ A (7, 15)
M = 7
N = M/2 + 1
```

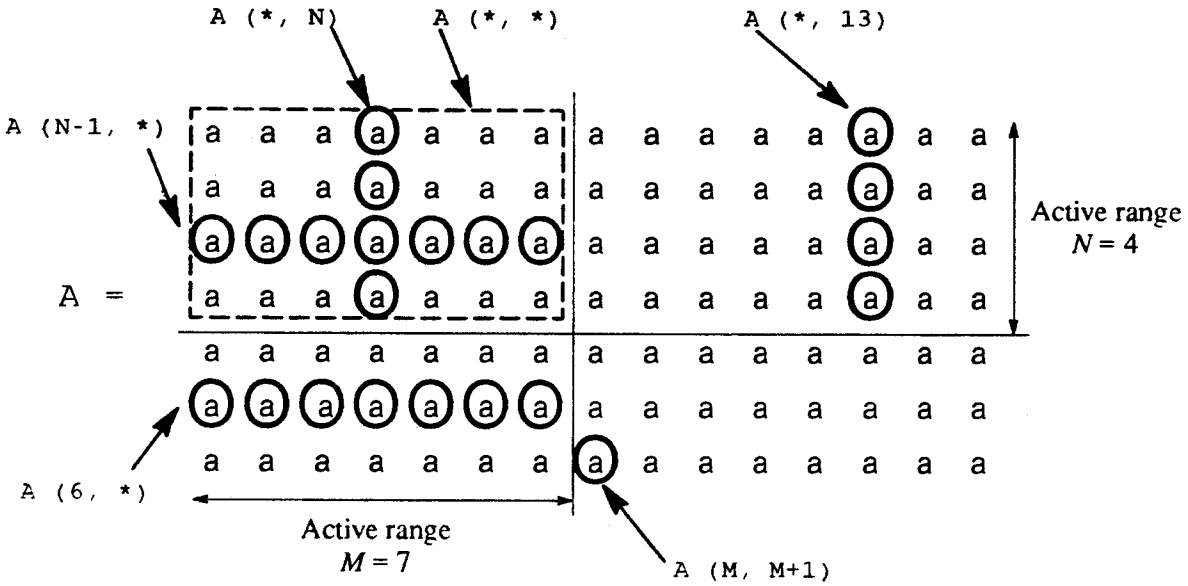


Figure 1–6. Exemple de sections d'un tableau en VECTRAN.

```
RANGE /N,M/ A (7, 15)
M = 7
N = M/2 + 1
```

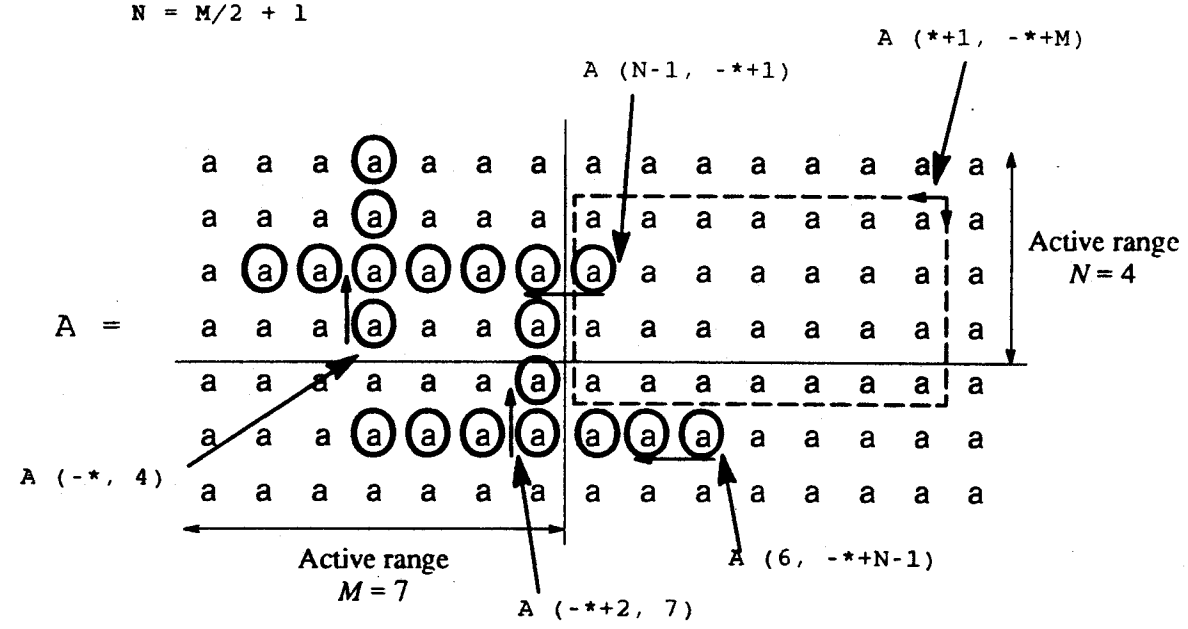


Figure 1–7. Exemple de sections avec shift et direction inverse en VECTRAN.

est une instruction qui, à l'exécution, associe un tableau virtuel à un tableau (le *real array*, tableau réel). Il indique quels sont les éléments du tableau virtuel parmi ceux du tableau réel. Une instruction **IDENTIFY** ne produit pas de nouvelle allocation mais un partage de certains éléments du tableau réel par le tableau virtuel. Exemple :

```
DIMENSION X (7, 7)
          .
          .
IDENTIFY /K/ XU (I) = X (I, I+1)
IDENTIFY /K/ XL (I) = X (I+1, I)
IDENTIFY /L/ Y (J)  = X (8-J, 2*J-1)
          .
          .
```

La première instruction **IDENTIFY** associe le nom *XU* à un tableau virtuel de range *K*; le *i*ème élément de *XU* est l'élément *X (i, i+1)*. Le tableau *XU* est donc formé des éléments de la diagonale supérieure du tableau *X* (figure 1–8).

Les indices des éléments dans le tableau réel correspondant à des éléments du tableau virtuel doivent s'exprimer à l'aide d'une combinaison linéaire des indices dans le tableau virtuel. Cette construction limite donc l'association d'un tableau virtuel à une tranche d'un vecteur réel ou à des éléments régulièrement espacés dans un tableau réel.

Une fois déclaré et associé par une instruction **IDENTIFY**, un tableau virtuel peut être manipulé en VECTRAN comme un tableau réel. Il est même possible d'associer un nouveau tableau virtuel à ce tableau. La combinaison de deux triplets *borne inférieure* : *borne supérieure* : *pas* produit un triplet; un tableau virtuel est donc constitué d'éléments régulièrement espacés dans un tableau réel (éventuellement contigus : pas de 1).

Expressions et opérateurs

La sémantique des opérateurs habituels de FORTRAN est étendue aux tableaux (y compris les tableaux virtuels). Une expression VECTRAN est valide si les opérandes sont conformes (*conformable*). Deux opérandes sont conformes s'ils ont le même range. Les scalaires sont conformes avec tous les tableaux. Notons aussi les instructions d'affectation **PACK** et **UNPACK** qui autorisent l'affectation d'une expression tableau à un tableau non conforme.

La sémantique de l'affectation d'un vecteur est définie de manière à prendre en compte toute dépendance pouvant intervenir entre la partie droite et la partie gauche d'une affectation. La partie droite est évaluée et rangée dans une zone temporaire. Ensuite les valeurs de cette zone temporaire sont effectivement rangées dans la partie gauche.

VECTRAN propose quelques autres opérateurs. L'opérateur de swap (échange) est noté ==. Notons cependant que la génération d'un swap ne peut se passer du passage par une zone temporaire et se suffire d'un registre vectoriel, une dépendance entre la partie droite et gauche du swap pouvant intervenir. Des opérateurs de manipulations de matrices (transposée, etc...) sont fournis. Les opérateurs unaires de réduction sont généralement notés *op/*. Par exemple, la somme des éléments d'un tableau *A* est obtenue par

```
DIMENSION X (7, 7)
            .
            .
IDENTIFY /K/ XU (I) = X (I, I+1)
IDENTIFY /K/ XL (I) = X (I+1, I)
IDENTIFY /L/ Y (J)  = X (8-J, 2*J-1)
            .
            .
```

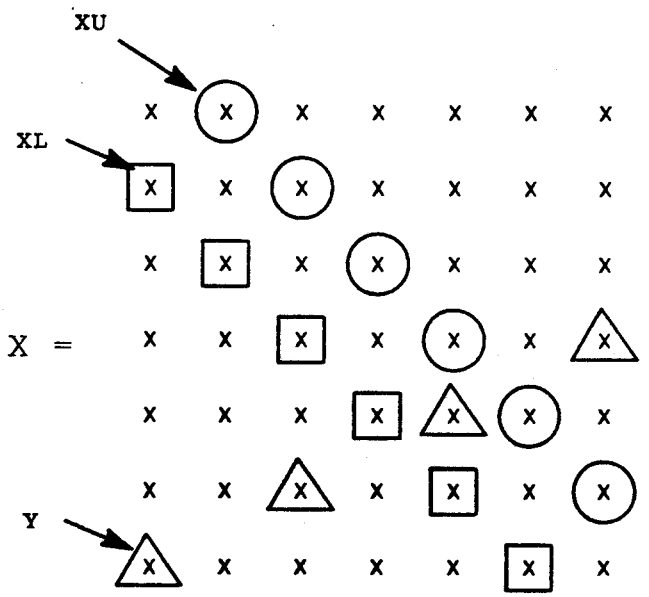


Figure 1–8. L'instruction **IDENTIFY**.

```
+/ A
```

D'autres opérations/fonctions prédéfinies retournent le minimum, le maximum d'un tableau, etc...

Affectations contrôlées — Instructions WHEN et AT

VECTRAN propose deux *affectation-like* opérateurs, le **WHEN** et le **AT**. La syntaxe du **WHEN** est :

WHEN (*logexpr*) *A* = *expr1* [, **OR** *B* = *expr2*]

logexpr, *A* et *B* sont des tableaux conformes. *logexpr* résulte en un tableau de *logical* (valeurs booléennes). Les expressions *logexpr*, *expr1*, et *expr2* sont évaluées. Puis selon la valeur vraie ou fausse de *logexpr (i)*, une des affectations *A (i)* = *expr1 (i)* ou *B (i)* = *expr2 (i)* est effectuée. La syntaxe du **AT** est identique à celle du **WHEN** :

AT (*logexpr*) *A* = *expr1* [, **OR** *B* = *expr2*]

La sémantique est différente. L'expression *expr1* (resp. *expr2*) n'est évaluée que pour les indices *i* tels que *logexpr (i)* est vrai (resp. faux).

Fonctions

Les fonctions VECTRAN (**FUNCTION** et **SUBPROGRAM**) sont des extensions des fonctions FORTRAN. Il est donc possible d'appeler des fonctions FORTRAN depuis VECTRAN, et inversement (à quelques limitations près, par exemple les tableaux paramètres). Comme en FORTRAN, tous les passages de paramètre se font par adresse; les fonctions appelées et appelantes se partagent donc les éléments des paramètres tableaux.

Les tableaux comme arguments

Il est possible de passer des tableaux comme argument à une fonction VECTRAN. La seule limitation est liée aux tableaux décrits. Le passage d'un tel tableau est réalisé par consommation de la description. C'est-à-dire que la description produit une liste de valeurs qui sont elles passées à la fonction appelée. Il n'est donc pas possible de partager ainsi des éléments de tableaux entre fonctions appelante et appelée. Une solution présentée par Paul et Wilson est de passer deux paramètres : le tableau et la description, puis de reconstruire le vecteur décrit dans la fonction appelée. Cette méthode est la seule envisageable pour les descriptions par une liste d'index. A l'inverse, le passage de paramètres de descriptions par triplets d'un vecteur est réalisé par le passage d'un tableau virtuel obtenu par une instruction **INDENTIFY**. Le compilateur est alors à charge de manipuler une structure interne qui ne peut être réduite à un pointeur vers un tableau comme il en est d'habitude en FORTRAN.

Les informations de rank, range, et type sont locales et associées au nom d'un tableau. Lors du passage en paramètre d'un tableau, ces informations sont perdues. Une déclaration locale à la fonction appelée redéfinit éventuellement ces informations pour le paramètre.

Résultat tableau

Une fonction peut retourner un tableau, elle doit alors apparaître dans une déclaration **RANGE** qui détremine les caractéristiques du tableau retourné. La taille de ce tableau doit être connue lors de la compilation afin de réaliser, à ce stade, son allocation.

Attributs ELEMENTAL *et* GENERIC

On peut spécifier qu'une fonction est **ELEMENTAL** lors de son appel. Les paramètres formels de la fonction doivent être scalaires, les paramètres effectifs peuvent être scalaires ou tableaux. Tous les paramètres tableaux doivent être conformes; le résultat est alors un tableau conforme avec les tableaux arguments de la fonction. Le résultat est obtenu élément par élément par une boucle implicite incluant un appel à la fonction scalaire pour chacun des éléments des tableaux paramètres effectifs.

L'attribut **GENERIC** est réservé aux fonctions prédéfinies. Les type et constructeur (scalaire/tableau) du résultat dépendent de ceux des arguments (le résultat est un tableau si les arguments sont des tableaux, etc.).

Conclusion

VECTRAN a le mérite d'être un des premiers langages vectoriels indépendant de toute machine cible. De nombreuses idées furent introduites par VECTRAN bien que certaines d'entre elles n'aient pu être reprises ensuite faute de leur trouver une implantation efficace sur les machines vectorielles (par exemple les fonctions). Les motivations de Paul et Wilson tel le principe de contention sont importantes. Les points forts de VECTRAN sont la compatibilité avec FORTRAN, l'instruction de création dynamique de tableaux virtuels **IDENTIFY**, la dynamicité des ranges des tableaux, etc. A l'inverse, on peut regretter une syntaxe trop lourde, la non-uniformité entre les descriptions par liste d'entiers et par triplet (par exemple lors du passage de paramètre) et l'absence de description par liste de booléens (seulement lors du rangement par **AT** et **WHEN**).

1–4.2 Le premier langage vectoriel "commercial" : FORTRAN 200

FORTRAN 200 est une extension de FORTRAN qui fut proposée par Control Data pour les machines Cyber 200, 203 et 205. FORTRAN 200 inclut d'une part une reconnaissance automatique des instructions vectorielles au sein de code scalaire, et d'autre part une extension "vectorielle" autorisant la manipulation de vecteurs en tant que tels. Nous nous intéressons ici surtout à cette extension vectorielle. Notons quand même qu'un tel langage autorise un portage en douceur d'une application existante écrite en FORTRAN scalaire. Par ailleurs, les constructions vectorielles proposées par FORTRAN 200 peuvent sembler trop proches de l'architecture de la machine ciblée et même limitées à celle-ci pour que ce langage puisse être qualifié de "langage vectoriel explicite". FORTRAN 200 est un langage vectoriel dédié à une machine cible. Le chapitre 9 "vector programming" du manuel de référence FORTRAN 200 [CDC83] est la principale source d'informations sur les possibilités offertes par FORTRAN 200 en matière de programmation vectorielle. Un chapitre du livre de Perrott [Perr87] est consacré en partie à FORTRAN 200.

La spécification ROWWISE

FORTRAN 200 est une extension de FORTRAN. Une des constructions introduites par FORTRAN 200 est la spécification **ROWWISE**. En FORTRAN, un tableau multi-dimension est alloué en mémoire colonne par colonne. Une spécification **ROWWISE** définit une allocation ligne par ligne. Par exemple suite aux déclarations des tableaux *COL* et *LIG*

```
DIMENSION   COL (3, 4)
ROWWISE     LIG (3, 4)
```

les éléments des tableaux sont rangés dans l'ordre suivant en mémoire :

```
1  :        COL (1, 1)        LIG (1, 1)
2  :        COL (2, 1)        LIG (1, 2)
3  :        COL (3, 1)        LIG (1, 3)
4  :        COL (1, 2)        LIG (1, 4)
5  :        COL (2, 2)        LIG (2, 1)
6  :        COL (3, 2)        LIG (2, 2)
7  :        COL (1, 3)        LIG (2, 3)
8  :        COL (2, 3)        LIG (2, 4)
9  :        COL (3, 3)        LIG (3, 1)
10 :        COL (1, 4)        LIG (3, 2)
11 :        COL (2, 4)        LIG (3, 3)
12 :        COL (3, 4)        LIG (3, 4)
```

Cette spécification **ROWWISE** est intéressante car sur la machine visée par FORTRAN 200, seules des suites d'éléments contigus en mémoire, éventuellement filtrés par un masque de bits, peuvent être traitées par les unités vectorielles. En adaptant l'allocation au traitement, il est ainsi possible de considérer des vecteurs aussi bien dans des colonnes que dans des lignes de matrices.

Vecteurs et descripteurs

FORTRAN 200 propose deux constructeurs autorisant la manipulation de vecteurs : les *vector references* (références de vecteurs, ou vecteurs) et les *descriptors* (descripteurs). Un vecteur est une série de valeurs allouées consécutivement en mémoire. Un vecteur référence une suite d'éléments d'un tableau. Un vecteur est de la forme suivante :

nom_de_tableau (base; longueur)

base est l'indice du premier élément du vecteur dans le tableau nom_de_tableau; longueur le nombre d'éléments formant le vecteur. Cette longueur du vecteur est limitée à 65535, reflétant une limitation de l'architecture de la machine. Outre les informations de base et de longueur, les éléments référencés par un vecteur sont déterminés par la déclaration du tableau, et l'éventuelle spécification de **ROWWISE** pour ce tableau. Soient les déclarations

```
DIMENSION A (10), B (10, 10)
ROWWISE C (10, 10)
```

les vecteurs	référencent les suites d'éléments

`A (3; 5)`	`A (3)`
	`A (4)`
	`A (5)`
	`A (6)`
	`A (7)`

`B (1,1; 5)`	`B (1, 1)`
	`B (2, 1)`
	`B (3, 1)`
	`B (4, 1)`
	`B (5, 1)`

```
C (1,1; 5)                    C (1, 1)
                              C (1, 2)
                              C (1, 3)
                              C (1, 4)
                              C (1, 5)
```

L'autre construction proposée par FORTRAN 200 est le descripteur. Un descripteur référence un vecteur par un nom. Un descripteur est implanté sous la forme d'un couple (nombre d'éléments, adresse du premier élément). L'association entre le descripteur et un vecteur est réalisée soit par une instruction **DATA**, soit par une affectation de descripteur **ASSIGN**. L'association au moyen de **DATA** est réalisée lors de la compilation alors que l'association **ASSIGN** est dynamique. Les initialisations

```
REAL A (5), B (10), C (9)
REAL V1, V2, V3 (5)
DESCRIPTOR V1, V2, V3
DATA V1, V2 /C (1;5), C (5;5)/
DATA V3 /A (1;5), B (1;5), 3*C (1;7)/
```

associent respectivement les descripteurs *V1* et *V2* avec les éléments *C (1)* à *C (5)* et *C (5)* à *C (9)* d'une part et les éléments du tableau de descripteurs *V3* respectivement avec les vecteurs *A (1;5)*, *B (1;5)*, *C (1;7)*, *C (1;7)*, et *C (1;7)* d'autre part.

Une association **ASSIGN** est une instruction; elle est donc réalisée dynamiquement lors de l'exécution. Deux formes de **ASSIGN** sont distinguées. La première forme de cette instruction réalise la même association qu'une instruction **DATA**. La seconde forme de **ASSIGN** alloue dynamiquement une zone de données pour stocker le nombre d'éléments souhaité. L'instruction **FREE** libère toutes les zones allouées dynamiquement dans l'unité de programme en cours.

```
.
.
REAL A (5), B (10), C (9)
REAL V1, V3 (5)
DESCRIPTOR V1, V3
.
.
ASSIGN V1, C (1;5)
ASSIGN V3 (1), A (1;5)
ASSIGN V3 (2), V1
.
.
ASSIGN V1, .DYN. INT_EXPR
ASSIGN V3 (3), .DYN. 100
ASSIGN V3 (4), .DYN. 250
.
.
FREE
.
.
```

Toutefois, à la suite d'une opération **ASSIGN**, le descripteur ne peut être manipulé qu'en tant que vecteur; il n'est pas possible d'accéder individuellement à ses éléments; *V1 [i]* est invalide.

Expressions et affectations

Les expressions FORTRAN 200 sont étendues aux vecteurs. Les références de vecteurs et les descripteurs peuvent apparaître dans les expressions. Toutes les références de vecteurs et descripteurs d'une expression doivent avoir la même longueur. Sémantiquement, les opérations sont effectuées élément par élément. L'instruction d'affectation est également étendue aux références de vecteurs et aux descripteurs. Une telle instruction affecte les éléments de tableau référencés par la partie gauche avec l'élément correspondant de la partie droite. Les deux morceaux de codes

```
REAL A (5), B (10), C (9)
C (5;5) = A (1;5) + 2 * B (1;5)
```

et

```
REAL A (5), B (10), C (9)
REAL V1, V3 (5)
DESCRIPTOR V1, V3
DATA V1 /C (5;5)/
DATA V3 /A (1;5), B (1;5), 3*C (1; 7)/
V1 = V3 (1) + 2 * V3 (2)
```

sont donc équivalents et produiront le même résultat (pas forcément le même code objet) que :

```
      REAL A (5), B (10), C (9)
      DO 10 i = 1, 5
10    C (i+5) = A (i) + 2 * B (i)
```

Le constructeur WHERE

FORTRAN 200 fournit un constructeur **WHERE** tel que celui défini au paragraphe 1–3.3.5. Les affectations de références de vecteurs et de descripteurs présentes dans un bloc **WHERE** sont alors contrôlées par un vecteur de bits, nommé *vecteur de contrôle*. C'est-à-dire que seuls les éléments de vecteurs, dont l'élément correspondant du vecteur de bits est à 1 (vrai), ont leur valeur modifiée lors d'une affectation. Les instructions qui ne sont pas des affectations vectorielles ne sont pas autorisées dans un bloc **WHERE**. Les expressions vectorielles d'un tel bloc doivent être formées d'opérations arithmétiques (+, –, *, et /) et d'appels à des fonctions vectorielles prédéfinies (*VSQRT*, *VABS*, etc...). Les vecteurs opérandes de ces appels de fonctions sont eux aussi contrôlés par le vecteur de bits. Tous les opérandes vecteurs d'un tel bloc doivent être de même longueur : la longueur du vecteur de contrôle.

L'utilisation du constructeur **WHERE** est facilitée par l'adjonction du type **BIT** aux types habituels FORTRAN. Une variable de type **BIT** représente un vecteur de bits. Ces objets sont manipulés par les expressions de FORTRAN 200 et produits par les opérateurs de comparaisons appliqués aux vecteurs, par exemple

```
REAL A (5), B (5)
BIT VBIT (5)
VBIT (1;5) = A (1;5) .GT. B (1;5)
```

Les fonctions et sous-programmes

FORTRAN 200 étend les fonctions et sous-programmes FORTRAN aux vecteurs. Des paramètres vecteurs peuvent être passés à une fonction; une fonction peut retourner un vecteur; et un ensemble de fonctions vecteurs prédéfinies est fourni.

Lors de la définition d'une fonction, on peut préciser que certains paramètres formels sont des descripteurs. Les paramètres effectifs devront alors être soit des descripteurs, soit des références de vecteurs. A l'appel de la fonction, une taille et une adresse de base de vecteur seront alors passées.

Une fonction peut retourner un résultat vecteur. Dans le corps de la fonction, on affecte le nom de la fonction comme un descripteur. L'instruction d'appel de la fonction précise comment est retourné ce résultat. Deux alternatives sont proposées : allouer une zone temporaire ou utiliser un vecteur pour retourner le résultat; elles sont illustrées par l'exemple suivant. Lors de l'appel, le "paramètre"suivant le ; spécifie soit la taille à allouer pour la zone temporaire, soit le nom du vecteur dans lequel retourner le résultat.

```
PROGRAM VECFUNC
REAL A (100), B (100), C (100)
C (1; 100) = VFONCT (A (1; 100), B (1; 100); 100)
END

FUNCTION VFONCT (VA, VB; *)
DESCRIPTOR VFONCT, VA, VB
VFONCT = VA + VB
RETURN
END
```

L'appel de la fonction *VFONCT* s'écrit aussi

```
C (1; 100) = VFONCT (A (1; 100), B (1; 100); C (1; 100))
```

et utilise *C (1 ; 100)* comme zone temporaire pour retourner le résultat.

Ce mécanisme autorise de manière simple la définition de fonction acceptant des arguments vecteurs de taille non fixée et le retour de vecteurs de taille variable.

La bibliothèque du FORTRAN 200 fournit des fonctions de réduction de vecteurs (par exemple **Q8SSUM** retourne la somme des éléments d'un vecteur). Outre le paramètre vecteur passé à ces fonctions, certaines acceptent un paramètre *vecteur de contrôle*; l'opération n'est alors réalisée que pour les éléments du vecteur dont l'élément correspondant du vecteur de contrôle est vrai. Les opérations de gather/scatter sont également réalisées par des appels à des fonctions de la bibliothèque *Q8* (**Q8VGATH** et **Q8VSCATR**).

Discussion

Les constructions fournies par le FORTRAN 200 sont fortement liées à l'architecture de la machine cible. La manipulation de vecteurs de bits est encouragée (déclaration **BIT**, constructeur **WHERE**) car ces objets sont facilement implantés sur les machines CDC. A l'inverse, les opérations de gather/scatter, qui ne sont disponibles que via la bibliothèque, ne sont pas efficacement implantées dans l'architecture de ces machines. La taille des vecteurs est au maximum 65535, reflétant ainsi une limitation imposée par

l'architecture (taille des pages de la mémoire paginée). Cependant, le fait qu'un même langage propose à la fois des outils syntaxiques d'expression du vectoriel et une reconnaissance automatique des structures vectorielles dans un programme scalaire est un point notablement intéressant.

Notons que les constructions de FORTRAN 200, à l'exception de la vectorisation automatique, sont reprises dans le langage VECTOR C développé par Li et Schwetman [LiSc84], [LiSc85], et [Li86].

1–4.3 Le langage ACTUS – Manipulation d'*extent of parallelism*

Le langage ACTUS fut proposé par R.H. Perrott dans le milieu des années 1970. ACTUS est une extension parallèle du langage PASCAL. Une première version de ACTUS fut implantée sur Cray 1; une seconde version légèrement modifiée et référencée comme ACTUS II est implantée sur ICL DAP.

Les papiers de Perrott [Perr79a] et [PCM83], présentent ACTUS. D'autres informations sont disponibles dans le manuel de référence [Perr79b]. L'implantation de ACTUS est brièvement discutée dans [PCMP83], [PCM84], et [PCMP85]. ACTUS II est introduit dans [PLD87]. L'environnement de programmation développé autour d'ACTUS est présenté dans [PeZa87] et [PeLu91].

Les points suivants sont donnés par Perrott comme sous-jacents à la conception de ACTUS en tant que langage pour machine SIMD :

1) le langage doit ignorer les caractéristiques propres d'une architecture cible particulière;

2) le programmeur doit être à même d'exprimer directement le parallélisme de son problème;

3) le taux de parallélisme doit être variable et non fixé par des considérations matérielles par exemple le nombre de processeurs élémentaires;

4) le contrôle du parallélisme doit être à la fois explicite et possible au travers les données.

De plus Perrott est convaincu de l'inadéquation de FORTRAN à la programmation vectorielle, tant par son aspect "ancien" opposé aux vues plus "génie logiciel" de Perrott, que par les extensions peu portables qui ont été proposées pour FORTRAN.

Tableaux parallèles ACTUS

ACTUS est développé autour de la notion de *extent of parallelism* ou *EOP*. Un EOP est associé à chaque instruction vectorielle ACTUS et à chaque déclaration de tableau. Un EOP indique le nombre de données sur lesquelles une instruction peut être appliquée simultanément. ACTUS est un langage de manipulation d'extent of parallelism. Lors de la déclaration d'un tableau, on indique la ou les dimensions qui seront accédées en parallèle par un : "parallel dot" à la place du .. de la déclaration PASCAL habituelle.

```
SCALAR : array [1..200] of real;
PARALLEL : array [1:200] of real;
```

déclare un tableau *SCALAR* dont les composantes ne pourront pas être accédées en parallèle et un tableau *PARALLEL* dont les composantes pourront être accédées en parallèle; son EOP est borné par 200. L'EOP

est déclaré indépendamment pour chaque dimension. Soit la multiplication d'une matrice *Matrice* de taille *M* * *N* par un vecteur *V* de taille *N* produisant un vecteur *U* de taille *M*.

```
const
      M = 200; N = 123;                    (* valeurs arbitraires *)
var
      Matrice : array [1:M, 1..N] of real;
      V : array [1..N] of real;
      U : array [1:M] of real;
      j : 1..N;
begin
      U [1:M] = 0.0;
      for j := 1 to N do
            U [1:M] = U [1:M] + Matrice [1:M, j] * V [j];        (* S1 *)
end
```

La matrice *Matrice* est déclarée avec un EOP de *M* sur la première dimension. Les éléments d'une colonne peuvent donc être accédés en parallèle (instruction *S1*). Les éléments d'une ligne doivent être accédés un par un, comme en PASCAL. Le nombre de dimensions accessibles en parallèle est limité à un; cela est motivé par le fait que les processeurs vectoriels ne présentent qu'un seul niveau de parallélisme. ACTUS II autorise deux dimensions d'un tableau à être accédées en parallèle, reflétant ainsi le tableau de processeurs de la machine visée, l'ICL DAP et plus généralement des machines tableaux.

Index sets et description en ACTUS

ACTUS propose des *index sets* qui sont des objets exprimant un EOP; la valeur de ces objets doit être fixée à la compilation; ils sont ensuite utilisés comme description d'une dimension parallèle d'un tableau. Illa valeur d'un tel objet est obtenue depuis une expression construite ssuivant la syntaxe

borne inférieure : [*pas*] *borne supérieure* ou *borne inférieure* : *borne supérieure*

```
const
      M = 200; N = 100; M1 = 199
var
      GRID : array [1:M, 1..N] of real;
index
      INSIDE = 2:M1
      ODD = 1 :[ 2 ] M1
      EVEN = 2 :[ 2 ] M
```

Les éléments pairs de la $i^{ème}$ colonne de *GRID* sont accédés en parallèle

```
GRID [ODD, i]
```

Les EOPs des tableaux présents dans une instruction doivent être identiques;

```
GRID [ODD, i] = GRID [EVEN, j]     (* INVALIDE, EOP non identiques *)
```

Il faut utiliser les opérateurs d'alignement de EOP `shift` ou `rotate` :

```
GRID [ODD, i] = GRID [ODD shift +1, j]
```

La manipulation d'EOP est facilitée par différents constructeurs et par l'extension des structures de contrôle scalaires à des opérandes tableaux. Le constructeur **within** définit le EOP courant qui est ensuite référencé par le signe dièse # :

```
within ODD do
      GRID [#, i] = GRID [# shift +1, j]
```

Le EOP courant est aussi affecté par les constructions **if**, **case**, **while**, **repeat**, et **for** dont les prédicats sont des tableaux.

```
within 1 : P do begin                    (* EOP *)
      A [#] := - B [#]                   (* 1:P *)
      if A [#] > 0.0 then                (* 1:P *)
            A [#] := A [#] / 2           (* 0 <= # <= P *)
      X [#] := A [#]                     (* 1:P *)
end
```

Dans l'instruction contrôlée par le prédicat tableau *A [#] > 0.0*, seules les valeurs de *A* strictement positives sont divisées par 2.

```
while A [1:N] > 0.0 do begin
      A [#] := A [#] -1
      B [#] := B [#] * A [#]
end
```

Les instructions de la boucle sont exécutées pour un EOP correspondant aux valeurs positives de *A* et tant qu'il existe une valeur positive de *A* (EOP non nul).

Les fonctions en ACTUS

Les fonctions ACTUS acceptent des paramètres tableaux et retournent des valeurs tableaux. Il est possible de passer des tranches d'un tableau à une fonction. Toutefois, les dimensions parallèles et scalaires des tableaux doivent correspondre.

```
var
      tableau : array [ 1..10 , 1:100 , 1..10 , 1..10 ] of real;
procedure p (var x : array [ extent, a .. b] of real);
      begin
      ...
      end
begin
      p (tableau [ 1 , 20:40, 2..4 , 1]);   (* extent=20:40, a=2, b=4 *)
      p (tableau [ 1 , 1:10, 1 , 3..7]);    (* extent=1:10, a=3, b=7 *)
end
```

La description par un vecteur de valeurs entières est réalisée comme une projection (cf. section 1–3.3.4, page 21). L'exemple suivant initialise les valeurs de la diagonale de *bigarray* à zéro.

```
var
      bigarray : array [1:10, 1..10] of real;
      indexarray : array [1:10] of integer;
begin
      within 1:10 do begin
            indexarray [#] = 1:10;
            bigarray [#, indexarray [#]] = 0.0;
      end
end
```

Lors de la déclaration d'une dimension scalaire d'un tableau, il est possible de préciser le nombre d'éléments de cette dimension qu'il est nécessaire de charger lors de transfert de la mémoire centrale vers une mémoire locale ou une mémoire cache.

Implantation

L'implantation de ACTUS est réalisée via une série de primitives de manipulations des EOP par des objets de type *runtimeset* (RTS) au sein d'une représentation intermédiaire de type arbre abstrait. Les RTS sont des EOP dont les caractéristiques ne sont pas fixées à la compilation. Ces primitives initialisent un RTS à partir d'un EOP (**setfrom**) ou à partir des éléments de valeur vraie d'un vecteur (**truevalues**), indique si un RTS est vide ou non (**anymember**), et retourne une RTS complément d'un autre RTS avec un EOP. Par exemple

```
while a [1:10] > 0 do
      a [#] := a [#] -1
```

est codé en (la construction **eop** *RTS* **do** est l'équivalente du **within** pour les RTS, les *#rtsi#* sont des variables de type RTS)

```
begin
#rts0# := setfrom (1 :[1] 10);
eop #rts0# do begin
      #rts1# := truevalues (a [#] > 0);
      if anymember (#rts1#) then
            repeat
                  eop #rts1# do begin
                        a [#] := a [#] - 1
                        #rts1# := truevalues (a [#] > 0)
                  end
            while anymember (#rts1#)
end
end
```

Discussion

Le langage ACTUS propose des constructions facilitant la déclaration et la manipulation d'*extent of parallelism* dénotant les ensembles d'éléments d'un tableau manipulables en parallèle. Il est à regretter la non "orthogonalité" et la limitation de certaines constructions proposées. Par exemple les EOP correspondant à une liste de valeurs booléennes ne peuvent être exprimés que par une construction **if**

vectorielle. Les index set (objets représentant un EOP triplet) ne sont pas manipulables par les instructions du langage; leurs valeurs doivent donc être fixées à l'initialisation. A l'inverse, les opérations **`shift`** et **`rotate`** des EOPs et leurs extensions en deux dimensions **`Vshift`**, **`Vrotate`**, **`Hshift`**, ... facilitent l'implémentation des descriptions, par exemple sous forme de communications aux voisins dans le cas d'une machine tableau.

1–4.4 La nouvelle norme FORTRAN : FORTRAN 90

FORTRAN 90 est la nouvelle norme du langage FORTRAN. Après la mise en place du dernier standard FORTRAN 77, un comité fut chargé de travailler sur une nouvelle révision du langage : FORTRAN 8x. Après de nombreux travaux, commissions, débats, et commentaires, le *x* fut fixé à 10 et le projet de norme approuvé. FORTRAN est un langage à usage général. FORTRAN est même l'un des langages les plus utilisés par les scientifiques et les numériciens. C'est certainement pourquoi FORTRAN 90 inclut des éléments de manipulation de vecteurs. Cette section ne liste et ne détaille pas toutes les constructions de FORTRAN 90. Nous ne nous intéressons qu'aux aspects de FORTRAN 90 qui concernent le vectoriel. On pourra consulter un des derniers drafts de la norme [ANSI91] ou le livre de Metcalf et Reid [MeRe90] pour de plus amples informations sur les autres aspects de FORTRAN 90. La publication de Reid [Reid89] résume en 10 pages les apports de FORTRAN 90.

Opérateurs et instructions

FORTRAN 90 est un langage vectoriel explicite. Les tableaux sont manipulés en tant qu'entités dans le langage. Les opérateurs scalaires et les fonctions prédéfinies ont été étendus aux tableaux. Par exemple l'expression

```
B + C * SIN (D)
```

est valide pour B, C, D vecteurs ou scalaires. La plupart des fonctions intrinsèques de FORTRAN 77 on été étendues aux arguments tableaux. Les expressions ne sont valides que si toutes leurs opérandes tableaux sont conformes, c'est-à-dire de même forme (même nombre de dimensions et même taille dans chaque dimension). L'affectation d'un tableau est définie comme évaluant entièrement sa partie droite avant de ranger le résultat dans la partie gauche.

Les littéraux

FORTRAN 90 étant un langage manipulant des tableaux en tant que tels, un constructeur est proposé pour la création de tableaux constants ou de valeurs littérales tableaux. Un tableau mono-dimension est construit comme une liste de valeurs immédiates entre parenthèses et barres obliques; par exemple

```
(/ 1, 2, 3, 5, 10 /)
```

une liste de valeurs consécutives ou régulièrement espacées peut former un tableau littéral comme par exemple

```
(/ (I, I=1,5) /)        équivalent à   (/ 1, 2, 3, 4, 5 /)
```

et

```
(/ (I, I=1,10,2) /)          équivalent à   (/ 1, 3, 5, 7, 9 /)
```

On peut combiner et emboîter les deux constructions :

```
(/ 1, 4, (I, I=1,3) /)       équivalent à   (/ 1, 4, 1, 2, 3 /)
```

et

```
(/ ((I, I=1,3), J=1,3) /)    équivalent à   (/ 1, 2, 3, 1, 2, 3, 1, 2, 3 /)
```

Ces littéraux sont mono-dimensionnels. Des littéraux multi-dimensionnels sont construits en utilisant la fonction intrinsèque **RESHAPE**;

```
RESHAPE (SHAPE = (/ 2, 3 /), SOURCE = (/ 1, 2, 3, 4, 5, 6 /) )
```

désigne la matrice

```
1     3     5
2     4     6
```

Ces constructions de valeurs littérales ne sont pas limitées aux tableaux d'entiers mais s'appliquent à tous les types FORTRAN.

L'allocation des tableaux

En FORTRAN 77, l'allocation des tableaux, comme celle de tous les objets, est réalisée de manière statique. FORTRAN 90 met en place deux nouvelles allocations pour les tableaux (les allocations *automatique* et *dynamique*) ; on distingue donc :

- **les tableaux statiques** Les tableaux statiques sont repris de FORTRAN 77. L'allocation d'un tableau statique est déterminée à la compilation dans la zone des données statiques; leur taille est donc connue dès la compilation.

- **les tableaux automatiques** Les tableaux automatiques sont alloués automatiquement à l'entrée d'une fonction et désalloués à la sortie de la fonction. Par exemple, leur taille est un paramètre passé à la fonction.

- **les tableaux dynamiques** Les tableaux dynamiques sont explicitement alloués et désalloués par le programmeur via les instructions **ALLOCATE** et **DESALLOCATE**. Pour qu'il en soit ainsi, le tableau doit être spécifié **ALLOCATABLE** lors de sa déclaration. Lors de cette déclaration, la taille de chacune de ses dimensions n'est pas donnée (spécification **DIMENSION**); elle sera précisée lors de l'allocation :

```
INTEGER N
REAL, DIMENSION (:,:), ALLOCATABLE :: A
...
READ (INPUT, *) N
ALLOCATE (A (N, N*N))
...
DESALLOCATE (A)
```

Les tableaux dynamiques seront alloués sur un tas.

Les descriptions

FORTRAN 77 autorise la sélection d'un élément d'un tableau. FORTRAN 90 étend cette sélection d'éléments dans deux directions : la sélection par un triplet et la sélection par une liste d'index (tableau d'entiers), et ce pour chaque dimension. La sélection par un triplet sélectionne des éléments contigus ou également espacés dans une dimension. La syntaxe générale est

[<lower>] : [<upper>] [: <step>]

Les valeurs par défaut des lower et upper sont les limites de variation des indices dans la dimension décrite (définis lors de la déclaration ou l'allocation); la valeur par défaut du step est 1.

```
A (I, 1:N)        ! les éléments de 1 à N de la ligne I
A (1:M:2, J)      ! les éléments impairs de 1 à M de la colonne J
```

La sélection par un tableau d'entiers mono-dimensionnel (vecteur d'index) réalise les opérations de gather/scatter. Par exemple

```
V ( (/ 1, 7, 3, 2 /) )
```

est composé dans l'ordre des éléments *V (1), V (7), V (3), V (2)*. Soient *Z* une matrice deux dimensions, *U* et *V* deux vecteurs d'entiers de taille 3, et 4 respectivement et de valeur

```
U = (/ 1, 3, 2 /)
V = (/2, 1, 1, 3 /)
```

Alors *Z (3, V)* est composé des éléments de la troisième ligne de *Z* dans l'ordre

```
Z (3, 2)    Z (3, 1)    Z (3, 1)    Z (3, 3)
```

et *Z (U, V)* est

```
Z (1, 2)    Z (1, 1)    Z (1, 1)    Z (1, 3)
Z (3, 2)    Z (3, 1)    Z (3, 1)    Z (3, 3)
Z (2, 2)    Z (2, 1)    Z (2, 1)    Z (2, 3)
```

Lorsqu'une valeur apparaît plusieurs fois dans le vecteur d'index, plusieurs éléments du vecteur décrit représentent un même élément du tableau initial. En FORTRAN 90, cette opération est illégale en écriture :

```
V ( (/ 1, 7, 3, 7 /) ) = (/ 1, 2, 3, 4 /)     ! Illégal
```

Les "pointeurs" comme alias

FORTRAN 90 introduit la notion de pointeur dans le langage FORTRAN. Des constructions et manipulations de structures sont également disponibles en FORTRAN 90 sous le vocable de *derived data types*. La combinaison de ces deux types d'objets autorise la manipulation de structures de données complexes. Nous nous intéressons ici à une utilisation particulière des pointeurs proposée par FORTRAN 90. La déclaration d'un pointeur sur un tableau définit le nombre de dimensions de ce tableau :

```
REAL, DIMENSION (:,:), POINTER :: PTR
```

PTR est un pointeur sur un tableau de deux dimensions de réels. On pourra l'associer à un objet défini comme cible d'un pointeur (mot clé **TARGET**). Suite à la déclaration du tableau *TAB*

```
REAL, DIMENSION (3, 5), TARGET :: TAB
```

Il est possible d'associer le pointeur *PTR* par

```
PTR => TAB
```

à la suite de quoi le pointeur *PTR* est un alias pour le tableau *TAB* et peut être utilisé comme tous les tableaux FORTRAN 90 (sauf en temps que paramètre, cf. ci-dessous). Plus intéressant est la possibilité d'associer une section (description par triplet) de tableau à un pointeur; par exemple :

```
PTR => TAB (L1:U1:S1, L2:U2)
```

Les expressions apparaissant dans la description par triplet sont évaluées lors de l'association du pointeur. Le pointeur *PTR* référence ensuite les éléments de *TAB* sélectionnés par la description. Les descriptions par une liste d'index ne peuvent pas être associées à un pointeur. Cette limitation simplifie la représentation de ces pointeurs mais n'est certainement pas satisfaisante pour le programmeur.

Constructeur WHERE

FORTRAN 90 reprend le constructeur **WHERE** d'affectation masquée de FORTRAN 200. Cette construction utilise un tableau de type **LOGICAL**. Les objets **LOGICAL** prennent l'une des deux valeurs **.TRUE.** ou **.FALSE.**. La syntaxe d'un bloc **WHERE** est

WHERE (<logical–expr>) <assignment–stmt–list> **ENDWHERE**

ou

WHERE (<logical–expr>) <assignment–stmt–list>
ELSEWHERE<assignment–stmt–list> **ENDWHERE**

Une <assignment–stmt–list> est une suite d'instructions composée uniquement d'affectations de tableaux de forme identique à celle de l'expression <logical–expr>. La partie droite d'une affectation de tableau contenue dans un **WHERE** n'est évaluée que pour les éléments dont l'élément correspondant de la <logical–expr> est vrai. Ces éléments sont ensuite rangés dans les éléments correspondant de la partie gauche.

FORTRAN 90 propose des fonctions prédéfinies dites "fonctions élémentaires". Ces fonctions sont des extensions des fonctions prédéfines scalaires. Le résultat est obtenu par application de l'opération scalaire à chaque ensemble d'éléments correspondants des vecteurs opérandes. Si une fonction "non élémentaire" apparaît dans l'expression partie droite d'une affectation d'un bloc **WHERE**, ses paramètres tableaux sont entièrement évalués et la fonction est entièrement évaluée. A l'inverse, si une fonction "élémentaire" apparaît dans l'expression partie droite d'une affectation, l'opération est effectuée sur ses paramètres tableaux masqués par la logical–expression. Mais un appel à une fonction "élémentaire"comme paramètre d'une fonction "non élémentaire" évaluera entièrement les paramètres tableaux de cette dernière. Exemple donné par le draft FORTRAN [ANSI91] et repris par [MeRe90] :

```
WHERE (A > 0)
      A = LOG (A)
      A = A / SUM (LOG (A))
ENDWHERE
```

La fonction *LOG* est "élémentaire", la fonction *SUM* ne l'est pas. La première affectation provoque un appel à *LOG* pour les seuls éléments positifs de *A*; la seconde affectation appelle *LOG* pour tous les éléments de *A* (entraînant éventuellement une erreur pour les valeurs négatives de *A*).

Les procédures

Les fonctions et sous-programmes FORTRAN 90 sont des extensions des fonctions et sous-programmes FORTRAN 77. Les fonctions peuvent maintenant retourner des valeurs tableaux; les paramètres formels peuvent être "assume shape"; de nouveaux attributs spécifient les caractéristiques des paramètres formels (par exemple **INTENT**); la majorité des fonctions intrinsèques sont étendues aux tableaux; de nouvelles fonctions intrinsèques de manipulation de tableaux sont disponibles. Ces aspects sont détaillés dans les paragraphes suivants.

Retour d'une valeur tableau

Une fonction peut retourner une valeur tableau. Le nombre de dimensions de ce tableau est fixé par la définition de la fonction; la taille de ce tableau doit être fixée lors de la compilation ou ne peut dépendre que des caractéristiques des paramètres effectifs.

Paramètres tableaux

Les tableaux sont passés comme paramètres aux procédures. Comme de règle en FORTRAN, le passage de paramètre se fait par adresse. Les paramètres effectifs sont généralement évalués et une copie de la valeur est passée à la fonction (aucune copie n'est réalisée au retour). Cependant, les paramètres qui sont des variables tableaux éventuellement décrites par un triplet ou par un scalaire sont passées directement à la procédure; le paramètre est partagé par les procédures appelante et appelée. Ainsi, avec la définition du sous-programme *ADD1*

```
FUNCTION ADD1 (TAB)
      REAL, DIMENSION (5) :: TAB
      TAB = TAB + 1
END
```

Dans la suite d'instructions

```
REAL, DIMENSION (10) :: V
CALL ADD1 (V)
CALL ADD1 (V (1:5) )
CALL ADD1 (V ( (/ 6, 7, 8, 9, 10 /) ))
```

la première instruction incrémente tous les éléments de *V*; la seconde instruction ajoute 1 aux éléments *V (1)* à *V (5)*; la troisième instruction ne modifie aucune des valeurs de *V*. Un tel comportement lors du passage en paramètre de vecteur décrits est inacceptable.

Paramètre "assume-shape"

Un paramètre formel tableau d'une procédure est dit *assume-shape* si les tailles de ses dimensions ne sont pas connues lors de la définition de la procédure. Cette taille sera celle du paramètre effectif lors de chaque

appel de la procédure. Notons que seule la taille de chaque dimension est passée avec le paramètre; les bornes inférieure et supérieure ne sont pas connues. Sauf spécification, la borne inférieure est 1 et la borne supérieure la taille dans la dimension. Soit le tableau de 11 éléments *A* :

```
REAL, DIMENSION (0:10) :: A
```

et *APARAM*, un paramètre formel déclaré *assume-shape* dans la fonction :

```
REAL, DIMENSION (:) :: APARAM
```

alors un élément *APARAM (i)* correspond à l'élément *A (i–1)*. Cette caractéristique des tableaux *assume-shape* oblige, lors d'un passage de paramètre tableau, les compilateurs FORTRAN à passer l'adresse du tableau et la taille de celui-ci dans chaque dimension.

Attribut des paramètres

Les spécifications **POINTER** et de **TARGET** d'un paramètre formel doivent correspondre à celles du paramètre effectif; les pointeurs de tableaux et les tableaux sont donc distingués lors du passage de paramètre. Il est possible de spécifier l'utilisation qui sera faite d'un paramètre lors de la définition d'une fonction par une des clauses

```
INTENT (IN)          INTENT (OUT)          INTENT (INOUT)
```

et ainsi éviter des copies intempestives de tableaux entre procédures appelante et appelée.

Procédures intrinsèques

FORTRAN 90 reprend la majorité des procédures intrinsèques de FORTRAN 77 sous forme de procédures élémentaires. Fonctionnellement, l'appel d'une procédure élémentaire avec des arguments tableaux est appliqué à tous les éléments. Citons les fonctions **MIN** retournant la valeur minimum de ses arguments, et **SIN** retournant le sinus de son argument comme exemple.

FORTRAN 90 fournit des fonctions retournant des informations sur ses arguments tel **ASSOCIATE** qui retourne vrai si et seulement si un pointeur et une cible passés en paramètre sont associés (le pointeur référence la cible). Des fonctions de multiplication de matrices et de vecteurs sont également disponibles. Des fonctions de réduction de tableaux retournent par exemple la somme des éléments du tableau argument (**SUM**) ou la valeur de l'élément de valeur minimum du tableau paramètre (**MINLOC**). Ces fonctions de réductions acceptent, outre un paramètre tableau, un argument **MASK**, vecteur de **LOGICAL** qui sélectionne les éléments sur lesquels appliquer la fonction,

```
SUM (A, MASK = A > 0)
```

retourne la somme des éléments positifs du tableau *A*. Les fonctions de transformations de tableaux, telles **MERGE** , **PACK** ou **RESHAPE** produisent un tableau à partir d'arguments tableaux et éventuellement scalaires. Au total, FORTRAN 90 propose plus de cent procédures intrinsèques.

Discussion

FORTRAN 90 est largement critiqué pour sa taille et sa complexité trop importante. Est-ce le contrepoids obligé de tout langage généraliste ou est-ce dû à l'historique de FORTRAN ? En ce qui nous concerne, les

extensions proposées pour le traitement vectoriel semblent parfois insuffisantes (il n'existe par exemple pas de description par vecteurs de bits), incomplètes (les pointeurs ne sont autorisés que sur les tableaux décrits par un triplet et non par une liste d'index), ou incohérentes (le masquage ou non par le vecteur de contrôle du **WHERE** des tableaux paramètres de fonctions). Les points forts de FORTRAN 90 sont bien entendu sa compatibilité avec FORTRAN 77 et donc la prise en compte immédiate de très nombreuses applications développées dans ce langage et des connaissances détenues par les programmeurs FORTRAN. On approuvera la puissance des constructions de valeurs littérales à l'aide d'expressions symboliques :

```
(/ I, I-1, I = 1, 100, 2 /)
```

qui peuvent autoriser d'intéressantes optimisations (le vecteur littéral précédent est par exemple la réunion de deux triplets; la prise en compte de cette remarque peut autoriser un compilateur à générer un accès décrit par un tel vecteur littéral comme deux accès par pas) On peut toutefois regretter la relative lourdeur de la syntaxe lors de la description par une liste d'index :

```
A ((/ 1, 2, 3 /))
```

D'ores et déjà, certains constructeurs proposent la prise en compte de toutes ou partie des extensions FORTRAN 90 au sein de leur langage FORTRAN. Citons MasPar FORTRAN [Masp91c] et CM FORTRAN qui est l'objet de la section suivante.

1–4.5 Le FORTRAN de la Connection Machine : CM FORTRAN

CM FORTRAN est une extension du langage FORTRAN fournit par Thinking Machine Corp. sur la Connection Machine. CM FORTRAN reprend une partie des extensions vectorielles de FORTRAN 90 et y ajoute des constructions relatives à l'alignement et la distribution des tableaux. La publication [TMC90a] est une introduction à CM FORTRAN.

Le traitement des tableaux CM FORTRAN est identique à celui de FORTRAN 90. Les tableaux sont donc manipulés en tant qu'entités du langage. Ils peuvent être décrits par des triplets et des vecteurs d'index. L'allocation des vecteurs est inspirée du fonctionnement de la machine cible. Les scalaires et les vecteurs uniquement accédés séquentiellement sont alloués sur la machine hôte; les vecteurs accédés en parallèle sont alloués sur les PEs. Au niveau du programmeur CM FORTRAN, la Connection Machine est constituée d'un nombre virtuellement infini de PEs. Les éléments de vecteurs sont alloués sur les PEs, à raison de un par PE virtuel, suivant une allocation qualifiée de *canonique* dépendant uniquement des informations de forme fournies lors de la déclaration; deux tableaux de même forme seront donc alloués sur le même ensemble de PEs (figure 1–9).

Clairement, pour augmenter les performances de la machine, le nombre de communications inter-processeurs doit être réduit. Pour cela CM FORTRAN propose les directives **LAYOUT** et **ALIGN** qui autorisent le programmeur à gérer de manière fine l'allocation des tableaux sur les PEs. La directive **LAYOUT** indique, pour chaque dimension d'un tableau, si les éléments doivent être alloués sur différents PEs (spécification **:NEWS**) ou dans la mémoire d'un PE (spécification **:SERIAL**). Cette allocation **:SERIAL** ne produit aucune communication lors de traitements mettant uniquement en œuvre des éléments de tableaux alloués sur un même PE (figure 1–10) :

```
DIMENSION   A (2, 4)
DIMENSION   B (2, 4)
```

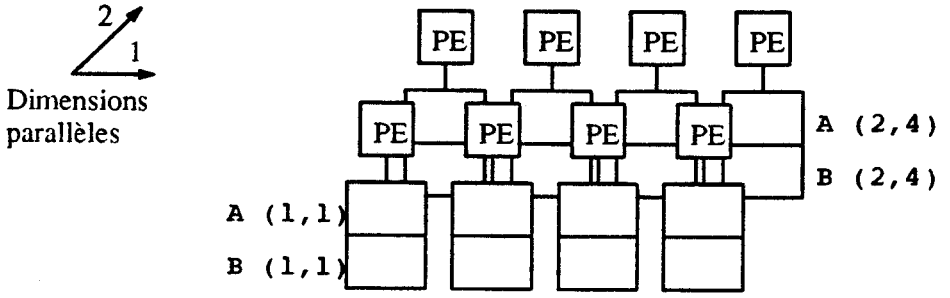


Figure 1–9. Allocation canonique sur les PEs.

```
      DIMENSION   B (3, 2, 4)
CMF$  LAYOUT      B (:SERIAL, :SERIAL, :NEWS)
```

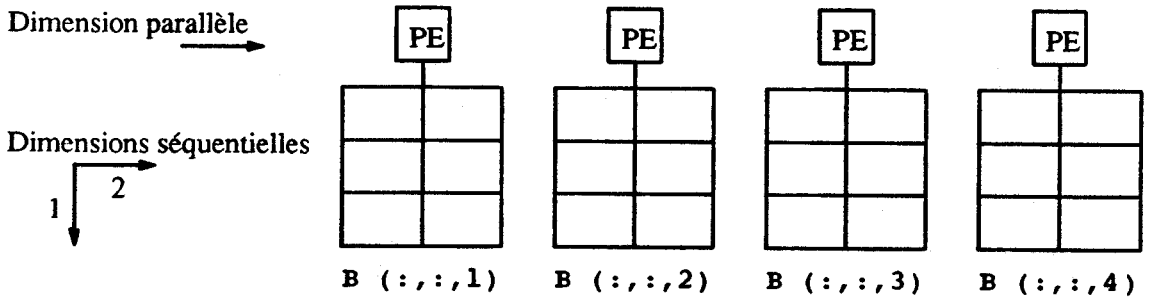


Figure 1–10. Allocation sur les PEs en fonction de la directive **LAYOUT**.

```
      DIMENSION   B (3, 2, 4)
CMF$  LAYOUT      B (:SERIAL, :SERIAL, :NEWS)
B (1,1,:) = (( B(2,1,:) + B (3,1,:) + B (2,2,:) + B (3,2,:)) / 4
```

De telles directives peuvent aussi forcer l'alignement d'éléments de deux tableaux sur un même jeu de PEs :

```
      DIMENSION   A (10, 256, 256))
      DIMENSION   B (256, 256)
CMF$  LAYOUT      A (:SERIAL, :NEWS, :NEWS)
CMF$  LAYOUT      B (:NEWS, :NEWS)

B = A (3,:,:)                    ! pas de communication inter-PEs
```

La directive **ALIGN** est plus particulièrement destinée à allouer les éléments de deux tableaux dans les mêmes PEs de manière contrôlée.

```
      DIMENSION   U (5), V (5), A (5, 5)
CMF$  ALIGN U (I) WITH A (I, 1)
CMF$  ALIGN V (I) WITH A (2, I)
```

aligne les éléments de *U* avec ceux de la première ligne de *A*, et les éléments de *V* avec ceux de la seconde colonne de *A* (figure 1–11).

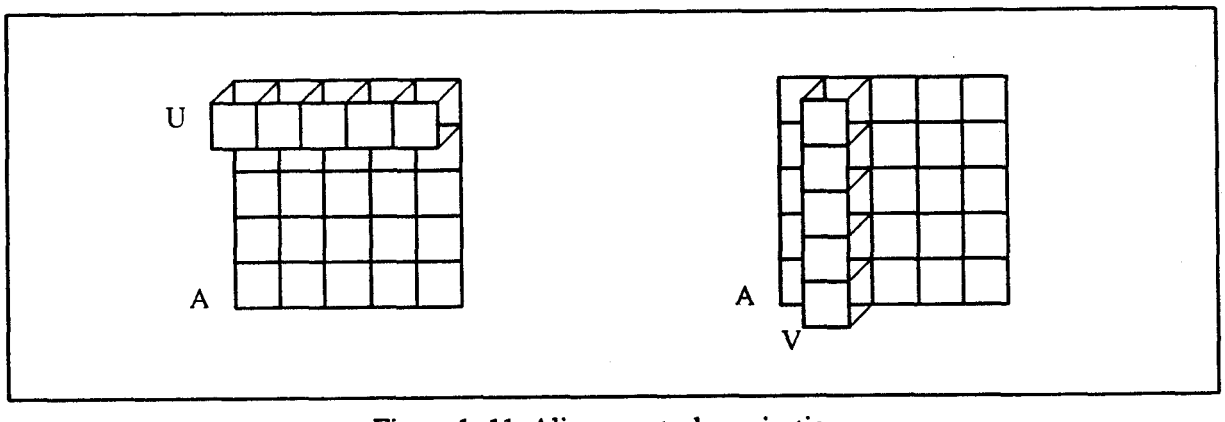


Figure 1–11. Alignements de projection.

Il est possible d'aligner des tableaux de tailles différentes :

```
      DIMENSION   R (5), Q (20)
CMF$  ALIGN R (I) WITH Q (I+5)
```

aligne les éléments de *R* sur ceux de *Q* tel que représenté à la figure 1–12.

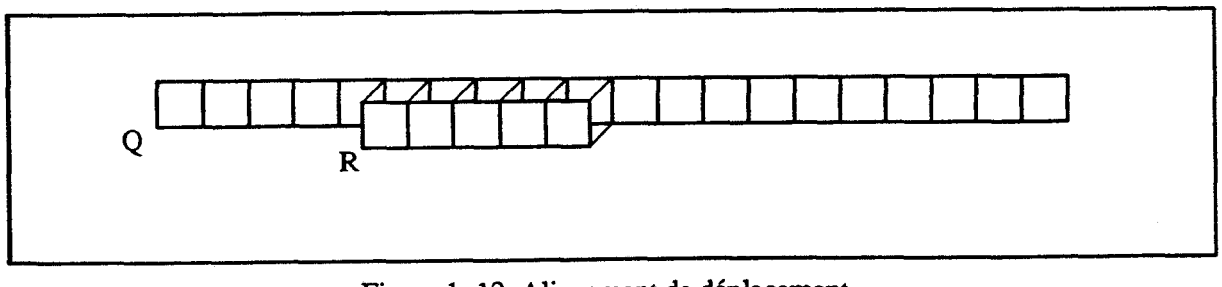


Figure 1–12. Alignement de déplacement.

Des alignements trop complexes correspondant par exemple à des transposées de tableaux ne sont pas supportés par CM FORTRAN. La limitation de base oblige les éléments de vecteurs à toujours être alloués de manière contiguë et croissante sur les processeurs.

L'allocation des tableaux sur la machine hôte ou sur les PEs et les conséquences des directives **LAYOUT** et **ALIGN** doivent être prises en compte par le programmeur lors du passage de paramètres de fonctions; les paramètres formels et effectifs correspondants doivent avoir la même allocation. Sans le respect de cette contrainte, le compilateur ne pourra retrouver les PEs détenant les éléments d'un tableau passé en paramètre. Le passage d'un tableau "aligné" à une fonction demande que le tableau repère soit visible depuis la fonction pour l'utiliser lors de la déclaration de paramètre :

```
      DIMENSION  R (5), Q (20)
CMF$  ALIGN R (I) WITH Q (I+5)
....
CALL SBRT (R, Q)
...
SUBROUTINE SBRT (TAB, REPERE)
      DIMENSION  TAB (5), REPERE (20)
CMF$  ALIGN TAB (I) WITH REPERE (I+5)
...
```

la déclaration du paramètre formel *TAB* utilise le tableau *Q* pour exprimer l'alignement. Cette contrainte n'est pas sans problème pour l'écriture de bibliothèques de fonctions par exemple. Malgré cela, CM FORTRAN est le langage de prédilection des numériciens sur la Connection Machine. Bien entendu, CM FORTRAN est le seul compilateur FORTRAN disponible sur la machine, d'où un certain succès auprès des numériciens. Mais les extensions offertes par CM FORTRAN sont au centre du langage : manipulation explicite des vecteurs, description par des listes d'index et des triplets, construction **WHERE**; elles font elles aussi partie de cette acceptation du langage. Notons aussi que les nouveaux compilateurs (construit sur le modèle "slice-wise") produisent du code bien plus performant ne nécessitant plus l'insertion de code bas niveau (PARIS, cf. section 1–4.9) dans le code FORTRAN pour obtenir des résultats intéressants.

1–4.6 Un langage général pour machines parallèles : FORTRAN D

FORTRAN D est une extension de FORTRAN visant toutes les classes de machines : scalaires, parallèles SIMD, et MIMD. FORTRAN D est développé à Rice University par l'équipe de Ken Kenendy. Le *D* est l'abréviation de Data, Décomposition ou Distribué ! Les publications [HKK+91], [FHK+91], et [HKT91a] sont respectivement une première présentation de FORTRAN D, le manuel de référence du langage, et une introduction à la compilation de FORTRAN D.

Distribution des données

FORTRAN D est une extension de FORTRAN incluant des outils permettant au programmeur de spécifier la distribution des données. La spécification de la distribution se fait en deux étapes : le *problem mapping* et le *machine mapping*. Le problem mapping déclare des décompositions (instruction **DECOMPOSITION**) qui peuvent être vues comme des tableaux non alloués. A l'aide de l'instruction **ALIGN**, on place les

tableaux sur ces décompositions. Le machine mapping place ensuite les décompositions sur les processeurs de la machine (instruction **DISTRIBUTE**).

- De nombreux alignements de tableaux sont possibles : déplacement par rapport à une décomposition, éclatement (rangement régulier suivant un pas) par rapport à une décomposition, permutation (ex. rowwise), regroupement (plusieurs éléments sur un même élément d'une décomposition), duplication (un même élément sur plusieurs éléments d'une décomposition, placement (d'un vecteur sur une matrice 2D par exemple). Se référer à la figure 1–13 pour quelques exemples d'alignements.

- Les distributions des décompositions sur les processeurs de la machine sont : **BLOCK**—les éléments consécutifs d'une décomposition sont "alloués" sur un même processeur, **CYCLIC**—l'élément d'indice *i* est sur le processeur *i modulo Nombre-de-PEs*, **BLOCKCYCLIC**—regroupement d'éléments voisins en bloc, puis distribution cyclique des blocs sur les processeurs (figure 1–14).

Les spécifications d'alignement **ALIGN** et de distribution **DISTRIBUTE** ne sont pas seulement des déclarations, ce sont de véritables instructions. L'alignement et la distribution d'un tableau peuvent donc varier au cours de l'exécution d'un programme.

Procédures et arguments tableaux

Il est permis de passer des tableaux alignés ou distribués comme paramètres de fonctions. Toutefois, la portée d'un alignement ou d'une distribution est locale à une fonction (les distributions et alignements ne sont donc pas affectés par des appels de fonctions, même si la fonction appelée change la distribution ou l'alignement). Le passage d'un tableau aligné/distribué nécessite donc préalablement un ré-alignement et/ou une re-distribution standard. De même lors du retour de la fonction, les tableaux paramètres doivent à nouveau être ré-alignés. La perte d'efficacité induite par un tel mécanisme n'est pas acceptable.

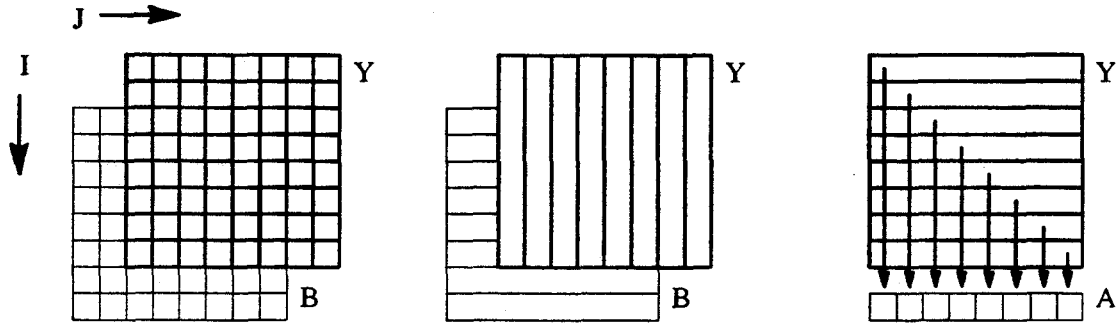
Boucle FORALL

En FORTRAN D, les tableaux ne sont pas manipulables en tant qu'entités. Cependant FORTRAN D propose une boucle **FORALL** dont la sémantique est la suivante : une itération de la boucle ne peut utiliser que des valeurs calculées avant la boucle ou au cours de cette itération. Ce qui peut être exprimé autrement de la manière suivante : une itération de boucle possède sa propre copie de l'espace de données existant avant l'exécution de la boucle. A la fin de la boucle, les valeurs des variables modifiées par différentes itérations sont *rassemblées*. Ce rassemblement affecte à chacune de ces variables la valeur assignée par la dernière itération. Les instructions de la boucle

```
FORALL I = 1, N
      X (IDX (I)) = ...
      ... = X (IDX (I+1))
ENDDO
```

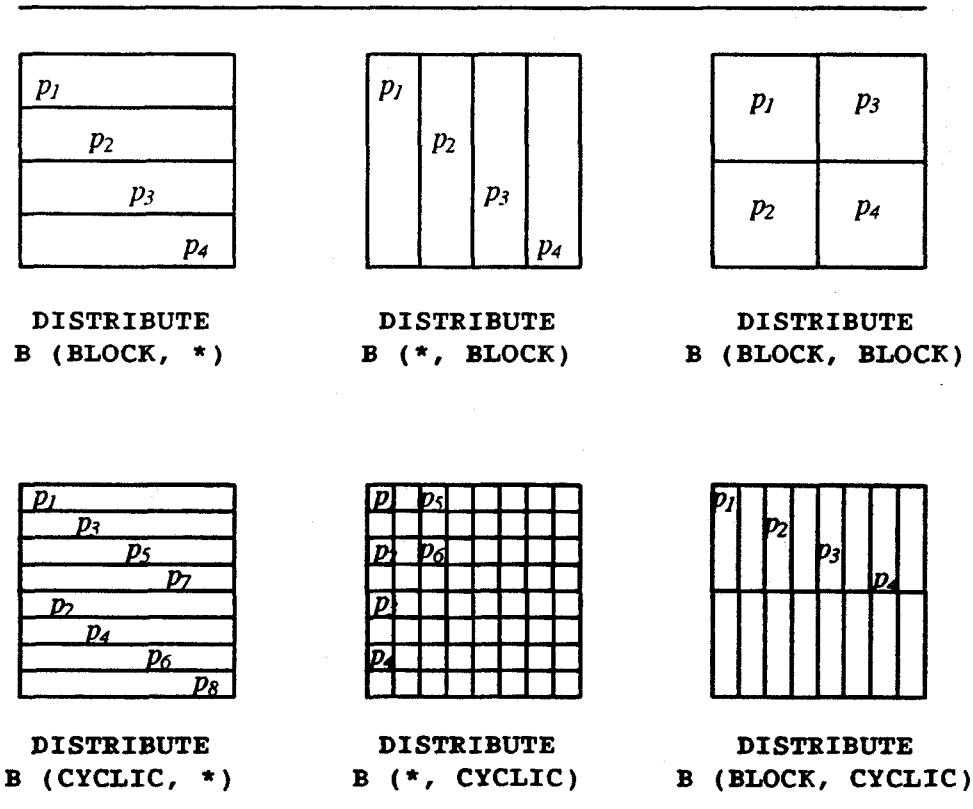
pourront être exécutées en parallèle, malgré la dépendances entre les itérations. FORTRAN D assure que les instructions de la boucle accéderont les valeurs de *X* antérieures à la boucle et non celles produites lors des itérations précédentes. La sémantique d'une telle boucle diffère donc de celle de la boucle FORTRAN classique.

```
REAL Y (N, N)
DECOMPOSITION A (N), B (N,N)
```




```
ALIGN Y (I, J)          ALIGN Y (I, J)          ALIGN Y (I, J)
with B (I-2, J+2)       with B (J+2, I-2)       with A (I)
```

Figure 1–13. Exemples d'alignements de tableaux sur des décompositions en FORTRAN D.




```
DISTRIBUTE              DISTRIBUTE              DISTRIBUTE
B (BLOCK, *)            B (*, BLOCK)            B (BLOCK, BLOCK)

DISTRIBUTE              DISTRIBUTE              DISTRIBUTE
B (CYCLIC, *)           B (*, CYCLIC)           B (BLOCK, CYCLIC)
```

Figure 1–14. Exemples de distributions d'une décomposition sur les processeurs.

Opérations de réductions

FORTRAN D fournit une série de fonctions de réduction. De plus, une instruction **REDUCTION** est introduite; nous expliquons pourquoi. Nous désirons écrire une fonction de réduction retournant la somme des éléments (exemple d'école, la fonction existe déjà en FORTRAN D)

```
REAL X (N), S
S = 0
FORALL I = 1, N
      S = S + X (I)
ENDDO
```

La sémantique de la boucle **FORALL** est telle que chaque itération de la boucle travaille sur une copie de la variable *S*. Toute réduction est donc impossible. On utilise donc l'instruction **REDUCTION** :

```
FORALL I = 1, N
      REDUCTION (MY_SUM, S, X (I))
ENDDO
```

La fonction *MYSUM* est une fonction du programmeur retournant la somme de ses arguments.

Localisation des traitements

FORTRAN D introduit une clause **ON**. Cette clause spécifie sur quels processeurs effectuer une itération de l'instruction **FORALL**. Trois formes existent :

A est une décomposition :

```
FORALL I = 1, N ON HOME (A(I))
```

X est un tableau :

```
FORALL I = 1, N ON HOME (X(I))
```

n$proc est une variable environnement retournant le nombre de processeurs de la machine cible; on précise un numéro de processeur à la clause **ON** :

```
FORALL I = 1, N ON MOD (I, n$proc) + 1
```

La proposition d'une telle clause est source d'optimisation dans les communications inter-PEs et le placement des traitements sur ces même PEs. Toutefois, une trop grande complexité dans l'expression associée à la clause **ON** pourrait mener à un effet inverse.

Discussion

FORTRAN D est un langage proposant de nombreuses et puissantes spécifications d'alignement et de distribution des éléments de tableaux sur un ensemble de processeurs. Ces constructions sont reprises au sein de *High Performance FORTRAN*, extension du langage FORTRAN 90 proposée pour les machines à parallélisme de données. Les machines ciblées par FORTRAN D ne se limitent pas aux machines SIMD mais recouvrent les machines MIMD telles l'Intel iPSC/860 pour laquelle un compilateur est en cours de développement [HKT91b].

Cependant les outils proposés sont parfois trop généraux et les premiers compilateurs FORTRAN D en limiteront la portée (les décompositions dynamiques ne seront, par exemple, pas traitées). Toutefois, la proposition de constructions puissantes menant ensuite à la recherche de mécanismes adéquats de compilation nous semble une voie intéressante.

1–4.7 Extensions du langage C pour la programmation des machines massivement parallèles : C*, MPL et POMPC

Nous nous intéressons dans cette section à trois langages dont les caractéristiques principales sont assez proches pour qu'ils puissent être présentés ensemble. Ces trois langages sont trois extensions de C et intègrent des constructions pour la programmation des machines massivement parallèles. Il s'agit de C*, MPL et POMPC :

- C* est une extension de C maintenant fournie sur les Connection Machine [TMC91b]. A l'origine C* fut défini par Rose et Steele [RoSt87], [Rose87] avec une sémantique quelque peu différente qui n'autorisait pas une projection efficace du langage sur la Connection Machine. Nous nous basons ici sur la nouvelle version de C*.

- MPL (MasPar Programming Language) est présenté comme une extension du langage C pour contrôler directement, dans un langage de haut-niveau, l'ACU (unité de contrôle des PEs et de traitement des scalaires) et les PEs de la MasPar. Le manuel de référence [Masp91a] et le guide de l'utilisateur [Masp91b] sont les principales sources d'informations sur MPL; le papier de Christy [Chri90] contient une introduction rapide à MPL.

- POMPC est une extension du langage C développée par Paris au sein du groupe POMP (Petit Ordinateur Massivement Parallèle) [Pari91], [HKMP91].

D'autres langages sont proches de ceux présentés ici. Citons le langage DBC (Data-parallel Bit-serial C) développé au Supercomputing Research Center [Schl91] et le langage multiC disponibles sur les machines Wavetracer [Wave91].

Classes de vecteurs

Les trois langages distinguent variables scalaires et vecteurs. Les variables scalaires sont allouées sur l'unité scalaire (frontal pour la Connection Machine, ACU pour la MasPar). Les variables vecteurs sont allouées sur les PEs à raison d'un élément par PE (ou PE virtuel). Ces variables vecteurs sont construites autour de la notion de **`shape`** en C*, de **`collection`** en POMPC, et de définition **`plural`** en MPL. Ces trois notions définissent les caractéristiques des vecteurs, à savoir le nombre de dimensions et la taille de chacune des dimensions.

En MPL, la taille d'une variable **`plural`** est définie par l'architecture de la machine; pour une machine 1024 processeurs, les variables **`plural`** sont formées de 1024 éléments; il n'existe donc qu'une classe de vecteurs en MPL : la classe **`plural`**.

Une **`shape`** C* ou une **`collection`** POMPC sont des classes de vecteurs. Les vecteurs d'une même **`shape`** ou **`collection`** sont de même taille et leurs éléments sont identiquement répartis sur les

processeurs; ces informations ne sont pas nécessairement définies lors de la compilation. La déclaration d'une classe définit éventuellement le nombre de dimensions et la taille de chacune (ces tailles doivent êter des puissances de 2 en C*) :

```
/* POMPC */                              /* C* */
collection [1152, 900] pixel;            shape [2048] [1024] pixel;
```

Suite à cette déclaration, on définit des vecteurs de la classe par

```
/* POMPC */          /* C* */              /* MPL */
pixel int image;     int:pixel image;      plural int image;
pixel double f;      double:pixel f;       plural double f;
```

En C* comme en POMPC, les caractéristiques de la classe peuvent n'être définies qu'à l'exécution par une fonction de la bibliothèque (**pc_start_collection** POMPC ou **allocate_shape** C*) :

```
/* C* */
shape pixel;
int dim [2];
...
dim [0] = 2048; dim [1] = 1024;
allocate_shape (&pixel, 2, dim);
...
deallocate_shape (&pixel);
```

On a la même construction en POMPC :

```
/* POMPC */
collection pixel;
int dim [2];
...
dim [0] = 1152; dim [1] = 900;
pc_start_collection (?pixel, dim [0] * dim [1], 2, dim, 0);
...
pc_free_collection (?pixel);
```

?pixel est une référence sur la **collection** *pixel*. L'opérateur **?** retourne aussi une référence sur la **collection** d'un opérande vecteur, les deux opérations suivantes produisent le même résultat :

```
/* POMPC */
?pixel
?image
```

Une particularité de POMPC est la possibilité, lors de la définition dynamique d'une **collection**, de préciser pour chaque dimension de la **collection** une préférence de distribution (dernier paramètre). Cette préférence est exprimée indépendamment de la machine cible sous forme de poids pour chaque dimension. Un poids fort indique une préférence à placer l'axe de la dimension sur un même PE virtuel. Les communications suivant cet axe seront donc internes aux PEs.

En POMPC, les opérations sur les collections se limitent à l'utilisation d'une collection lors de la déclaration de vecteur, l'allocation et la libération de collection (**pc_start_collection**, **pc_free_collection**), l'accès à la taille et la géométrie de la collection (**dimof**, **positionof**,

etc...), et la création de vecteur d'adresses de routage pour atteindre un vecteur d'une collection (**pc_built_address**).

C* fournit des pointeurs sur les **shape**. Ces pointeurs peuvent être déréférencés pour accéder à la **shape**. De tels pointeurs peuvent être passés en paramètre aux fonctions utilisateurs. L'opérateur **shapeof** retourne la **shape** du vecteur opérande gauche.

La taille des **shape** C* est définie par le programmeur. De plus, C* prédéfinit une **shape** à une dimension nommée **physical** dont le nombre d'éléments est égal au nombre de processeurs du système sur lequel le programme s'exécute. L'utilisation de cette **shape** apporte un gain significatif de performances.

Dans chacun des langages, un vecteur d'activité est associé à une classe; il désigne l'ensemble des éléments actifs de la classe. Les structures de contrôle vectorielles gèrent l'activité de ce vecteur (cf. infra).

Espace d'adressage et pointeurs

Les trois langages distinguent clairement deux espaces d'adressage : la mémoire scalaire et la mémoire des PEs. L'espace mémoire de chaque PE, est organisé de manière similaire; un emplacement mémoire contient pour tous les PEs une même variable. La gestion de cette mémoire est donc centralisée par le processeur scalaire; l'adresse d'une variable vecteur est un scalaire résidant sur ce processeur scalaire. C'est une conséquence du fonctionnement SIMD.

MPL et POMPC autorisent la définition de vecteurs de pointeurs. Ces vecteurs reflètent une possibilité d'adressage indirect dans la mémoire de chaque PE ou dans la mémoire du processeur scalaire depuis les PEs. C* ne fournit pas de tel mode d'adressage car l'architecture des processeurs de la Connection Machine ne supporte pas l'adressage indirect dans la mémoire des PEs. Exemples :

```
/* MPL */
plural int * AcuToPes;
int * plural PetoAcu;
plural int * plural PeToLocalPe;
```

La variable *AcuToPes* est un pointeur détenu par l'ACU qui référence une même adresse dans la mémoire de chacun des PEs. Chaque PE détient 1—un pointeur *PeToAcu* référençant une adresse sur l'ACU, et 2—un pointeur *PeToLocalPe* référençant une adresse dans la mémoire locale du PE. Notons bien qu'un pointeur **plural** ne peut référencer une adresse que dans la mémoire locale du PEs; la mémoire des autres PEs est atteinte par des primitives de communications (cf. infra). Ces caractéristiques de MPL sont partagées par POMPC; on écrit :

```
/* POMPC */
collection pixel;
pixel int * ScalToPes;
int * pixel PEtoScal;
pixel int (* pixel) PeToLocalPE;
```

Les trois langages fournissent la possibilité de définition de tableaux dans la mémoire des PEs. Chaque PE détient un tableau. Les éléments de ces tableaux ne pourront, bien entendu, pas être référencés en parallèle. Les vecteurs de tableaux POMPC et MPL sont comme en C représentés par un vecteur de pointeurs référençant le premier élément du tableau. A l'inverse, C* n'autorisant pas la définition de vecteurs de

pointeurs, les vecteurs de tableaux C* sont une simple zone mémoire sur chaque PE dont l'adresse est sur le processeur scalaire.

Une caractéristique propre à MPL est la possibilité de déclaration de variables **plural register**. Ces variables sont, si possible, allouées dans les registres des PEs; il est ainsi possible de contrôler finement les accès à la mémoire et aux registres des PEs. Cette caractéristique n'est pas reprise en C* car les PEs de la Connection Machine ne possèdent pas de registres.

Fonctions

Chacun des trois langages autorise le passage de paramètres vecteurs et le retour de valeurs vecteurs. En MPL et C*, les constructeurs (scalaire ou vecteur) des paramètres effectifs et formels doivent correspondre. En POMPC, un paramètre effectif scalaire peut être promu vecteur si le paramètre formel correspondant est vecteur.

Les langages POMPC et C* proposant différentes classes de vecteurs, des mécanismes sont mis en place pour obtenir une certaine généricité des fonctions vis-à-vis des classes de vecteurs. C* utilise un pointeur sur une **shape void** pouvant référencer n'importe quelle classe de vecteurs.

```
/* C* */
f (ptr)
int:void * ptr;
{
      with (shapeof (*ptr))
      ...
}
```

POMPC ne fixe pas la **collection** des paramètres formels; elle le sera pour chaque appel en fonction de la collection des paramètres effectifs. On peut écrire

```
/* POMPC */
collection pixel;
collection array;
collection line;

array int abs (array int n)
{
      where (n<0) return -n;
      elsewhere return n;
}

a () {
      pixel int p, q;
      line int a, b, c;
      q = abs (p);      /* abs () valable pour la collection pixel */
      a = abs (b);      /* et pour la collection line */
      c = abs (-1);     /* -1 est converti en vecteur de la collection
line */
}
```

La fonction *abs* est définie sur la **collection** *array*. La seule contrainte imposée est que le paramètre et la valeur retournée appartiennent à la même **collection**. La fonction *abs* est ensuite utilisable pour des

vecteurs de toute **collection** (par exemple *pixel* et *line*). Les conversions de type et promotion de scalaires en vecteurs de la **collection** sont réalisées.

Structures de contrôle SIMD

Le vecteur d'activité d'une classe de vecteurs est contrôlé par les extensions des structures de contrôle scalaires de C. Une telle structure définit un sous-ensemble de PEs dits actifs parmi l'ensemble des PEs. Les instructions contenues dans la structure de contrôle ne sont exécutées que sur les PEs actifs. Les aspects sémantiques de ces structures sont étudiés par Bougé et Levaire [BoLe91].

MPL et POMPC étendent toutes les structures de contrôle scalaires à des expressions vecteurs. C* propose uniquement une extension du **if** scalaire : le **where**. Le **if** vectoriel s'appelle aussi **where** en POMPC; le **while** vectoriel s'appelle **whilesomewhere**; le **for**, **forwhere**, etc... Les structures scalaires et vectorielles de MPL ont le même nom. Ce choix est malheureux. La non concordance de sémantique entre une même structure de contrôle ayant un opérande scalaire ou **plural** est gênante; il eut mieux fallu changer le nom des structures de contrôle **plural** comme en POMPC.

Si les structures des différents langages sont similaires, quelques différences subsistent entres les diverses extensions.

Une particularité de MPL est qu'un bloc d'instructions pour lequel aucun PE n'est actif n'est pas exécuté. Le code suivant est donc valide :

```
/* MPL */
plural int factorielle (n)
plural n;
{
      if (n <= 1) return 1;
      else return n * factorielle (n-1);
}
```

en particulier les appels récursifs de la fonction *factorielle* se terminent lorsque aucun PE n'est plus actif. A l'inverse, en POMPC les deux alternatives d'un **where** sont toujours exécutées, même si le vecteur d'activité est vide pour une des alternatives. Le code

```
array int factorielle (array int n)
{
      where (n <= 1) return 1;
      elsewhere return n * factorielle (n-1); /* erreur */
}
```

mène donc à une récursion infinie : l'appel à l'intérieur du bloc **elsewhere** est systématiquement exécuté; c'est au programmeur de se prémunir de tels cas de figure et d'inclure un test englobant le bloc d'instructions en question.

Un autre détail est le comportement, en fin de boucle, des PEs désactivés lors de l'itération. En MPL, un PE désactivé lors d'une itération d'une boucle **while plural** ne sera plus activé (il effectue un saut **break** sur la fin de la boucle). En POMPC, les éléments non actifs lors d'une itération de boucle **whilesomewhere** ne le sont pas systématiquement pour les itérations suivantes; une construction

```
/* POMPC */
whilesomewhere (expr)
```

est équivalente à

```
/* POMPC */
while (global_or (expr))
      where (expr)
```

Les instructions **break**, **continue** et **return** sont étendues aux structures vectorielles. Seuls les PEs actifs sont concernés par de telles instructions. Le flot d'instructions ne suit donc pas ces instructions; seul le vecteur d'activité est mis à jour.

En plus de l'extension des structures scalaires existantes de C, les langages intègrent tous un nouveau constructeur qui "réactive" tous les PEs; ce constructeur est nommé **all** en MPL, **everywhere** en C* et POMPC. Par exemple

```
/* MPL */
plural int active;
all {
      active = 0;
}
active= 1;
```

range le bit de contexte des PEs dans la variable *active*. Cette construction autorise la sauvegarde du vecteur d'activité courant, la mise en place d'un nouveau contexte (vecteur d'activité) et l'exécution d'instructions associées avant de revenir à l'ancien contexte.

Les communications inter-PEs et PEs/processeur scalaire

Deux niveaux de communications inter-PEs sont distingués sur les machines massivement parallèles visées par les trois langages étudiés ici. De plus, les communications entre le processeur scalaire et les PEs recouvrent la diffusion de valeurs depuis le processeur scalaire vers les PEs, les communications entre un PE donné et le processeur scalaire et les opérations de réduction des valeurs détenues par les PEs vers le processeur scalaire.

Les trois langages proposent une prise en charge de ces opérations de communication dans le langage via une syntaxe adaptée. De plus des fonctions de bibliothèque de base des langages fournissent les communications non disponibles au niveau de la syntaxe.

Différents opérateurs de MPL reflètent ces mécanismes de communications : **xnet**, **router**, **proc**, et **globalor**. EN C*, les communications sont réalisées via l'opérateur de description à gauche. Cet opérateur est aussi disponible en POMPC. Cependant POMPC fournit aussi un opérateur de plus bas niveau, l'opérateur de réarrangement **[. .]**, autorisant une gestion plus fine des communications. POMPC introduit aussi un opérateur de communication **<-**, alors que C* utilise l'opérateur d'affectation **=**.

Communications en MPL

Nous détaillons les constructions de MPL de gestion des communications. L'opérateur **globalor** est appliqué à une expression **plural**; il retourne la valeur produite par le OR-tree sur l'expression. La

diffusion d'une valeur détenue par l'ACU vers les PEs est réalisée implicitement par l'utilisation d'une valeur scalaire dans une expression **plural** :

```
/* MPL */
plural int pl;
int i;
pl = i;          /* diffusion de la valeur de i vers chacun des PEs */
```

Les communications entre les PEs sont basées sur une identification des PEs par un entier. Les variables MPL prédéfinies **nproc**, **nxproc**, et **nyproc** contiennent respectivement le nombre de PEs de la machine et les tailles horizontale et verticale de la grille de PEs. Les valeurs de celles-ci sont déterminées à l'exécution. Les fonctions MPL peuvent donc être produites (compilées) indépendamment de la configuration effective de la machine. Les PEs sont numérotés 0 à **nproc**; pour chaque PE, ce numéro est contenu dans la variable **plural** prédéfinie **iproc**. Les variables **plural** prédéfinies **ixproc** et **iyproc** contiennent, pour chaque PE, ses abscisse et ordonnée dans la grille des PEs.

Le constructeur **proc** désigne l'instance détenue par un PE d'une expression **plural**. Le PE est désigné par son numéro ou par ses abscisse et ordonnée. Le résultat de **proc** peut être une r-value (sa valeur est accédée) ou une l-value (une variable est affectée). Par exemple, la variable scalaire *val* récupère la valeur de *plus* du PE *i* :

```
/* MPL */
plural plus;
int i, val;
val = proc [i]. plus;
```

Autre exemple :

```
/* MPL */
plural int pl;
int i;

/* la boucle suivante est equivalente (mais moins efficace !) à
 *    pl = iproc;
 */
for (i = 0; i < nproc, i++)
      proc [i]. pl = i;
```

Les constructeurs **xnet** sont utilisés pour accéder la mémoire des PEs dans une direction et à une distance identique pour tous les PEs via des communications de voisinage. Huit constructeurs correspondent aux 8 directions N, NE, E, SE, S, SW, W, et NW. La grille des PEs est torique; les bords nord/sud et est/ouest sont reliés. Le code suivant réalise un "lissage" de la valeur de *tonalite* par les valeurs de ses 9 voisins :

```
/* MPL */
plural real tonalite;
tonalite += xnetW [1]. tonalite + xnetE [1]. tonalite;
tonalite += xnetN [1]. tonalite + xnetS [1]. tonalite;
tonalite /= 9.0;
```

Quelques précisions sont à donner pour l'évaluation des différentes parties d'une telle instruction de communication :

```
/* MPL */
plural int tab [2], result;
int i = 3;
tab [0] = 0; tab [1] = 1;
result = xnetE [i]. tab [iproc%2]
```

Intéressons-nous à la valeur reçue par le PE numéro 0. Cette valeur est obtenue depuis le PE numéro 3. La valeur obtenue est celle de l'expression *tab [iproc%2]* sur le PE 3, soit 1. Notons bien que cette évaluation a lieu sur le PE 3.

Le constructeur **router** assure les communications globales entre les PEs. Une expression **plural** indique, pour chaque PE, le PE avec lequel communiquer :

```
/* MPL */
plural int i, j, k;
/* envoie la valeur de k au PE i et la range dans j : */
router [i]. j = k;
/* charge dans k la valeur de j détenue par le PE i : */
k = router [i]. j;
```

Supposons sur cette dernière instruction que la variable *i* contienne dans deux PEs différents une même valeur (mettons ι). Les deux PEs détenant la valeur ι accèdent à la valeur de *j* sur le PE ι. L'évaluation de *j* étant réalisée sur ce PE, deux PEs différents ne peuvent accéder deux valeurs différentes sur le PE ι. En résumé, aucune adresse ne transite entre les PEs lors de communications exprimées par les constructions par **xnet** ou **router**.

MPL dispose d'une bibliothèque de fonctions assurant de plus amples fonctionnalités de communications : **rfetch**, **rsend**, **xfetch**, et **xsend**. Ces fonctions autorisent le transit de l'adresse, dans la mémoire du PE, où écrite (resp. lire) les données transférées (resp. à transférer). Par exemple les arguments de la fonction **ps_rfetch** sont :

- Le numéro du PE d'où charger une valeur;
- un pointeur **plural char * plural** référençant l'adresse, dans le PE distant, à laquelle obtenir la valeur;
- un pointeur dans la mémoire locale où ranger la valeur obtenue; et
- la taille de la donnée à transférer.

Ces arguments sont bien évidement (pour une fonction) évalués localement; différents PEs peuvent donc obtenir une donnée différente (valeur locale du pointeur **plural char * plural** différente) sur un même PE (numéro du PE identique). L'opérateur **router** n'est qu'une forme simple des fonctions **rfetch** et **fsend**.

De plus une bibliothèque de fonctions gère les échanges de données entre le DPU (ACU + PEs) et le frontal ainsi que le déclenchement (synchrone et asynchrone) de tâches entre ces deux entités.

Communications en POMPC

POMPC introduit un opérateur <- de communication entre une partie droite et gauche. Ces parties droite et gauche sont obtenues par les opérateurs de réarrangement. L'opérateur de réarrangement **[. .]** retourne un vecteur dont on précise (par un vecteur d'adresses) l'adresse de chaque élément. Par exemple

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

Le vecteur de **double** résultant appartient à la collection *line*; son $i^{ème}$ élément est l'élément j de *vector*, j étant la valeur de l'élément i de *address*. Les vecteurs d'adresses sont obtenus par la fonction **pc_built_adress**; en effet ces adresses dépendent des caractéristiques (nombre de dimensions et taille des dimensions) de la collection du vecteur décrit.

L'opérateur de réarrangement à gauche est une abréviation laissant à la charge du compilateur la construction du vecteur d'adresses. On peut écrire

```
/* POMPC */
... [x, y] vector ...
```

pour

```
/* POMPC */
addr = pc_built_adresses (?vector, x, y)
... vector [. addr .] ...
```

Fournir un opérateur de réarrangement et une fonction de production des adresses au niveau du langage autorise, par exemple, un calcul unique des adresses pour plusieurs instructions de communication.

Les communications de voisinages sont exprimées à l'aide de références aux coordonnées courantes des éléments suivant les différentes dimensions de la collection (fonction **pc_coord**). En deux dimensions par exemple, le voisin de droite est

```
/* POMPC */
collection [1152, 900]pixel;
pixel int image
... [pc_coord (0)+1, pc_coord (1)] image ...
```

pouvant être abrévié en `[ .+1, . ]`, le . représentant la coordonnée courante dans la dimension.

D'autres caractéristiques sont propres à POMPC. Pour chaque axe de communication (dimension de la collection), on précise s'il y a rebouclage torique ou non par la fonction **pc_set_torus**. Si une opération de communications est susceptible de produire plusieurs données pour un élément, il est possible de préciser un opérateur d'accumulation des données pour produire le résultat (addition, multiplication, et, ou, etc.).

Exemple, lissage des valeurs par celles des 26 voisins dans une collection 3 dimensions toriques :

```
/* POMPC */
collection [16, 16, 16] volume;
volume real r;
pc_set_torus (?volume, all_dim)
r += (<- [ .+1, ., . ] r) + (<- [ .-1, ., . ] r);
r += (<- [ ., .+1, . ] r) + (<- [ ., .-1, . ] r);
r += (<- [ ., ., .+1 ] r) + (<- [ ., ., .-1 ] r);
r /= 27;
```

*Communications en C**

Les opérations de réduction de données parallèles sont obtenues par les opérations ***op***= dont l'opérande gauche est scalaire et l'opérande droit vecteur. Par exemple, on obtient la somme des éléments du vecteur *vect* dans la variable *sum* par

```
/* C* */
shape courante;
double:courante vect;
double sum = 0;
sum += vect;
```

C* ne fournit qu'une construction pour la gestion des communications inter-PEs et des communications entre un PE donné et le processeur scalaire : l'opérateur de description à gauche. On écrit par exemple

```
/* C* */
shape courante
int:courante index0, index1;
double:courante source, dest;

[index0] [index1] dest = source;        /* instruction 'send' */
dest = [index0] [index1] source;        /* instruction 'get' */
```

Pour un PE donné, les vecteurs d'entiers *index0* et *index1* contiennent les coordonnées dans la dimension du PE où envoyer/lire la valeur du PE. On est donc en présence d'une communication générale entre PEs.

Les communications aux voisins utilisent la fonction prédéfinie **pcoord**. Cette fonction retourne, pour chaque PE, sa coordonnée sur l'axe de la dimension passée en paramètre. Par exemple, le code suivant incrémente la valeur locale par la valeur du voisin à droite dans la seconde dimension :

```
/* C* */
shape courante
double:courante vect;

vect += [pcoord (0)] [pcoord (1) + 1] vect;
```

L'expression **pcoord (*axe_courant*)** pouvant être abréviée par un point, on écrit plutôt

```
/* C* */
vect += [.] [. + 1] vect;
```

On détermine si les axes de communication sont toriques ou non par l'utilisation de modulo (opérateur **%%**) dans le calcul des coordonnés et la fonction **dimof** qui retourne la taille d'une dimension donnée. L'exemple précédent avec rebouclage torique sur la seconde dimension s'écrit :

```
/* C* */
shape courante
double:courante vect;

vect += [.] [(. + 1) %% dimof (courante, 1)] vect;
```

L'adressage d'un processeur donné est réalisé par une opération de description à gauche dont tous les opérandes sont scalaires.

```
/* C* */
shape [2048] [1024] courante
double:courante vect;
double r, s;

[0] [0] vect = r;
s = [2047] [1023] vect;
```

initialise la valeur du premier élément du vecteur *vect* avec la valeur de *r* détenue par le processeur scalaire et retourne la valeur du dernier élément de *vect* dans la variable scalaire *s*.

Contrôle de l'activité des PEs lors de la communication

Comme toutes les instructions des langages C*, MPL et POMPC, les instructions de communications ne sont effectives que sur l'ensemble des PEs actifs. Un PE non actif n'émet pas de valeur lors d'une instruction d'émission (send), mais un autre PE peut lui envoyer une donnée et ainsi changer des valeurs détenues localement. De même, un PE inactif ne reçoit pas de valeur lors d'une instruction de réception (get). Toutefois les valeurs qu'il détient peuvent être lues par d'autre PEs.

Discussion

Les trois langages présentés sont des langages de relativement bas niveau. Cependant, à l'évidence, MPL est un langage de plus bas niveau, reflétant les caractéristiques et particularités de la machine MasPar : taille des variables vecteurs identique au nombre de PEs, variables parallèles `register`, rebouclage torique des deux axes de parallélisme. L'avantage d'un tel langage est bien entendu le contrôle fin de la machine qui peut être obtenu par un programmeur averti et d'une part les performances qui s'en suivent, d'autre part, la simplicité relative du compilateur. Quelques points semblent discutables. Par exemple, l'absence de virtualité en MPL est le contre-coup du contrôle fin de la machine; le traitement d'un problème de taille donnée (différente du nombre de PEs) est laissé à l'utilisateur.

De même, mais dans une moindre mesure, certaines limitations de la Connection Machine, par exemple l'absence d'adressage indirect dans les processeurs élémentaires se retrouve en C* (absence de vecteurs de pointeurs et de "vrais" tableaux C). A l'inverse les opérations de communications sont très proches des opérations de description habituelles des langages vectoriels et facilitent ainsi l'apprentissage du langage.

POMPC est un langage défini indépendamment de la machine massivement parallèle visée. C'est certainement là son principal atout. Des projections sur Connection Machine, MasPar et autres machines SIMD sont possibles. Un point intéressant de POMPC est la possibilité pour le programmeur de spécifier une préférence pour la projection des axes des `collection` sur un même PE ou sur les différents axes du réseau de communications inter-PEs et ce, indépendamment de la machine cible.

Ces trois langages sont à la frontière entre les langages de haut niveaux et les langages intermédiaires. Nicolas Paris énonce cela en présentant POMPC comme un macro-assembleur pour les machines SIMD comme le langage C peut l'être pour les machines traditionnelles [HKMP91].

1–4.8 Un système de programmation pour machines massivement parallèles : PARALLAXIS

PARALLAXIS est un langage destiné à implanter les algorithmes massivement parallèles. PARALLAXIS est développé par Bräunl et son équipe de l'université de Stuttgart [Bräu89], [BBES91]. PARALLAXIS

est une extension de MODULA-2 autorisant la manipulation explicite du parallélisme et la définition d'une architecture adaptée à l'algorithme. Cette définition est exprimée sous la forme de connexions (mot clé **CONNECTION**) entre un ensemble de processeurs élémentaires (mot clé **CONFIGURATION**). Par exemple le programmeur peut définir

```
CONFIGURATION anneau [12];
CONNECTION horaire : anneau [i] -> anneau [(i+1) mod 12] . antihoraire
           antihoraire : anneau [i] -> anneau [(i-1) mod 12] . horaire
```

La configuration est formée de 12 PEs. Un PE possède deux ports *horaire* et *antihoraire* correspondants à deux fonctions de communication. Le port *horaire* du PE *i* est connecté au port *antihoraire* du PE *(i+1) mod 12*. Les PEs sont ainsi connectés en anneau (figure 1–15). Une fois définie, une configuration est utilisée pour un système incluant des définitions de vecteurs et des instructions de manipulation de ces vecteurs. Les objets vecteurs sont définis comme possédant un élément dans chaque PE de la configuration.

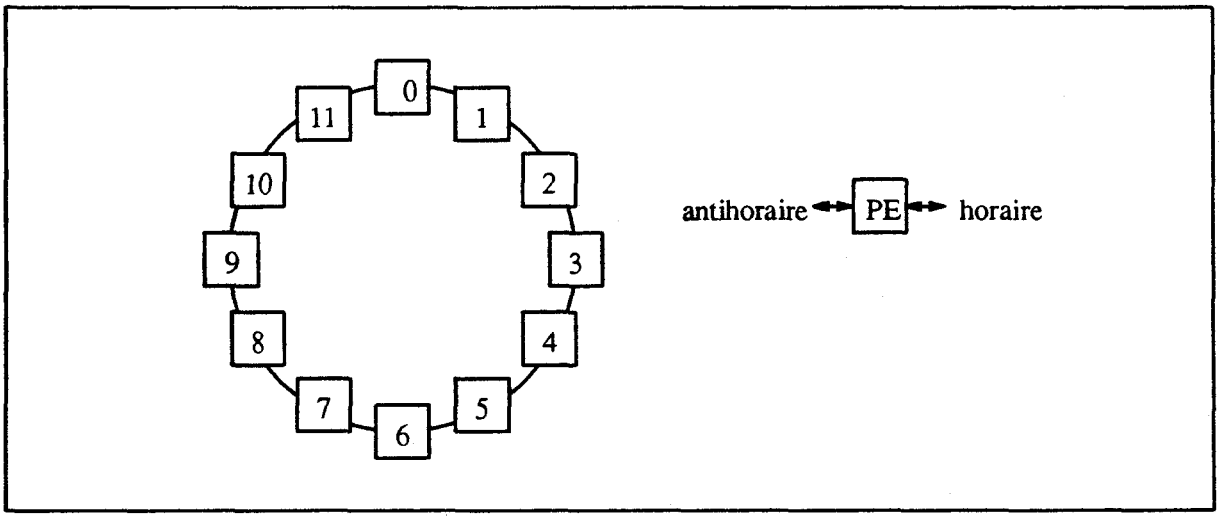


Figure 1–15. Connexion des PEs en anneau.

L'ensemble des PEs actifs est déterminé par un bloc **PARALLEL/ENDPARALLEL** spécifiant des sections de numéros de PEs actifs ou par des constructions **IF**, et **WHILE** déterminant les PEs actifs en fonction de la valeur locale d'une expression vectorielle.

```
VECTOR a, b : REAL;
PARALLEL [3 .. 7]
      ...
ENDPARALLEL

PARALLEL
      IF a > b THEN
            ...
      END
ENDPARALLEL
```

Le **IF** réduit l'ensemble des PEs actifs à ceux dont la valeur locale de $a < b$ est vraie.

Les instructions de propagation utilisent les connexions entre PEs pour communiquer des valeurs :

```
PROPAGATE. horaire (a)
```

émet la valeur locale de *a* sur le port *horaire*. Cette valeur sera rangée par le PE destinataire à l'emplacement de *a*. Notons le comportement des PEs inactifs lors d'une opération **PROPAGATE**. Les valeurs à destination de PEs inactifs sont perdues, aucune valeur n'est reçue depuis un PE inactif. Il est possible de ne pas écraser la valeur en indiquant une destination différente :

```
PROPAGATE. horaire (a+1, b)
```

ou de communiquer avec un PE situé à une distance plus éloignée suivant une direction :

```
PROPAGATE. horaire ^ 5 (a)
```

PARALLAXIS offre aussi des communications dites de réduction

```
SCALAR s : REAL;
VECTOR x : REAL;
s := REDUCTION. SUM (x)
```

affecte la somme des composantes de *x* détenues dans des PEs actifs à *s*. Ces réductions sont réalisées sans tenir compte de la topologie définie pour le système.

L'approche de PARALLAXIS est originale : le programmeur adapte une architecture virtuelle à son problème. Le point noir de cette spécification est bien entendu la projection de cette architecture virtuelle sur la machine réelle. Les communications qui apparaissent comme locales au sein d'un système PARALLAXIS peuvent ne pas être générées ainsi mais devoir entraîner des communications globales sur la machine cible. Le programmeur n'a pas le contrôle du coût des communications.

1–4.9 PARIS

PARIS (PARallel Instruction Set) est un jeu de primitives manipulant des structures de données disposées sur les processeurs élémentaires de la Connection Machine [TMC89]. Ces primitives sont incluses sous forme d'appels à des fonctions de bibliothèques dans un source FORTRAN, C ou LISP. L'exécution d'une primitive PARIS au niveau de la machine hôte déclenche l'envoi de code binaire au séquenceur qui le convertit en une séquence de nano-instructions diffusées à tous les PEs.

Le code PARIS est soit directement inclus par le programmeur dans son code source ou généré par un compilateur à partir d'un langage de haut niveau tel le CM FORTRAN. (C'était le cas pour les versions antérieures à la version 1.0 du compilateur CM FORTRAN).

PARIS présente la Connection Machine comme une machine virtuelle dont le nombre de PEs n'est pas limité. Ces processeurs virtuels sont ensuite alloués sur les processeurs physiques de la machine, chaque PE physique étant divisé en autant de PEs virtuels que nécessaire (le ratio nombre de PEs virtuel sur nombre de PEs physiques est nommé *VP ratio*). Cette abstraction rend les programmes PARIS indépendants du nombre de processeurs physiques existants sur une configuration donnée.

La manipulation de données parallèles est réalisée par l'association d'un processeur virtuel (ou *VP*) à chaque élément de la donnée. Des ensembles de VP (ou *VP sets*) sont créés et détruits par des appels à des routines PARIS (par exemple **allocate_vp_set**). Cette création précise la géométrie (nombre de

dimension) des VPs. Suite à la création d'un VP set, on lui alloue de la mémoire sous la forme d'un *field* (ou champ). Un champ correspond à une variable ayant une instance sur chaque VP d'un VP set. Un champ est un emplacement mémoire alloué sur chaque VP; il est référencé en PARIS par un *field_id* (deux allocations sont distinguées : l'allocation sur une pile et l'allocation sur un tas; les zones allouées sur la pile devant être libérées dans l'ordre inverse de leur allocation). Exemple (en C/PARIS) :

```
CM_geometry_id_t ma_geometrie;
CM_vp_set_id_t mon_vp_set;
int mes_dimemsions [2];
CM_filed_id_t ma_var1, ma_var2, ma_var3;

/* creation d'une geometrie deux dimensions 1152 * 900 */
mes_dimensions [0] = 1152; mes_dimensions [1] = 900;
ma_geometrie = CM_create_geometry (mes_dimensions, 2);

/* creation d'un vp_set suivant cette geometrie */
mon_vp_set = CM_allocate_vp_set (ma_geometrie);

/* allocation de trois variables d'entiers dans ce vp_set */
ma_var1 = CM_allocate_heap_field (CM_type_len (int));
ma_var2 = CM_allocate_heap_field (CM_type_len (int));
ma_var3 = CM_allocate_heap_field (CM_type_len (int));
```

Les noms préfixés par `CM_` dénotent des constructions (types, fonctions, constantes, etc...) de PARIS. Les variables utilisateurs sont préfixées par `mon_`, `ma_` ou `mes_` dans cet exemple.

L'exécution d'instructions n'est possible que sur un VP set courant; le VP set courant est défini par

```
CM_set_vp_set (mon_vp_set);
```

Pour ce VP set courant, il est possible de réaliser des opérations entre les variables/champs :

```
/* initialisation de ma_var1 a 5 et ma_var2 a zero */
CM_s_move_const_always_1L (ma_var1, 5, CM_type_len (int));
CM_s_move_zero_always_1L (ma_var2, CM_type_len (int));
/* ma_var3 = ma_var1 + ma_var2 */
CM_s_add_always_3_1L (ma_var3, ma_var2, ma_var1, CM_type_len (int));
```

On détermine un sous ensemble du VP set courant de VP actifs par une condition; les instructions suivantes ne sont réalisées que sur ces VP actifs :

```
/* Positionne le flag actif pour les VP tels que ma_var1 > ma_var2 */
CM_s_gt_1L (ma_var1, ma_var_2, CM_type_len (int));
/* ma_var3 = ma_var1 + ma_var2 pour les VP actifs */
CM_s_add_3_1L (ma_var3, ma_var2, ma_var1, CM_type_len (int));
/* retour a tous les VP actifs */
CM_set_context ();
```

Des instructions de communications aux voisins sont par exemple :

```
/* ma_var3 += ma_var2 [Nord], c'est a dire la valeur de ma_var2 vers
le haut, dans la dimension 0 */
CM_get_from_news_1L (ma_var3, ma_var_2, 0, CM_upward, CM_type_len
(int));
CM_s_add_2_1L (ma_varl, ma_var2, CM_type_len (int));
```

Ces primitives PARIS sont incluses dans un programme C (ou LISP ou FORTRAN) :

```
f () {
      CM_filed_id_t ma_varl, ma_var2;
      int i;
      for (i = 0; i <= 1000; i++) {
            CM_s_gt_1L (ma_varl, ma_var_2, CM_type_len (int));
            if (test ())
                  CM_s_add_2_1L (ma_varl, ma_var2, CM_type_len (int));
      }
}
```

Il est clair que PARIS est un langage de bas niveau et que seules des considérations d'efficacité peuvent en motiver l'utilisation. Celle-ci doit être réduite aux portions de codes dont on désire contrôler finement la gestion des opérations parallèles. Il apparaît par ailleurs évident que les instructions PARIS reflètent fidèlement l'architecture de la Connection Machine. Cependant, la disponibilité de registres sur les processeurs flottants de la Connection Machine ne peut être prise en compte au niveau de PARIS. Se reporter à la section 1–5.4 pour une critique du modèle d'abstraction de la Connection Machine fourni par PARIS.

1–4.10 Résumé

Nous avons donné un large éventail des langages utilisés pour la programmation à parallélisme de données. De nombreuses caractéristiques communes ont émergé de ce tour d'horizon. Nous tentons maintenant de résumer les particularités de chacun des langages. Pour ce faire, nous avons dressé une liste de caractéristiques et donnons la position de chacun des langages vis-à-vis de ces caractéristiques.

Ces caractéristiques sont : Le nom du langage, une date, le langage de base duquel il est dérivé, les machines ciblées par le langage (machines pipelines vectorielles, machines tableaux, machines massivement parallèles, ou une machine particulière), la forme des objets vecteurs manipulés, leur virtualité (taille, nombre de dimensions, placement sur la machine) et leur dynamicité, les caractéristiques de l'allocation de ces vecteurs, les opérations de descriptions existantes et/ou les opérations de communication disponibles, les structures de contrôles parallèles proposées, les caractéristiques des fonctions et des passages de vecteur en paramètre.

Ces listes sont en annexe B.

1–5 Implantation des langages à parallélisme de données

Nous discutons maintenant brièvement des techniques mises en place pour la projection des langages à parallélisme de données sur les machines cibles. Après un bref descriptif des outils de vectorisation

automatique, nous développerons quelques points centraux dont la prise en charge par les compilateurs est essentielle pour une génération de code performant.

1–5.1 Les vectoriseurs automatiques

Nos travaux sont axés sur la programmation vectorielle explicite. Dans ce cadre, l'utilisation d'outils de découverte automatique du parallélisme de données est inutile, ce parallélisme étant exprimé directement par le programmeur. Nous donnons cependant ici la structure générale d'un vectoriseur; en effet de nombreux travaux dans ce domaine ont une relation directe avec les recherches en matière de compilation des langages vectoriels explicites.

Les vectoriseurs sont les outils de détection du parallélisme utilisés dans un système de développement de programmes vectoriels dont le langage utilisé est un langage scalaire. Ces outils sont la première phase de la compilation, ils sont chargés de découvrir le parallélisme dans les sources écrits en langage scalaire. Typiquement, un vectoriseur détecte si une boucle scalaire correspond à une instruction vectorielle. Une analyse des dépendances entre données est nécessaire pour déterminer si l'ordre imposé dans l'écriture scalaire peut ne pas être respecté; la boucle est alors dite vectorisable.

Dans les systèmes proposés par les constructeurs, la détection des dépendances est évitée et la vectorisation facilitée lors de l'utilisation des *directives de compilation*. Ces directives sont des indications fournies par le programmeur pour signifier que telle boucle est vectorisable. Elles sont en général incluses dans des commentaires et sont donc ignorées par les compilateurs à qui elles ne sont pas destinées. En CFT77 (langage FORTRAN des machines Cray), la directive `IVDEP` indique que la boucle qui suit est vectorisable :

```
*     N est toujours positif ou nul
CDIR$ IVDEP
      DO 100 I = LOW, UP
 100  V (I) = V (I+N) + W (I)
```

De nombreuses transformations sont réalisées pour augmenter le nombre de boucles vectorisables. Ces transformations sont basées sur

- la recherche d'informations sur les valeurs limites des variables et des expressions (propagation des valeurs et analyse des expressions dont les valeurs ne sont pas connues à la compilation) qui peuvent déterminer si une boucle est vectorisable, et
- la transformation de boucles amenant l'élimination de dépendances.

Le livre de Banerjee [Bane88b] est une synthèse des travaux réalisés en matière d'analyse des dépendances; un aperçu de ce livre peut être trouvé dans [Bann88a]. L'article de Padua et Wolfe [PaWo86] décrit de nombreuses optimisations mises en œuvre lors de la vectorisation et de la parallélisation de boucles. Le livre de Wolfe [Wolf89] est une reprise de sa thèse; il y développe les transformations de boucles intéressantes pour les traitements vectoriel et parallèle. On trouvera d'autres transformations de boucles dans [Wolf91]. Le livre de Zima [ZiCh90] est un état de l'art des systèmes de compilation de programmes scalaires pour machines vectorielles.

1–5.2 Découpage d'une instruction vectorielle en boucle

Nous discutons ici d'une technique classique autorisant le traitement de vecteurs de longueur quelconque sur une machine traitant des vecteurs de longueur fixe. Typiquement cette longueur est la longueur des registres d'une machine vectorielle pipeline, ou le nombre de processeurs élémentaires d'une machine tableau. Cette technique est habituellement désignée par le terme "strip-mining". Elle consiste à découper un vecteur de taille quelconque en tranches (les strips) pouvant être traitées par la machine. Une instruction vectorielle

```
A (1:N) = 2 * B (1:N)
```

est réécrite pour une machine traitant des vecteurs de taille VRS (Vector Register Size), VL (Vector Length) étant la taille effective de traitement des vecteurs

```
DO I = 1, N, VRS
      VL = MIN (VRS, N - (I + 1))
      A (I:I+VL-1) = 2 * B (I:I+VL-1)
ENDDO
```

Un tel découpage n'est pas possible en cas de dépendance entre les parties droite et gauche de l'affectation. On doit alors gérer une seconde boucle de passage par un temporaire (*TEMP*) :

```
DO I = 1, N, VRS
      VL = MIN (VRS, N - (I + 1))
      TEMP (I:I+VL-1) = 2 * B (I:I+VL-1)
ENDDO
DO I = 1, N, VRS
      VL = MIN (VRS, N - (I + 1))
      A (I:I+VL-1) = TEMP (I:I+VL-1)
ENDDO
```

Le surcoût engendré par le passage par un temporaire est double : il nécessite une allocation en mémoire de *TEMP* et oblige de nombreuses opérations de lecture/écriture mémoire supplémentaires.

1–5.3 Fusion de boucles

Considérons maintenant la génération de code pour deux instructions vectorielles (dont nous supposons pour la simplicité de l'exposé les parties gauche et droite libres de toute dépendance).

```
A (1:N) = 2 * B (1:N)
C (1:N) = 5 + D (1:N)
```

En appliquant le découpage des instructions en boucle, on obtient

```
DO I = 1, N, VRS
      VL = MIN (VRS, N - (I + 1))
      A (I:I+VL-1) = 2 * B (I:I+VL-1)
ENDDO
DO I = 1, N, VRS
      VL = MIN (VRS, N - (I + 1))
      C (I:I+VL-1) = 5 + D (I:I+VL-1)
ENDDO
```

Une transformation intéressante est d'appliquer une fusion des deux boucles; on obtient :

```
DO I = 1, N, VRS
      VL = MIN (VRS, N - (I + 1))
      A (I:I+VL-1) = 2 * B (I:I+VL-1)
      C (I:I+VL-1) = 5 + D (I:I+VL-1)
ENDDO
```

Pour réaliser la fusion, il est nécessaire que les deux instructions vectorielles traitent des vecteurs de même taille (*N*). De plus cette fusion des boucles n'est possible que sous des conditions de non dépendance entre les deux instructions vectorielles (se référer à [Wolf91] pour plus de précision).

Le gain obtenu par la fusion de deux boucles est important si l'on considère le parallélisme, dans l'activité des unités fonctionnelles d'une machine vectorielle pipeline, pouvant être engendré par la réalisation de deux instructions au lieu d'une au sein d'une itération. De plus, certains cas favorables diminuent le nombre de lectures mémoire et augmentent l'utilisation des registres vectoriels :

```
A (1:N) = 2 * B (1:N)
C (1:N) = 5 + A (1:N)
```

Au sein d'une itération de la boucle, la tranche de *A* est produite dans un registre; ce registre est utilisé par l'instruction suivante. Ceci n'est pas possible si les deux boucles ne sont pas fusionnées. De nombreux codes correspondent à ce cas de figure, par exemple les évaluations partielles des expressions vectorielles.

```
A (1:N) = 2 * B (1:N) + C (1:N)
```

est généré

```
DO I = 1, N, VRS
      VL = MIN (VRS, N - (I + 1))
      TEMP (I:I+VL-1) = 2 * B (I:I+VL-1)
ENDDO
DO I = 1, N, VRS
      VL = MIN (VRS, N - (I + 1))
      A (I:I+VL-1) = TEMP (I:I+VL-1) + C (I:I+VL-1)
ENDDO
```

qui est transformé par fusion des boucles (en l'abscence de dépendance)

```
DO I = 1, N, VRS
      VL = MIN (VRS, N - (I + 1))
      TEMP (I:I+VL-1) = 2 * B (I:I+VL-1)
      A (I:I+VL-1) = TEMP (I:I+VL-1) + C (I:I+VL-1)
ENDDO
```

De plus le temporaire *TEMP* n'a pas à être alloué en mémoire, un registre vectoriel *(VR1)* suffit :

```
DO I = 1, N, VRS
      VL = MIN (VRS, N - (I + 1))
      VR1 = 2 * B (I:I+VL-1)
      A (I:I+VL-1) = VR1 + C (I:I+VL-1)
ENDDO
```

Les deux techniques de découpage des instructions vectorielles en boucles et de fusion de boucles ont jusqu'à maintenant surtout été utilisées au sein des systèmes de production automatique de code vectoriel à partir de langages scalaires. Ces méthodes doivent être prises en compte par les compilateurs de langages explicites pour atteindre un niveau de performances acceptable.

1–5.4 Virtualisation du nombre de processeurs

La programmation en langage PARIS est réalisée sur une machine virtuelle possédant un nombre illimité de processeurs élémentaires (section 1–4.9). Le concept de VP ratio (VPR) est sous-jacent à la projection de cette architecture virtuelle sur la Connection Machine. VPR processeurs virtuels sont alloués sur un processeur physique de la Connection Machine; par exemple une variable vecteur *A* est instanciée VPR fois sur chaque PE physique (nous référençons ces VPR variables par *A(i)* pour $i \in \{1 .. VPR\}$). Un traitement sur les PEs virtuels est réalisé par une boucle de VPR traitements sur les PEs physiques. Par exemple l'instruction PARIS correspondant à

```
A = 2 * B
```

est réalisée

```
DO I = 1, VPR
      A (I) = 2 * B (I)
ENDDO
```

La génération de cette boucle à partir de l'instruction PARIS est réalisée de manière statique lors de l'écriture de la bibliothèque PARIS. C'est ce code qui sera exécuté pour la réalisation de l'instruction.

Une séquence de deux instructions vectorielles ne peut donc pas être fusionnée. Cela amène un surcoût d'autant plus important que les optimisations telles l'utilisation de registres pour stocker les valeurs temporaires ne peuvent plus être mises en place.

Ces raisons simples de limitation des capacités de la Connection Machine ont amené l'abandon progressif du modèle PARIS pour une compilation dans le style de celle réalisée sur la MasPar, dont les techniques sont proches de celles des compilateurs des machines vectoriels pipelines (pour ces aspects du moins). (Ces modèles de compilation sont désignés par le vocable "slice-wise" en raison du rangement des données dans la mémoire de PEs. Une Connection Machine, de 64 K PEs par exemple, est alors considérée comme une machine construite autour de 2048 "nœuds". Chaque nœud est composé d'un PE (flottant comprenant des registres), d'une mémoire, d'un mécanisme de routage et de 32 PEs un bit.)

1–6 Conclusion

Dans ce chapitre, nous avons donné un aperçu des machines à parallélisme de données et nous nous sommes particulièrement arrêtés sur leurs caractéristiques devant être prises en compte par les langages et/ou les compilateurs. Nous nous sommes ensuite intéressés aux langages proposés pour la programmation à parallélisme de données; nous rappelons maintenant quelques points importants :

- Il a été montré que les deux grandes familles de machines à parallélisme de données (à savoir les machines vectorielles pipelines d'une part, les machines tableaux et massivement parallèles d'autre part) possèdent de nombreuses similitudes.

- Nous avons fait le tour des propositions de syntaxes pour la programmation à parallélisme de données. De nombreuses constructions sont devenues de fait habituelles, sinon naturelles.

- Nous avons noté les propositions d'intégration dans les langages d'outils laissant le programmeur exprimer les connaissances qui lui sont propres sur son programme. Cette expression a pour but de faciliter, sinon autoriser, la génération de code efficace.

- Cependant, certains problèmes rencontrés lors de la compilation de ces langages ne peuvent être résolus par les seules informations fournies dans la plupart des langages.

Nous possédons maintenant une liste de considérations qui ne peuvent être écartées lors de la conception d'un langage et/ou d'un compilateur pour la programmation à parallélisme de données.

Chapitre 2

Le langage EVA

Nos travaux de conception d'un langage vectoriel ont été entrepris alors que la nouvelle norme FORTRAN (FORTRAN 8x à l'époque) était en cours d'élaboration. Entre autres modifications de la norme FORTRAN 77, la nouvelle norme de FORTRAN développait des concepts de vecteurs et d'opérations vectorielles qui nous semblaient limités. Nos travaux sur EVA sont donc des propositions d'outils plus performants pour la programmation vectorielle. Cet aspect de performance est double. D'une part, nous désirons fournir au programmeur des outils de haut niveau pour la programmation explicite de ses problèmes. D'autre part, nous souhaitons mettre en place des structures de haut niveau qui aideraient la production de code efficace sur une machine vectorielle. Ce chapitre est donc divisé en deux parties. Une première partie présente le langage EVA et développe les apports de EVA au niveau de la programmation vectorielle. Une seconde partie traite des constructions de haut niveau que nous avons mises en place et de leur implantation sur machine vectorielle pipeline via le langage intermédiaire vectoriel DEVIL. Bien des aspects du langage EVA peuvent être considérés à la fois comme facilitant la programmation et la compilation. On ne s'étonnera donc pas de quelques répétitions entre les deux parties de ce chapitre.

D'autres publications et rapports sont relatifs au langage EVA. L'article de Dekeyser, Marquet et Preux [DMP90a] est une première présentation du langage. Les rapports [Marq90] et [Marq91] contiennent respectivement la grammaire et le manuel du langage EVA. DEVIL, le langage intermédiaire utilisé par EVA, est présenté dans [DMP90b]. Le manuel de DEVIL et la présentation de MAD, la machine virtuelle de DEVIL, peuvent être trouvés dans [Preu90]. D'autres informations relatives à DEVIL sont disponibles dans la thèse de Philippe Preux [Preu91].

Chapitre 2 — Section 1

La programmation en EVA

Cette section présente les constructions de haut niveau que le langage EVA propose au programmeur afin de faciliter la programmation vectorielle. Nous discutons les choix qui ont été faits lors de la conception de EVA. Des travaux ont été réalisés pour répondre à la question suivante : *Qu'attend un programmeur scientifique d'un langage vectoriel ?* Citons le rapport que fait Wetherell d'une enquête menée auprès de scientifiques du *Department of Energy* (DoE) [Weth80]. Citons la réponse que lui font Perrott et Stevenson [PeSt81a] suite à leur propre enquête [PeSt81b]. De ces travaux ressort une liste de considérations que nous nous sommes attachés à prendre en compte. (Se référer au mémoire d'habilitation de Jean-Luc Dekeyser [Deke90] dans lequel sont listés les desiderata des chercheurs du DoE et les propositions correspondantes de EVA.)

2–1 EVA est un langage vectoriel explicite

L'approche traditionnelle de la programmation vectorielle utilise un langage scalaire associé à un vectoriseur chargé de retrouver les constructions vectorielles dans des sources scalaires. Non pas que nous bannissions cette approche, mais il nous semble que la voie à suivre pour le développement de nouveaux outils de programmation vectorielle ne passe pas forcément par là.

Nous proposons donc un langage vectoriel **explicite**. Les objets et les constructions vectoriels sont apparents dans le langage et directement manipulés par le programmeur. Comme à de nombreux programmeurs [LiSc84, PeSt81a, Weth80], la manipulation explicite de vecteurs nous semble plus naturelle que l'écriture de DO loops codant ces mêmes opérations vectorielles. De plus une syntaxe explicite est à l'origine d'une plus grande clarté et lisibilité des programmes [CPHS83]. Un langage structuré incluant des constructions vectorielles est aussi un outil indispensable à la recherche en algorithmique vectorielle, un contrôle fin des opérations ne pouvant être obtenu par des constructions scalaires [CPHS83, LiSc84].

EVA est donc un langage vectoriel explicite. Tout algorithme manipulant des vecteurs utilise une syntaxe vectorielle. L'utilisation d'une structure de contrôle itérative ne produit jamais de code vectoriel. Les

vecteurs sont explicitement déclarés et utilisés dans les expressions et instructions vectorielles du langage. Deux types de données existent en EVA : les scalaires et les vecteurs. La plupart des opérateurs acceptent des opérandes des deux types. Un opérande scalaire apparaissant dans une opération vectorielle est considéré comme un vecteur de taille suffisante dont tous les éléments ont comme valeur celle du scalaire. Ces caractéristiques sont communes à la majorité des extensions vectorielles de langages existants (cf. la première partie de ce document).

2–2 Le vecteur EVA — Une structure de données pour la programmation vectorielle

Le vecteur est la structure de données manipulée par un langage vectoriel. Cette section discute de la conception du vecteur EVA. Traditionnellement, un vecteur est un tableau de valeurs pouvant être accédées en parallèle. Deux raisons majeures guident ce choix. Les structures de données doivent être adaptées aux problèmes traités : le tableau est particulièrement adapté aux algorithmes mis en place dans le cadre de la programmation scientifique. Un principe fondamental dans le cadre de la programmation (vectorielle) est "les structures de données doivent s'implanter facilement et efficacement sur la machine cible". Suivant ce principe, Bossavit et Meyer dégagent de leur analyse sur le Cray-1 [BoMe81] que les structures de données doivent se limiter aux traditionnels tableaux dont les éléments sont contigus en mémoire et aux listes d'éléments régulièrement espacés en mémoire. Nous avons bâti les vecteurs EVA à partir de cette structure de base.

2–2.1 Les opérations de descriptions

Une opération spécifique au traitement vectoriel est l'opération de description. Elle sélectionne un ensemble d'éléments d'un vecteur. Quatres types de sélections sont généralement distingués :

Sélection d'une tranche de vecteur Deux valeurs, lower et upper, déterminent, respectivement, les indices minimum et maximum des éléments sélectionnés du vecteur.

Sélection d'éléments régulièrement espacés dans le vecteur Un triplet lower, upper et step sélectionne les éléments du vecteur d'indice lower, lower + step, lower + 2 * step, etc...

Sélection par un masque de bits A chaque élément du vecteur est associée une valeur binaire. Les éléments sélectionnés sont ceux dont la valeur du bit correspondant est vraie. Ces opérations de description sont nommées compress/extend.

Sélection suivant une liste d'indices Les éléments sélectionnés sont les éléments dont l'indice est précisé dans la liste d'indices. Ces opérations de description sont traditionnellement nommées gather/scatter.

Ces opérations sont des opérations d'accès à des *sous-vecteurs*. Un accès à un sous-vecteur devrait autoriser :

- la lecture des composantes formant le sous-vecteur,
- l'écriture de ces éléments,
- le passage de ce sous-vecteur en paramètre à une fonction, et
- l'utilisation de ce sous-vecteur comme un vecteur. En particulier, il doit être possible de lui appliquer une opération de description.

Les modifications réalisées sur les éléments du sous-vecteur doivent pouvoir être répercutées aux éléments du vecteur; et ce, sans recopie des éléments dans le vecteur suite à l'opération. Considérons un algorithme quelconque traitant différemment les éléments de rang pair et impair d'un vecteur. Le programmeur ne doit pas être obligé de ranger les éléments à traiter dans un vecteur temporaire. Le choix entre les deux solutions doit lui être laissé. Ce choix peut être guidé par des raisons de performances ou autres.

```
V vecteur
V' = pair (V)
Traiter V' selon l'algorithme des éléments de rang pair
pair (V) = V'
V'= impair (V)
traiter V' selon l'algorithme des éléments de rang impair
impair (V) = V'
```

ou

```
V vecteur
Traiter pair (V) selon l'algorithme des éléments de rang pair
Traiter impair (V) selon l'algorithme des éléments de rang impair
```

2–2.2 Vecteurs et sous-vecteurs dans les langages vectoriels (et non-vectoriels)

La structure de données proposée par la grande majorité des langages vectoriels est le tableau. Toutefois, des fonctionnalités telles que celles décrites précédemment sont disponibles dans certains langages. Nous résumons ici les approches qui vont dans ce sens. De plus amples informations sont disponibles dans la première partie de ce document.

EQUIVALENCE L'instruction **EQUIVALENCE** du langage FORTRAN est un premier stade d'un partage d'éléments entre un vecteur et des sous-vecteurs [Fich85], [Metc88]. Une **EQUIVALENCE** peut définir un tableau "virtuel" comme une tranche d'un tableau FORTRAN. Le tableau virtuel ainsi défini est manipulé comme les autres tableaux FORTRAN. (Certaines extensions de FORTRAN traitent différemment les tableaux virtuels et les tableaux standards.)

Explicit descriptor Le langage VECTOR C est une extension vectorielle du langage C qui fut développée par Li et Schwetman [LiSc85]. Un vecteur de VECTOR C est représenté par un *descriptor*. Le compilateur gère des *implicit descriptors* pour son usage propre. Le programmeur peut définir des *explicit descriptors*. Un explicit descriptor est un couple (adresse de base des éléments, nombre d'éléments); il

référence donc une tranche d'un vecteur. Cette association adresse de base/longueur est dynamique et modifiable lors de l'exécution. Les opérations standards du langage C sont étendues aux descriptors. Cette construction fut reprise de FORTRAN 200.

Double descriptor Le langage STAR FORTRAN est une extension de FORTRAN implantée sur CDC STAR-100 [Hart77]. La version 2.0 du langage introduit la manipulation de tableaux, de vecteurs (association d'un tableau et du nombre d'éléments contenus dans le tableau) et de *double descriptors*. Un double descriptor est l'association d'un pointeur sur un vecteur et un pointeur sur une chaîne de bits. Les double descripteurs sont directement manipulés en STAR FORTRAN. Un double descriptor référence les éléments du vecteur dont la valeur correspondante dans la chaîne de bits est vraie.

Virtual array Le langage VECTRAN est une extension de FORTRAN réalisée par Paul et Wilson [PaWi75], [PaWi78]. VECTRAN inclut des manipulations explicites de tableaux et vecteurs. L'instruction `IDENTIFY` de VECTRAN construit, à l'exécution, un *virtual array*. Un virtual array est une référence à une tranche d'un tableau ou à des éléments régulièrement espacés d'un tableau. Ces virtual arrays sont ensuite manipulés en VECTRAN comme peuvent l'être les tableaux; ils peuvent même servir de base pour la construction de nouveaux virtual arrays. L'instruction `IDENTIFY` et les virtual arrays sont présentés par Paul et Wilson comme l'extension de FORTRAN la plus puissante proposée par VECTRAN.

Access pattern Dans le langage HELLENA, Jégou introduit le concept de *patron d'accès* [Jégo87a], [Jégo87b]. Un patron d'accès définit un nouvel objet dont les composantes sont des composantes d'un objet initial. Les patrons d'accès prédéfinis peuvent être étendus par la définition de nouveaux patrons.

Les pointeurs La nouvelle norme de FORTRAN—FORTRAN 90—propose au travers des *pointeurs arrays* le nommage d'une sous-section d'un vecteur [ANSI91], [MeRe90]. FORTRAN 90 définit des objets `POINTEUR` qui peuvent être dynamiquement associés à une section *lower* : *upper* : *step* d'un tableau. Une fois l'association réalisée, l'objet `POINTEUR` est utilisé dans les expressions comme le sont les tableaux FORTRAN 90.

Ces approches sont limitées suivant différentes directions :

1 L'introduction de multiples structures de données et constructeurs mènent à des incompatibilités entre les objets—par exemple lors du passage de paramètres (FORTRAN 90).

2 L'association statique, et fixée à la compilation, entre une "zone de données" et un "descripteur" limite l'intérêt de telles constructions (`EQUIVALENCE` de FORTRAN).

3 La "consommation" de l'association entre la "zone de données" et un "descripteur" (VECTOR C, HELLENA) étend à peine les opérateurs classiques de descriptions.

4 La limitation dans le choix des "descripteurs" ne permet pas d'assurer toutes les opérations de descriptions habituelles (VECTRAN, STAR FORTRAN, etc...).

2–2.3 Le vecteur EVA : deux zones

Nous étudions ici la principale construction proposée par EVA : le vecteur EVA. Il n'existe pas de tableau en EVA. Les objets de base sont les scalaires et les vecteurs. Les scalaires sont traditionnels et sans

originalité. EVA propose aussi des objets structures et multi-vecteurs; les caractéristiques de ces deux derniers seront présentées dans des sections spécifiques. Nous détaillons maintenant le vecteur EVA.

Un vecteur peut contenir plusieurs valeurs d'un même type : ses *éléments*. Mais un vecteur EVA n'est pas un classique tableau des langages de programmation tel FORTRAN. Un vecteur EVA est l'association d'une liste et d'un moyen de sélectionner, parmi cette liste, les éléments du vecteur. La liste peut être vue comme un tableau FORTRAN (tableau contigu de valeurs). Le moyen de sélectionner les éléments du vecteur peut préciser que le vecteur est formé de toutes les composantes du tableau; ou être composé d'une liste des index des éléments du vecteur; ou préciser pour chaque composante si elle appartient au vecteur ou non. Toute référence à un vecteur se rapporte à tous ses éléments.

En EVA, les vecteurs sont, pour la plupart, dynamiques. Le nombre de composantes (et d'éléments) d'un vecteur n'est a priori pas connu à la compilation.

Les vecteurs sont les structures de données de base du langage EVA. Les vecteurs sont manipulés comme entités par les opérateurs, peuvent être passés en paramètres aux fonctions, et retournés comme valeurs des fonctions.

En EVA, nous avons intégré les descripteurs dans les vecteurs. Un vecteur EVA est donc formé de deux zones (figure 2–1)

- une zone d'allocation, et
- une zone de description.

Par rapport aux exigences mises en évidence dans les sections précédentes et aux réponses proposées par les langages vectoriels habituels, notre proposition de vecteur EVA offre quelques points forts.

Nous ne proposons qu'une structure de données : le vecteur (en plus des objets scalaires). Nous évitons ainsi les incompatibilités éventuelles entre un type vecteur et un type tableau par exemple (les tableaux étant vus comme des vecteurs ne pouvant être accédés en parallèle).

La liaison entre un vecteur et ses zones de descriptions et d'allocation est dynamique. Un même vecteur peut changer de zone au cours de l'exécution d'un programme; de plus les zones sont créées dynamiquement. Nous ne limitons en rien la taille et la dynamicité des structures de données, donc des problèmes traités au sein du langage.

L'association au sein d'un vecteur d'une zone d'allocation et d'une zone de description et le partage d'une zone d'allocation entre plusieurs vecteurs autorise la manipulation de "sous-vecteurs" sans obliger des copies et recopies d'éléments. On peut écrire

```
Traiter pair (V) selon l'algorithme des éléments de rang pair
```

De nombreuses opérations de description sont disponibles en EVA. De plus les descriptions peuvent être combinées : un vecteur décrit est un vecteur; il peut à nouveau être décrit.

Nous détaillons, dans les sous-sections suivantes, les caractéristiques des vecteurs EVA.

2–2.3.1 Zone d'allocation

La zone d'allocation est une zone mémoire contiguë contenant un certain nombre de valeurs du type du vecteur (booléen, entier ou flottant). Elle peut être comparée aux classiques tableaux de langages tel FORTRAN. Ces valeurs sont nommées *composantes du vecteur*.

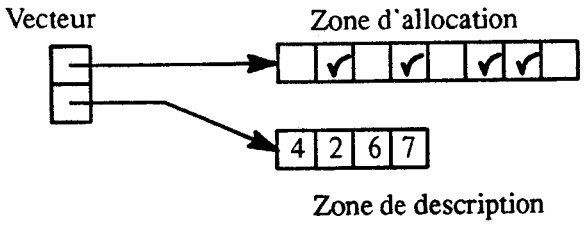
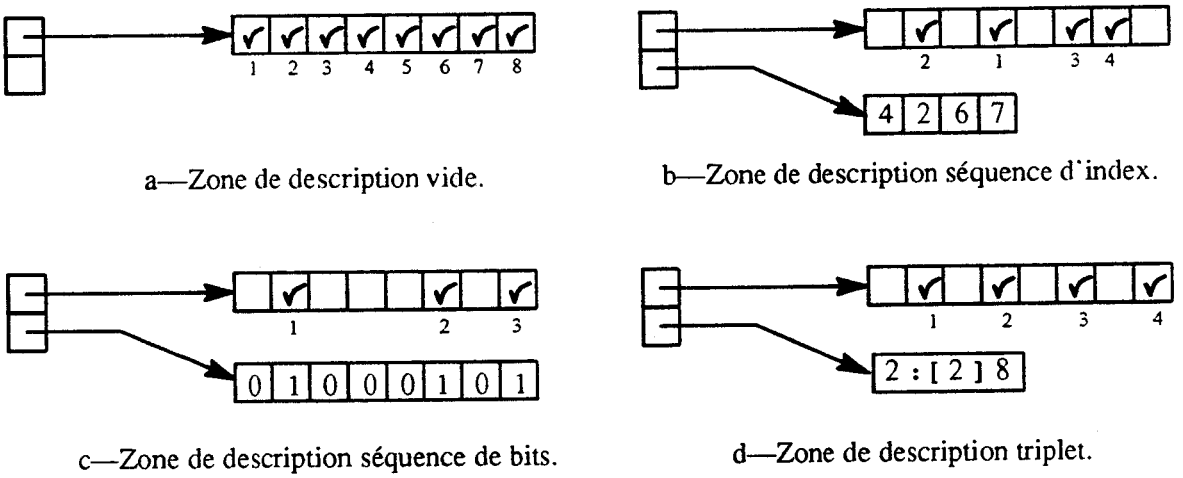


Figure 2–1. Le vecteur EVA.



a—Zone de description vide.

b—Zone de description séquence d'index.

c—Zone de description séquence de bits.

d—Zone de description triplet.

Figure 2–2. Les différents types de zones de description d'un vecteur EVA.

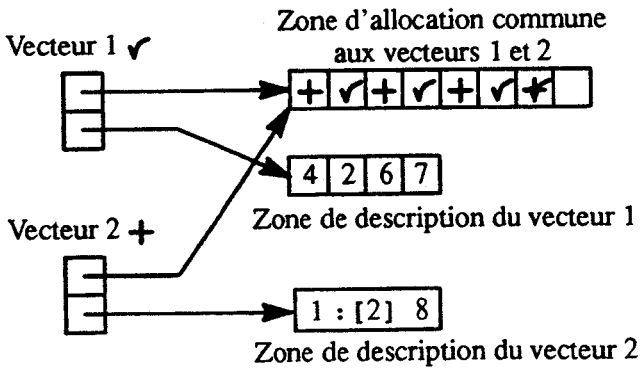


Figure 2–3. Partage d'une zone d'allocation entre deux vecteurs.

Une zone d'allocation n'est pas propre à un vecteur. Plusieurs vecteurs peuvent se partager une même zone d'allocation. Nous revenons sur cette caractéristique à la section 2–2.3.6.

2–2.3.2 Zone de description

La zone de description contient un moyen de sélectionner les éléments du vecteur parmi les composantes de la zone d'allocation; cette zone peut être

vide Le vecteur est alors formé de toutes les composantes de la zone d'allocation. Le vecteur possède donc autant d'éléments qu'il y a de composantes dans la zone d'allocation.

Elle peut être constituée de (figure 2–2) :

une séquence d'index (entiers) Les éléments de la zone d'allocation dont l'indice est dans la séquence forment les éléments du vecteur. L'ordre dans lequel ces éléments sont sélectionnés est défini par l'ordre de leur indice dans la séquence d'index. Un même élément de la zone d'allocation peut être sélectionné plusieurs fois par répétition du même index. Le nombre d'éléments du vecteur est égal au nombre d'index de la zone de description.

un triplet *lower, upper, step* La fonctionnalité est identique à celle d'une zone séquence d'index; les éléments de la séquence d'index sont produits par un triplet. Les éléments sélectionnés forment alors une tranche de la zone d'allocation (pour un pas de 1) ou sont régulièrement espacés dans la zone d'allocation. L'accès à de tels éléments est, de manière générale, plus économique que l'accès via une séquence d'index.

une séquence de bits Les éléments du vecteur sont obtenus par masquage de la zone d'allocation. Les éléments sélectionnés sont ceux dont le bit correspondant est vrai. Les éléments sont sélectionnés dans leur ordre dans la zone d'allocation. Le vecteur est formé d'autant d'éléments qu'il y a de valeurs booléennes vraies dans la zone de description.

Une zone de description est propre à un vecteur; nous revenons sur ce fait ci-dessous.

2–2.3.3 Composantes et éléments d'un vecteur

La terminologie utilisée dans ce document différencie les termes *composantes* et *éléments* d'un vecteur. Les composantes d'un vecteur forment la zone d'allocation du vecteur. Les éléments d'un vecteur sont les composantes de ce vecteur qui sont sélectionnées par la zone de description du vecteur.

Généralement, le terme vecteur désigne l'ensemble de ses zones d'allocation et de description. Il sous-entend donc "éléments du vecteur". Les opérations appliquées aux les vecteurs sont réalisées sur les éléments des vecteurs; les composantes d'un vecteur sont une zone de stockage d'éléments. (Une exception notable est l'opérateur d'association <- qui lie une zone d'allocation et une zone de description pour former un vecteur; cet opérateur est décrit à la section 2–3.2.)

Cette référence est réalisée que le vecteur soit en partie droite d'une affectation (*r-value*) ou en partie gauche (*l-value*). Un vecteur r-value référence les valeurs des éléments sélectionnés. Un vecteur l-value autorise la mise à jour des valeurs de ces éléments.

2–2.3.4 Longueur d'un vecteur

La *longueur d'un vecteur* est le nombre d'éléments composant le vecteur. Si la zone de description est vide, la longueur est égale au nombre de composantes du vecteur. Si la zone de description est une séquence d'index, la longueur du vecteur est le nombre d'index formant cette liste. Si la zone de description est une liste de valeurs booléennes, la longueur est le nombre de valeurs booléennes vraies dans cette liste.

2–2.3.5 Type des vecteurs EVA

Le type d'un vecteur EVA est celui de ses composantes. Les vecteurs EVA sont des vecteurs de valeurs réelles (simple ou double précision), entières, caractères ou booléennes. Nous insistons sur les vecteurs de booléens. Ceux-ci pouvant être représentés de manière économique (un bit par valeur) sur certaines machines vectorielles. Les opérations sur ce type d'opérandes sont elles aussi économiques; nous les avons introduites en EVA. Les opérations de description par liste de valeurs booléennes ne sont performantes que si la représentation de ces listes de valeurs booléennes est adaptée (par exemple une valeur par bit, et non une valeur par octet ou mot); l'introduction des vecteurs de booléens dans le langage autorise les compilateurs à implanter ces objets en tant que tels.

2–2.3.6 Partage de zones

Une zone d'allocation peut être partagée par plusieurs vecteurs. Ils possèdent alors chacun leur "point de vue" sur la zone d'allocation. Deux vecteurs se partageant la même zone d'allocation peuvent sélectionner des éléments communs (figure 2–3).

A l'inverse une zone de description d'un vecteur lui est propre et il n'existe aucun moyen en EVA de référencer cette zone en dehors de son utilisation implicite pour sélectionner les éléments du vecteur. EVA réalise les copies nécessaires pour éviter tout partage de cette zone de description. Cette propriété d'accès non partagé sur les zones de description est importante. Une fois les caractéristiques de cette zone connues, on est certain qu'elles ne seront pas affectées par des opérations dans lesquelles le vecteur n'est pas impliqué. On a donc une propriété de stabilité des zones de descriptions : la zone de description d'un vecteur est uniquement modifiée (et référencée) par (dans) les opérations réalisées sur le vecteur.

2–2.3.7 Représentation interne des vecteurs : l'en-tête de vecteur

La représentation interne d'un vecteur est formée d'un en-tête. Au niveau programmeur EVA, il n'est pas nécessaire de connaître la constitution précise d'un en-tête. Toutefois, il peut être intéressant de noter que l'en-tête d'une variable vecteur contient :

- Une référence vers la zone d'allocation du vecteur à laquelle est associé un compteur de références vers cette zone (partage d'une zone d'allocation entre différents vecteurs).
- Une référence vers une zone de description. Il lui est bien évidemment associé le type de cette zone (vide, séquences d'index, de booléens, ou triplet).
- Le nombre de composantes et d'éléments du vecteur.

C'est via cet en-tête qu'est manipulée une variable vecteur. C'est également un tel en-tête (une référence sur un tel en-tête) qui est passé à une fonction lorsque l'on passe un paramètre vecteur à une fonction (que ce paramètre effectif soit une variable ou une expression temporaire).

2–2.4 Les vecteurs triplets

Une sous-catégorie des vecteurs d'entiers est formée par les triplets. Les vecteurs triplets sont construits à partir du constructeur de triplet

```
: [ ],
```

pouvant être abrégé en

```
:
```

Un triplet est de la forme (dans la syntaxe EVA)

lower : [*step*] *upper*

Un tel vecteur triplet est fonctionnellement équivalent au vecteur d'entiers formé de la suite de valeurs

lower, *lower* + *step*, ... *lower* + *n* * *step*.

Pour *step* positif, *n* est le plus grand entier positif tel que *lower* + *n* * *step* ≤ *upper*;
pour *step* négatif, *n* est le plus grand entier positif tel que *lower* + *n* * *step* ≥ *upper*.

Les triplets sont manipulés en tant que tel (c'est-à-dire comme trois valeurs entières) en EVA. Plusieurs avantages sont liés à cette manipulation :

- Trois entiers représentent un vecteur de taille quelconque. Le rangement en mémoire et l'accès d'un vecteur triplet est plus rapide.
- La réalisation des opérations qui sont des lois internes sur les triplets (addition, soustraction) et celles combinant un triplet et un entier scalaire qui produisent un triplet (addition, multiplication) est optimisée.
- Les opérations de description d'un vecteur par un triplet sont optimisées par rapport à celles de description par un vecteur d'entiers. On se ramène d'opérations de gather/scatter à des opérations d'accès par pas ou même d'accès contigus dans le cas où la valeur de *step* est 1.

Le constructeur de triplet est utilisé pour la construction des valeurs littérales triplets telles

```
1 : 100    ou    4 : [2] 200
```

Ce même constructeur est utilisé pour la définition d'expressions vecteurs d'entiers par exemple

```
int i, j , k
... (i + j) :[ k ] (25 * i + j) ...
```

L'association d'un vecteur triplet à un vecteur d'entiers conserve la forme triplet. Le vecteur est alors nommé *vecteur triplet*. (Cf. infra la description de l'opérateur d'association <-.)

```
vtriplet <- 1 :[2] 10
```

En EVA, une variable vecteur d'entier triplet n'est pas distinguée d'une variable vecteur d'entier. C'est le compilateur qui décide de la représentation d'un vecteur —triplet ou étendue— à partir de son utilisation. Des opérations d'extension de vecteurs triplets peuvent intervenir pour changer la forme d'un vecteur triplet en un vecteur étendu. Par exemple, l'addition d'une expression vecteur non triplet réalise une telle extension (`1|2|3|4|5` est un vecteur littéral formé des valeurs 1, 2, 3, 4, et 5) :

```
vtriplet +:= 1|2|3|4|5
```

L'extension de triplet est automatiquement prise en compte par le compilateur qui génère le code nécessaire à sa réalisation, la zone d'allocation étendue sera allouée dynamiquement et automatiquement libérée.

2–2.5 Les vecteurs temporaires

Les vecteurs temporaires sont utilisés pour stocker les résultats intermédiaires lors de l'évaluation d'expressions. C'est aussi ce type de vecteurs qui est retourné par les fonctions EVA (cf. section 2–10.4 page 115). Ces vecteurs ne sont pas manipulés directement par le programmeur EVA. Un vecteur temporaire est composé d'une suite contiguë d'éléments; il ne possède ni en-tête ni zone de description. La suite d'éléments formant le vecteur temporaire peut être considérée comme une zone d'allocation à laquelle n'est pas associée de zone de description. EVA gère aussi des vecteurs temporaires triplets; ils sont représentés par une liste de trois valeurs.

2–2.6 Les valeurs littérales vecteurs

Les valeurs vecteur littérales EVA sont composées à l'aide des deux opérateurs de concaténation

`|`

et de concaténation répétée

`//`

Le constructeur `|` concatènent les valeurs de ses opérandes droite et gauche. Le littéral

`1|1|3|5`

est formé des valeurs 1, 1, 3 et 5. Un vecteur bâti à l'aide du constructeur de concaténation répétée `//` est composé de *N* concaténations de son opérande droit, *N* étant la valeur de son opérande gauche. Les littéraux

`1|1|1` et `3 // 1`,

tout comme

`1|2|1|2` et `2 // (1|2)`,

dénotent la même suite de valeurs.

De plus des valeurs vecteur d'entiers littérales sont formées à l'aide des opérateurs de constructions de triplets

`:` et `:[ ]`

(cf. la section 2–2.4 ci-dessus). Il existe aussi des valeurs littérales vecteurs de booléens sous forme de chaînes de valeurs booléennes par exemple

`b'1010_0001_1110'` ou `h'A1E'`.

EVA défini aussi le *vecteur nul* comme un couple de crochets

```
[ ]
```

il n'est composé d'aucun élément; il est de n'importe quel type. Ce vecteur est utilisé comme initialisation d'un vecteur, par exemple pour la réalisation d'opérations de concaténation successives.

2–2.7 Déclaration de vecteurs

La déclaration d'un vecteur détermine le nom du vecteur et son type. Les types disponibles sont : booléen, caractère, entier, réel simple et double précision. De plus une allocation/association initiale peut être faite lors de la déclaration (se reporter à la section 2–3 ci-après décrivant l'opérateur d'association EVA). Exemples de déclaration de vecteurs EVA :

```
vect int vectint
vect real v1, v2, v3
```

2–2.8 Projections multi–dimensionnelles des vecteurs EVA

Les vecteurs EVA sont mono-dimensionnels par défaut. Des informations supplémentaires peuvent être associées à un vecteur pour autoriser des projections multi-dimensionnelles. Notons bien que ces informations sont indépendantes, dans la majorité des cas, de la longueur du vecteur. Elles sont attachées à une variable vecteur et spécifiée par le programmeur lors de la déclaration. Ces informations sont :

- Le ***rang*** et la ***dimension*** Le rang (rank) d'un vecteur est le nombre de dimensions d'un vecteur. Le rang d'un vecteur est fixé à la compilation et est défini par la déclaration du vecteur. Toutefois, les vecteurs EVA sont multi-rangs : un vecteur peut avoir plusieurs rangs; chaque opération d'accès aux éléments du vecteur indique le rang effectivement utilisé. Par exemple

  ```
  vect int multi [100] [10, 3, 4] [12, 8]
  ```

  Le vecteur *multi* est accessible comme un vecteur de rang 1, 2 ou 3 :

  ```
  multi [10] = ...
  multi [1:10, 1:5] = ...
  multi [1:10, 2, 1|3] = ...
  ```

  Les vecteur multi-dimensionnels sont alloués de telle manière que deux éléments de la dernière dimension soient contigus en mémoire. On a donc l'équivalence entre les deux accès

  ```
  multi [43]    et    multi [3, 4]
  ```

- Les ***bornes*** et la ***forme*** Les bornes précisent pour une dimension donnée la variation des indices des éléments du vecteur. La forme d'un vecteur est la réunion, pour toutes les dimensions, des informations de bornes. Les bornes sont des couples *borne-inférieure*, *borne-supérieure*. Les valeurs des bornes inférieures et supérieures n'ont pas à être connues à la compilation. La valeur par défaut de la borne inférieure est 1. Exemple

  ```
  vect real  v [2 .. 12]
             w [f (14) + 1]    -- équivalent à w [1 .. f(14) + 1]
  ```

Dans la déclaration du vecteur *w*, l'expression *f(14)* est une valeur retournée par un appel à la fonction *f*; elle est évaluée lors de la déclaration de la variable. Les descriptions des vecteurs prennent ensuite en compte les informations de forme.

La forme peut aussi ne pas être précisée par le programmeur. La borne supérieure de la dernière dimension du vecteur dépend alors de la longueur effective du vecteur (nombre d'éléments). Un tel vecteur est qualifié d'*assumed-length*. Le symbole **?** est utilisé comme expression de la borne supérieure lors de la déclaration.

```
vect real vass [20, ?] [?] <- 110
vass <- 1000
```

Lors de sa déclaration, le vecteur *vass* contient 110 éléments (association <- 110); le vecteur est donc manipulé comme *vass [20, 5] [110]*. L'instruction suivante alloue 1000 éléments au vecteur *vass*; il est donc ensuite manipulé comme *vass [20, 50] [1000]*.

Les vecteurs mono-dimensionnels et les bornes de rang 1 sont *assumed-length* par défaut. Une déclaration

```
vect real  vass1
           vass2 [12, 10]
```

est donc équivalente à

```
vect real  vass1 [?]
           vass2 [?] [12, 10]
```

Ces projections multi-dimensionnelles des vecteurs facilitent la manipulation par le programmeur de tableaux à plusieurs dimensions telles les matrices. EVA ne limite pas les accès en parallèle aux éléments d'un vecteur multi-dimensionnel à ceux d'une seule dimension comme le font les langages CFD, ACTUS, VECTRAN [LiSc85, pp.141]. Tous les éléments d'un vecteur multi-dimensionnel peuvent être accédés en parallèle en EVA. Toutefois, les éléments d'un vecteur EVA sont toujours rangés de manière linéaire et contiguë en mémoire. Il est donc à charge du compilateur EVA de transformer un accès multi-dimension en un accès linéaire.

Les informations de rang et forme d'un vecteur sont attachées à son nom. Elles ont donc une visibilité limitée à la portée de ce nom. Ces informations ne sont pas passées lors du passage d'un vecteur en paramètre. Lors d'un tel passage, mises à part les zones d'allocation et de description, seule la longueur du vecteur est passée. La déclaration dans la fonction appelée du paramètre formel est libre de définir des informations de rang et forme qui seront propres à cette fonction appelée.

```
vect real vecteur [100, 20]
...
fformel (vecteur)
fforme2 (vecteur)
...
fformel (v)
      vect real v
      ...
end

fforme2 (v)
      vect real [1000 .. 2500]
      ...
end
```

Le vecteur *vecteur* de rang 2 est passé à la fonction *fformel*. La déclaration du paramètre ne précise aucune information de forme; le vecteur est alors *assumed-length*, mais mono-dimensionnel; les informations de forme associées à *vecteur* lors de sa déclaration ne sont pas visibles. Le même vecteur est ensuite passé à la fonction *fforme2*; le paramètre est déclaré mono-dimensionnel avec une forme précise; ce sont ces informations qui sont associées au vecteur au sein de la fonction *fforme2*.

2–3 Allocation et association de vecteurs

Si la manipulation des éléments des vecteurs EVA est réalisée par les opérateurs habituels, il nous a fallu introduire des opérations spécifiques pour la définition, l'allocation et l'initialisation de ces vecteurs. Afin de clarifier les choses, nous avons distingué les opérations de déclaration et d'allocation des vecteurs. Dans son enquête [Weth80], Wetherell rapporte le désir de structures de données de taille dynamique et modifiable à l'exécution. Perrott indique aussi que le programmeur doit pouvoir exprimer un parallélisme variable et non fixé à la compilation [Perr79]. Les vecteurs EVA possèdent ces caractéristiques de dynamicité. Toutefois, pour des raisons de gain d'efficacité, certains vecteurs peuvent avoir leurs caractéristiques fixées dès leur déclaration.

2–3.1 L'allocation des vecteurs EVA

Un vecteur EVA est constitué d'un en-tête référençant une zone d'allocation et une zone de description. Un tel en-tête est de taille fixe et constante. L'allocation des en-têtes est toujours réalisée lors de la compilation. Il n'y a pas de création dynamique de nouveaux objets en EVA; seules les zones d'allocation et de description peuvent être allouées dynamiquement (cf. la sous-section suivante).

2–3.2 L'opérateur d'association

Nous détaillons ici l'opérateur EVA d'association, noté <-. Cet opérateur lie un vecteur cible avec ses zones d'allocation et de description. Il réalise

- l'allocation (dynamique ou non) des zones d'allocation,
- le partage de zone d'allocation, et
- la création de zone de description.

Trois types d'association sont distingués : l'association *dynamique*, l'association *de partage*, et l'association *dynamique initialisée*. Le type de l'association dépend des caractéristiques de la partie droite de l'association.

Association scalaire ou **association dynamique** Une expression scalaire est associée au vecteur. La valeur de cette expression est la taille d'une zone d'allocation qui est allouée pour le vecteur. La zone de description du vecteur est vide.

Association de partage La partie droite de l'association est limitée à un nom de vecteur W décrit ou non par une expression de description D :

$$V <\text{-}\ W \quad \text{ou} \quad V <\text{-}\ W\ [\ D\ ]$$

Le vecteur V partage la zone d'allocation du vecteur W. Dans le premier cas, $V <\text{-}\ W$, la zone de description de V est initialisée avec une copie (éventuelle) de la zone de description de W. Dans le second cas, la zone de description de V est obtenue par *description* de la zone de description de W par l'expression D.

Association dynamique initialisée Une expression vectorielle, non réduite à un nom de vecteur ou à un nom de vecteur décrit (cf. association de partage), est associée à un vecteur. Une zone d'allocation est allouée pour le vecteur, elle est initialisée avec une évaluation de cette expression. (L'allocation et la copie peuvent parfois être évitée; la zone d'allocation du vecteur est alors la zone d'allocation du temporaire.)

2–3.3 Exemples d'associations

Considérons le code EVA déclarant quatre vecteurs

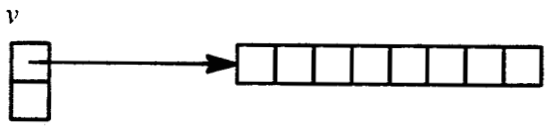
```
vect real v <- 8
vect int ind <- 4|2|6|7
vect real impair <- v [1 :[2] 8]
          bizz   <- v [ind]
```

Le vecteur *v* est initialisé par une association scalaire. Le vecteur *ind*, construit par une association dynamique initialisée, est composé d'une zone d'allocation contenant les valeurs 4, 2, 6, et 7. Une association de partage est réalisée sur chacun des vecteurs *impair* et *bizz*. Ils partagent donc tous deux la zone d'allocation de *v*. La zone de description de *bizz* est une copie du vecteur *ind* (figure 2–4).

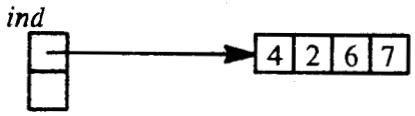
2–3.4 Réalisation de l'association

Nous discutons ici de la mise en place des opérations d'association. Nous ne présentons pas l'implantation de l'opérateur <- (pour cela, on se reportera à la partie 2 de ce chapitre), mais donnons les informations complémentaires nécessaires au programmeur désirant manipuler les vecteurs EVA.

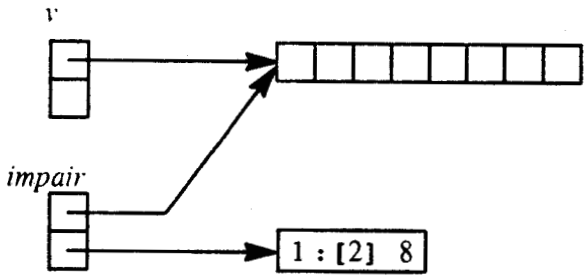
```
vect real v <- 8
```




```
vect int ind <- 4|2|6|7
```




```
vect real impair <- v [1 :[2] 8]
```




```
vect real bizz <- v [ind]
```

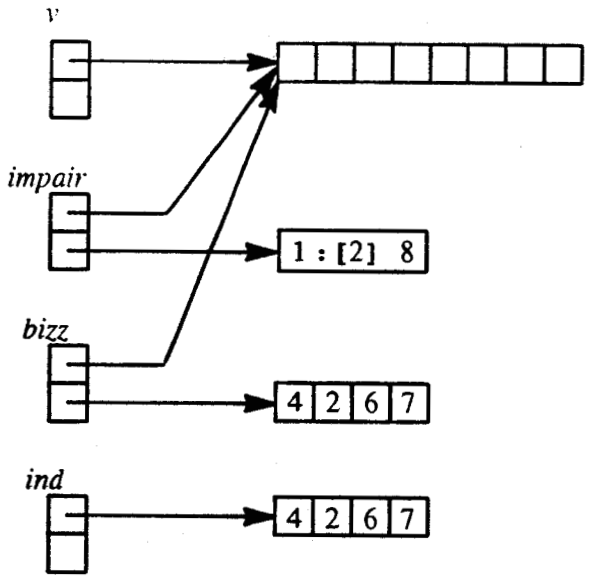


Figure 2–4. Les vecteurs résultant des associations.

Une association peut être appliquée à un vecteur lors de sa déclaration. Une opération d'association est aussi une instruction; l'association d'un vecteur avec ses zones d'allocation et de description n'est donc pas figée lors de la déclaration du vecteur. Cependant, nous définissons des vecteurs à association unique (déclarés `unassoc`) qui doivent être associés lors de leur déclaration et qui ne pourront plus l'être ensuite; les optimisations possibles sur ces vecteurs `unassoc` seront présentées à la section 2–5.2.

Préalablement à toute association, les anciennes zones d'allocation et de description du vecteur cible sont libérées. Toutefois, une zone d'allocation encore référencée par un autre vecteur ne sera pas libérée. Associée à chaque zone d'allocation, EVA gère un compteur de références sur cette zone d'allocation.

La zone de description d'un vecteur lui est propre. La production d'une zone de description pour un vecteur lors d'une association est donc toujours réalisée par l'allocation d'une nouvelle zone et recopie des valeurs. Dans le cas où la zone à copier "meurt" après l'association, on évite une nouvelle allocation et une recopie en reprenant cette zone comme zone de description. Par exemple :

```
a <- b [c + d]
```

l'expression $c + d$ est évaluée et rangée dans un temporaire; ce temporaire deviendra la zone de description de a.

2–4 L'opérateur de description

2–4.1 Les descriptions retenues en EVA

Comme énoncé plus haut, les opérations de description sélectionnent un ensemble d'éléments parmi ceux d'un vecteur. Cet ensemble forme un sous-vecteur. Les quatres sélections vectorielles —sélection d'une tranche de vecteur, sélection d'éléments régulièrement espacés dans le vecteur, sélection par un masque de bits, et sélection suivant une liste d'indices— sont disponibles en EVA. Ne pas proposer une de ces fonctionnalités limiterait la facilité d'expression des algorithmes vectoriels.

Un opérateur unique, le [ ] assure toutes les fonctionnalités de description. Le type de sélection dépend du type de l'opérande droit. (En EVA, cinq types de base sont disponibles pour les vecteurs : les types booléen, caractère, entier, et réels simple et double précision. Trois types seulement sont disponibles pour les scalaires; les valeurs scalaires booléennes et caractères sont codées par des entiers.)

Sélection d'une tranche de vecteur et sélection d'éléments régulièrement espacés dans le vecteur Ces sélections sont obtenues par description du vecteur par un triplet *lower*, *upper*, *step*. L'expression *step* est optionnelle; sa valeur par défaut est 1; on sélectionne alors une tranche du vecteur.

Sélection par un masque de bits Elle est réalisée lorsque l'opérande droit est un vecteur de valeurs booléennes.

Sélection suivant une liste d'indices Elle est réalisée par un opérande droit vecteur d'entiers.

De plus la description d'un vecteur par une expression scalaire entière de valeur n produit un résultat scalaire : le $n^{ième}$ élément du vecteur.

EVA ne teste pas les dépassements de bornes inférieures et supérieures lors des accès à des sous-vecteurs. Il est donc de la responsabilité du programmeur de s'assurer que de tels cas ne surviendront pas. rendus systématiques, de tels tests de débordement pénaliseraient l'exécution. De plus, le déroulement qui découlerait d'un tel débordement devrait pouvoir être contrôlé par le programmeur. (Par exemple, il semble inadmissible d'interrompre a priori l'exécution ou de "laisser passer" de tels accès une fois qu'ils ont été détectés.)

2–4.2 Interprétation de la description par une liste de valeurs entières

Il existe au moins deux interprétations de la description d'un vecteur par une liste d'index (cf. [HoJe88, pp 389–sq] et la première partie de ce document).

L'interprétation de projection, proposée par ACTUS par exemple, n'a pas été retenue en EVA. La seconde interprétation, généralisation de l'indiçage par un entier scalaire, a été retenue par EVA.

La mise en place de l'interprétation de projection demande une stricte conformité de longueur entre le vecteur décrit et le descripteur. Une telle conformité ne peut être assurée en EVA. De plus, la seconde interprétation est plus consistante : la réduction de rang n'est possible que par une description scalaire [LiSc85]. Cette interprétation généralisant la description par un scalaire, elle est aussi plus proche de l'intuition des programmeurs déjà familiarisés avec un langage scalaire et autorise plus facilement une réécriture vectorielle des boucles. Une boucle

```
      REAL V (100, 200)
      INTEGER INDI (100), INDJ (200)
      ...
      DO 10 i = 1, 100
      DO 10 j = 1, 200
10          V (INDI (i), INDJ (j)) = ...
```

est réécrite en

```
      V (INDI, INDJ) = ...
```

(bien que la sémantique des deux opérations ne soit pas identique, en particulier en présence de dépendances entre partie gauche et droite de l'affectation vectorielle)

2–4.3 Réalisation de ces descriptions en EVA

EVA propose donc les différents types de description au travers un opérateur unique, le [ ]. Cet opérateur s'applique à toutes les expressions vectorielles du langage EVA; la description n'est pas limitée au nom de vecteur. Cette possibilité laisse le programmeur choisir entre les deux codes

```
vect real va, vb, vc
vect int int_expr
va = vb [int_expr] + vc [int_expr]
```

et

```
va = (vb + vc) [int_expr]
```

le choix de l'un ou de l'autre code est, par exemple, guidé par le taux d'éléments sélectionnés dans les vecteurs *vb* et *vc*. D'autre part, toutes les expressions EVA sont valides comme vecteur de description. Encore une fois, le vecteur de description n'est pas limité à un nom.

```
vect real v <- 100
vect int index <- 1 :[2] 99
v [index] = 1.0
v [index + 1] = 0.0
```

range 1.0 dans les éléments de rang impair de *v*, 0.0 dans les éléments de rang pair.

2–4.4 Multi-description

Le résultat d'une description par un vecteur produit un vecteur. EVA ne limite pas l'utilisation de ce vecteur résultat. Ce vecteur peut donc à nouveau être décrit. Ces descriptions multiples sont consommées de gauche à droite. Ces multi-descriptions sont utilisables à droite et à gauche des opérateurs d'affectations. Par exemple

```
v [d] [2:*] += v [d]
```

ajoute la valeur de son "prédécesseur" à chaque élément du vecteur *v [d]*. On pourrait écrire

```
vect real w <- v [d]
w [2:*] += w
```

et ainsi factoriser la description par *d*.

2–5 Vecteurs constants et à association unique

Les objets constants (**const**) et à association unique (**unassoc**) sont des objets sur lesquels les opérations de modifications ne peuvent pas toutes s'appliquer. Les opérateurs de modifications sont l'association et les affectations. Les limitations imposées au programmeur lors de la manipulation de ces vecteurs autorisent des optimisations lors de leur implantation.

2–5.1 Vecteurs constants

Un vecteur constant est un vecteur sur lequel aucun opérateur d'association ou d'affectation ne peut être appliqué après sa définition. Un tel vecteur doit donc être associé lors de sa définition. Comme il n'y a

ensuite plus moyen de modifier la valeur des éléments d'un tel vecteur, l'association initiale doit aussi donner une valeur initiale à ses éléments. Le vecteur ne peut donc pas être initialement associé avec une association dynamique. Toutefois, la valeur de la partie droite de l'association n'a pas à être évaluable à la compilation; toute expression EVA est valide.

Par souci de cohérence, une association de partage dont le vecteur cible est un vecteur constant doit avoir un vecteur source constant (éventuellement décrit). Une limitation est également appliquée lorsqu'un vecteur constant est vecteur source d'une opération d'association. Le vecteur cible de l'association doit aussi être un vecteur constant. (Une alternative est d'autoriser toutes les associations, mais de réaliser une copie afin de ne pas altérer la nature constante des vecteurs constants). De même, un mécanisme spécifique est mis en place lors du passage de tels vecteurs en paramètre; se reporter à la section 2–10.3.

Les vecteurs constants sont introduits pour autoriser la mise en place d'optimisations. Par exemple, l'allocation d'un vecteur constant peut être effectuée à la compilation si sa longueur est évaluable à ce stade. D'autre part, les caractéristiques d'un vecteur constant étant figées, ses attributs (au sens EVA/DEVIL; cf. 2–14.3) sont fixés lors de la déclaration et de l'association initiale. La génération de code utilise ces informations et détermine à la compilation des informations qui ne seraient sinon pas connues avant l'exécution. Par exemple une assignation initiale

```
const vect int vconst <- 1 :[pas] 200
```

assure que le vecteur constant *vconst* sera toujours un vecteur dont la zone d'allocation est sous forme triplet et qui ne possède pas de zone de description. (La valeur de *pas* n'a pas à être connue lors de la compilation).

2–5.2 Vecteurs à association unique

Un vecteur de classe de modification à association unique —un vecteur **unassoc**— est un vecteur qui n'est associé qu'une fois : lors de sa définition. Toute association ultérieure est interdite. Toutefois des affectations de ce vecteur (de ses éléments) sont possibles.

Les limitations associées aux vecteurs à association unique sont moindres que celle associées aux vecteurs constants. Cependant, certaines optimisations lors de l'utilisation de tels vecteurs restent valides. Par exemple si l'association initiale d'une vecteur **unassoc** est une association dynamique et si l'opérande droit de cette association est une expression scalaire évaluable à la compilation, alors l'allocation de la zone d'allocation du vecteur est réalisée à la compilation. Par exemple, la zone d'allocation du vecteur *alloc_pile* sera allouée sur la pile (via le segment *local* de la machine MAD, machine virtuelle associée à DEVIL, cf. la seconde section de ce chapitre) :

```
f ()
      unassoc vect real alloc_pile <- 1000
      ...
end
```

Comme pour les vecteurs constants, un mécanisme particulier est mis en place lors du passage en paramètre de tels vecteurs à une fonction. Une copie de l'en-tête du vecteur (copie des références sur les zones d'allocation et de description, donc partage de ces zones) est construite et passée à la fonction. Ainsi,

le vecteur ne peut pas être ré-associé par la fonction appelée mais les valeurs des éléments du vecteur peuvent être changées si le paramètre est affecté dans la fonction appelée. (Se référer à la section 2–10.3.3 traitant des passages de paramètres.)

2–6 Les objets instanciables

En EVA, une expression de description peut dépendre des caractéristiques de forme du vecteur décrit. Une telle description est dite *instanciable*. Les expressions et les objets instanciables sont construits à partir d'un nombre restreint de primitives instanciables fournies par EVA. Ces primitives référencent les valeurs des bornes inférieures et supérieures des différentes dimensions d'un vecteur ainsi que la taille de chacune de ces dimensions.

Nous définissons l'*instanciation* d'une primitive, d'une expression ou d'un objet instanciable comme la détermination de sa valeur. Cette instanciation est effective lors de l'utilisation de l'instanciable comme descripteur d'un vecteur. L'instanciation est réalisée relativement au vecteur décrit; les primitives instanciables prenant leur valeur des caractéristiques de ce vecteur.

2–6.1 Les primitives instanciables

Les primitives instanciables sont l'étoile `*`, et les trois primitives **`lower`**, **`upper`**, et **`size`**.

L'étoile `*` référence la valeur de la borne inférieure ou supérieure de la dimension décrite selon sa position dans l'expression de description. A gauche d'un constructeur de triplet `:` ou `:[ ]`, l'étoile référence la borne inférieure; partout ailleurs, l'étoile référence la borne supérieure. Exemple :

```
vect real   vcible [5..17] <- 13,
            vimpair, vpair, vcentre
real dernier_pair

vimpair <- vcible [* :[2] *]
vpair <- vcible [*+1 :[2] *]
dernier_pair = vpair [*]
vcentre <- vpair [*+1 : *-1]
```

Mise à part l'étoile, trois primitives instanciables sont disponibles. La primitive **`lower`** est instanciée en la valeur de la borne inférieure de la dimension, la primitive **`upper`** en la valeur de la borne supérieure de la dimension, et la primitive **`size`** en la taille (nombre d'éléments) de la dimension. Les deux expressions *v* `[* : *]` et *v* **`[lower :upper]`** sont donc équivalentes.

La valeur d'un instanciable est définie par les informations de forme fournies lors de la déclaration du vecteur. Toutefois, pour les vecteurs *assumed-length*, la valeur de la borne supérieure dépend non pas de la déclaration de forme, mais de la longueur effective du vecteur. Les primitives **`upper`** et **`size`** sont donc également liées à cette longueur effective. Exemple :

```
vect real assumed <- 1000
assumed [*] = 12.3
...
assumed <- 500
assumed [*] = 13.4
```

Lors de la première affectation, le vecteur *assumed* comporte 1000 éléments, c'est donc son 1000ème élément qui est affecté. Lors de la seconde affectation le vecteur comporte 500 éléments, c'est le 500ème qui est affecté.

Notons bien que pour les vecteurs non *assumed-length*, les primitives instanciables référencent les informations de forme qui ne dépendent pas de la longueur effective du vecteur. Suite à la déclaration

```
vect real v [100] <- 150
     ... v ...              -- 150 elements
     ... v [*:*] ...        -- 100 elements
```

v contient 150 éléments, mais est déclaré de forme 100; les deux références au vecteur *v* ne sont donc pas identiques.

Les informations de taille et de borne référencées par les primitives se rapportent aux informations de la dimension dans laquelle elles apparaissent. Dans une dimension donnée, les informations des autres dimensions sont référencées par des primitives paramétrées. Les deux paramètres sont, dans l'ordre, le rang (nombre de dimensions) et la dimension concernée. Cette information de rang est parfois nécessaire car, rappelons-le, les vecteurs EVA sont potentiellement multi-rangs; cette information est toutefois optionnelle. Exemple

```
vect real v [12, 6] [36]
v [upper (2, 1)] = 1
```

affecte l'élément *v [12]*.

2–6.2 Les objets instanciables

Les objets instanciables sont construits à partir de primitives instanciables et/ou d'autres objets instanciables. Ces objets ne peuvent être utilisés qu'au sein d'une expression de description de vecteur. Leur valeur n'est évaluée que lors de l'instanciation, en fonction des caractéristiques du vecteur décrit.

Les objets instanciables sont déclarés comme les autres objets EVA. Toutefois, ces objets doivent être déclarés constants; ils doivent donc être initialisés lors de leur déclaration (affectation initiale pour les scalaire, association initiale pour les vecteurs). L'expression initialisant ces objets doit être une expression instanciable (qui peut être réduite à une primitive instanciable).

La valeur d'un objet instanciable n'est pas évaluée lors de la déclaration de cet objet, mais lors de l'instanciation. Toutefois, des évaluation partielles, n'incluant pas de primitives, d'expressions ou d'objets instanciables, sont réalisées lors de la déclaration d'un objet instanciable.

Exemple :

```
const int milieu = (upper - lower)/2
const vect int extremes <- lower | lower+1 | upper-1 | upper
vect real  v10 [10],
           v1,
           v50 [50 .. 99]
int taille
...
v10 [milieu] = 12.0
v1 <- taille
v1 [extremes] = v50 [extremes]
```

Les variables *milieu* et *extremes* sont instanciables; une expression construite à base de primitives instanciables leur est associée/affectée lors de leur déclaration.

milieu est instancié lors de son utilisation comme description de *v10*; elle prend la valeur *(v10'upper – v10'lower)/2* soit *(10–1)/2=4*. (La quote ' est un appel à un attribut EVA. Les attributs *upper* et *lower* dénotent respectivement les bornes inférieure et supérieure d'un vecteur)

extremes est ensuite instanciée lors de la description de *v50*; elle prend la valeur *50/51/99/100*. Cette même variable *extremes* est instanciée lors de la description de *v1*. *v1* est une variable dite *assumed-length*; les limites de variation de ses indices dépendent donc de la taille effective du vecteur; *extremes* prend donc la valeur *1/2/taille–1/taille*.

Les objets instanciables multi-dimensions

Les objets instanciables multi–dimensions sont des structures. (Ces objets structures ne sont pas définis dans ce document, on se reportera au manuel de référence EVA si nécessaire.) Leur valeur est obtenue depuis un agrégat (construction de structures littérales suivant la syntaxe << *liste–de–champs* >>). Chaque champ élémentaire est le descripteur de la dimension correspondant à son ordre dans la liste des champs.

```
# include "instype.h"
-- ce fichier contient les définitions de types DIM_VV and DIM_SV

const type DIM_VV corner2d   = <<lower | upper, lower | upper>>
const type DIM_SV first_line = <<lower, * : *>>
vect real twoD [paraml, param2]
....
twoD [corner2d] = 0.0
-- est equivalent à
-- twoD [1, 1] = twoD [1, param2] = twoD [paraml, param2] = twoD
[paraml, 1] = 0.0

twoD [first_line] +:= 1.0
```

2–7 Spécification de longueur

Un problème rencontré dans les langages de manipulation de vecteurs est celui que nous nommerons la *conformance des opérandes*. (Certains auteurs (par exemple le référence FORTRAN) utilisent ce terme de *conformance* dans un autre sens.) Considérons la somme de deux vecteurs : $V + W$. La sémantique de l'opération est définie comme

$$V[i] + W[i], i \in 1..N$$

La conformance est le problème de détermination de la valeur de N. Une règle telle que "N doit être la taille de V et W" semble trop restrictive.

- Elle limite la dynamicité des opérations et des vecteurs si elle doit être vérifiée lors de la compilation (c'est le cas du langage ACTUS).

- Elle ralentit l'exécution si elle est testée à l'exécution.
- Elle entraîne une ambiguïté dans le résultat en cas de non respect de la règle si elle n'est pas testée (c'est le cas de FORTRAN).

D'autres approches, telle l'extension de LRLTRAN [Zwak75], réalisent les opérations arithmétiques sur la longueur du plus long des vecteurs; le vecteur le plus court est "rallongé" par des éléments neutres pour l'opération.

VECTOR C [LiSc85] n'oblige pas deux vecteurs impliqués dans une opération à avoir le même nombre d'éléments. Si un opérande vectoriel partie gauche est de longueur moindre que les opérandes droits, le traitement est effectuée sur cette plus petite longueur. Si un opérande gauche d'affectation est de longueur plus importante que les opérandes droits, les éléments supplémentaires de cet opérande gauche sont affectés de la valeur de l'élément neutre de l'opération réalisée à droite.

Dans l'extension du langage C réalisée par Gisselquist sur Cray 2 [Giss86], le programmeur fixe la valeur du registre VL (vector length) au travers une macro qui est un appel de code assembleur. Gisselquist note qu'«une telle macro est peut-être suffisante». Elle nous semble de trop bas niveau.

Des règles précises permettant de déterminer et de spécifier la longueur sur laquelle traiter les vecteurs sont un besoin évident. De telles règles ont été mises en place en EVA. EVA propose trois règles dites *règles de longueurs*. Une des règles est associée à toute expression ou instruction vectorielle. L'application de la règle détermine la *longueur effective*, i.e. la longueur (nombre d'éléments de vecteur) sur laquelle sera effectuée l'opération. Cette association est soit implicite : la règle par défaut s'applique alors; soit explicite, c'est le programmeur qui indique la règle à appliquer. L'application d'une autre règle masque temporairement l'application de la règle courante. Nous définissons le concept de segment, détaillons les trois règles et introduisons les blocs de longueurs. Des exemples illustrent enfin l'application de ces règles.

2–7.1 La notion de segment

Un *segment* est une séquence de code sur laquelle s'applique l'une des règles des longueurs. On distingue donc trois types de segments, associé chacun à une des règles des longueurs :

- **Les segments à longueur par défaut** La longueur effective est la longueur du premier opérande vecteur du segment.
- **Les segments à longueur explicite** La longueur effective est spécifiée par la valeur d'une expression entre %.
- **Les segments à longueur minimum** La longueur effective est pour chaque opérateur le minimum des longueurs des opérandes vectorielles.

Une instruction est un arbre de segments. Cet arbre est construit selon les règles suivantes :

- une instruction est un segment,
- l'opérateur de description (les crochets [ ]) parenthèsent une liste de segments séparés par des virgules,

- un appel de fonction crée un segment pour chacun de ses paramètres,
- une expression parenthésée crée un segment pour l'évaluation de l'expression.

La plupart des opérateurs ne découpent pas l'expression qu'il compose en segments. Cependant les opérateurs de concaténation, de *merge*, de *mask* et d'association créent un segment pour chacun de leurs opérandes (figure 2–5).

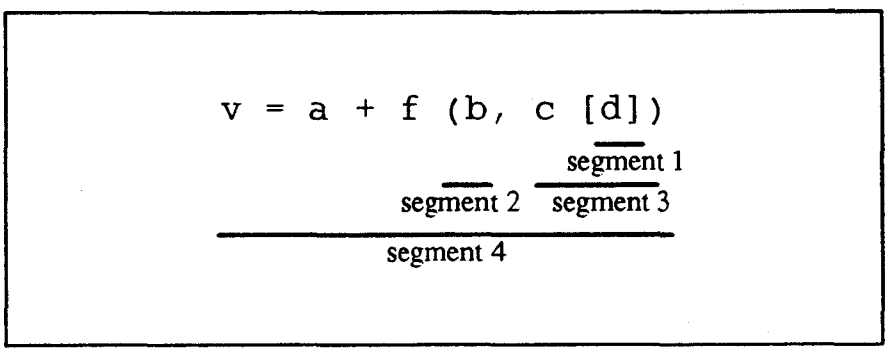


Figure 2–5. Les segments de longueur pour l'instruction
`v = a + f (b, c [d]).`

En l'absence de spécification, un segment est un segment par défaut. Un segment peut être préfixé par une spécification de segments entre `%`.

- Une expression entre `%` crée un segment à longueur explicite. La longueur effective est la valeur de l'expression.
- Une spécification `% < %` crée un segment à longueur minimale.
- Une spécification `% ? %` crée un segment à longueur par défaut.

2–7.2 Les blocs de longueur

Les blocs de longueur ont été introduits pour spécifier l'application d'une règle des longueurs pour une séquence d'instructions. Comme pour un segment, le bloc de longueur est préfixé par une spécification entre `%`. Exemple

```
% 1000 % {
      v = v1 + v2
      v3 = v4 [d1] * v5 [d2]
}
```

Toutes les instructions du bloc seront réalisés sur une longueur de 1000. La spécification d'une longueur unique pour un bloc d'instructions vectorielles autorise la mise en place d'optimisations telles la fusion de boucles (se référer au chapitre 1).

2–7.3 Entité vecteur

Nous définissons la notion d'*entité vecteur*. Une entité vecteur est un objet vecteur avec toutes les descriptions qui lui sont associées. Par exemple, soient v, w, d trois vecteurs, dans l'expression

```
v [w [d]] [d]
```

d est une entité vecteur (présente deux fois), *w* [*d*] et *v* [*w* [*d*]] [*d*] sont des entités vecteurs. Mais ni *v*, ni *w*, ni *v* [*w* [*d*]] ne sont des entités vecteurs. Ce sont les longueurs des entités vecteurs qui déterminent la longueur de traitement des instructions pour certaines règles (voir ci-après).

2–7.4 La règle de la longueur par défaut

Selon la règle de la longueur par défaut, la longueur effective est celle du premier opérande entité vecteur apparaissant dans le segment. Soient les déclarations

```
vect int v10 <- 10,
         v20 <- 20,
         v30 <- 30
```

v10, v20 et v30 sont donc respectivement des vecteurs de 10, 20 et 30 éléments. Dans l'instruction

```
v10 = v30 * v20 + v10
```

les opérations d'addition, de multiplication et d'affectation sont toutes réalisées sur 10 éléments.

2–7.5 La règle de la longueur explicite

Selon la règle de longueur explicite, la longueur effective est la valeur de l'expression entre `%` préfixant le segment (ou le bloc pour une instruction). Cette expression doit être une expression scalaire entière (non nécessairement évaluable à la compilation). Par exemple dans l'instruction

```
% 5 % v10 = v20 + v30
```

les opérations d'addition et d'affectation sont réalisées sur 5 éléments.

2–7.6 La règle du minimum

Selon la règle de la longueur minimum, la longueur effective est pour chaque opérateur du segment, le minimum des longueurs de ses opérandes entités vecteurs. Le code nécessaire sera généré pour calculer le minimum des longueurs pour chaque opération. Dans l'instruction

```
% < % v10 = v30 + v20
```

l'addition est réalisée sur 20 éléments, l'affectation sur 10.

2–7.7 L'attribut `'length`

L'attribut **`'length`** référence la longueur effective valide à l'endroit où l'attribut est utilisé. Par exemple

```
% f(n) % {
        unassoc vect int temp <- 'length
        ...
}
```

Déclare un vecteur local au bloc *temp* dont la taille de la zone d'allocation est la longueur effective de traitement des opérations vectorielles du bloc (ici *f(n)*).

2–7.8 Utilisation des spécifications de longueurs

L'avantage majeur des spécifications de longueur est de préciser la longueur effective sur laquelle sont réalisées les opérations. De plus, la spécification de la longueur implicite ou explicite (par le programmeur) clarifie la programmation, par exemple lors de descriptions complexes. Elle autorise parfois la suppression de certaines descriptions.

```
vect real v, w
vect int d <- dlength, e
```

Le vecteur *d* contient *dlength* éléments. Dans la majorité des langages, on doit écrire

```
v [d] = w [e] [1:dlength]
```

En EVA on évite une description par une spécification de la longueur

```
% dlength % v [d] = w [e]
```

Mais on peut aussi laisser agir la règle par défaut pour obtenir le même résultat :

```
v [d] = w [e]
```

Toutefois le principal avantage de ces spécifications est le suivant : lors de la spécification d'une longueur explicite pour un bloc d'instructions, la séquence d'instructions du bloc est exécutée sur une longueur fixe. Le découpage en boucle (strip-mining) est alors réalisable au niveau du bloc d'instructions et non pas au niveau d'une instruction. (Cependant, ce regroupement d'instructions dans une même boucle de strip-mining peut être interdit par d'éventuelles dépendances entres les instructions.)

2–8 L'expression des dépendances en EVA

En EVA, le parallélisme est essentiellement exprimé au niveau des données. Toutefois, certaines constructions du langage expriment un parallélisme au niveau des instructions. L'expression de ce parallélisme est assurée par le programmeur en terme de dépendances ou non-dépendances. Le langage EVA ne propose pas d'outils généraux d'expression du parallélisme de tâches. Si tel était le but recherché, le déclenchement de tâches écrites en EVA pourrait se faire au travers d'un ensemble de primitives ou par l'appel de sous-programmes EVA depuis un langage de gestion du parallélisme.

Les sous-sections suivantes présentent la sémantique habituelle de l'affectation vectorielle, les différentes affectations vectorielles de EVA, les spécifications de dépendances ou non-dépendances entre les

instructions EVA, et discutent enfin de la combinaison des dépendances entre parties droite et gauche d'affectations et des dépendances entre instructions.

2–8.1 Sémantique de l'affectation vectorielle

Une affectation est qualifiée de vectorielle lorsque sa partie gauche est un vecteur. La partie droite de l'affectation est soit scalaire, soit vecteur. La sémantique d'une affectation *V = Expr* est telle que cette opération d'affectation est équivalente aux affectations

TMP = Expr
V = TMP

Cette sémantique assure la prise en compte d'éventuelles dépendances entre parties droite et gauche de l'affectation. Soit le code

```
v [2 : 11] = v
```

Pour que cette opération ait un sens quel que soit l'ordre de traitement des éléments, il faut effectivement que l'affectation vectorielle se fasse en deux passes. (Le problème est identique pour une partie droite scalaire obtenue par description d'un vecteur. Par exemple *v = v [2].*)

Cette sémantique est celle de la majorité des langages vectoriels explicites (FORTRAN 90, CF77 sur machines Cray, CMF FORTRAN sur la Connection Machine).

2–8.2 Les affectations vectorielles EVA

EVA propose deux opérateurs d'affectations. La sémantique de l'opérateur d'affectation = prend en compte le passage par un temporaire entre la lecture de la partie droite et l'écriture de la partie gauche. EVA propose aussi un opérateur d'affectation : = qui évite le passage par une zone temporaire. La spécification de la dépendance ou de la non dépendance entre parties droite et gauche de l'affectation est donc réalisée par le programmeur au moyen des deux opérateurs d'affectation.

Par l'utilisation de l'opérateur =, le programmeur indique qu'aucun des éléments de vecteurs écrits dans la partie gauche de l'affectation ne sont lus dans la partie droite.

```
vect real   v <- 1000
      v1 <- v [1 :[2] *]
      v2 <- v [2 :[2] *]
      v3 <- v [1 :[3] *]
```

Pour ajouter la valeur des éléments pairs à celle des éléments impairs de *v*, on écrit

```
v1 += v2
```

Comme il n'existe pas de dépendances entre les les deux vecteurs *v1* et *v2*, on écrira plutôt

```
v1 +:= v2
```

Par contre le résultat de

```
v3 +:= v1
```

est imprévisible du fait de la dépendance entre les vecteurs *v1* et *v3*.

Ces informations de dépendances exprimées par le programmeur sont une aide précieuse lors de la génération de code, notamment pour des opérations telles les gather/scatter par une liste d'index pour lesquelles aucune reconnaissance automatique de non-dépendance n'est possible au sein de quelque système de vectorisation automatique que ce soit.

2–8.3 Dépendances de flot

Du point de vue du programmeur, il n'existe pas de parallélisme au niveau des instructions en EVA : le déclenchement d'une instruction n'est effectif que lorsque le traitement de l'instruction précédente est terminé. A l'inverse, et dans une même instruction, l'expression des dépendances de flot (par des opérations d'affectation) exprime un certain parallélisme. C'est le parallélisme de chaînage des pipelines des machines vectorielles pipelines. Les deux séquences de code

```
c := d + e
a := b + c
```

et

```
a := b + (c := d + e)
```

ne sont donc pas équivalentes; la seconde exprime plus de parallélisme. La reconnaissance de cette seconde forme à partir de la première pourrait être réalisée automatiquement; cependant notre approche en EVA est que le programmeur aide le compilateur qui ne met en place aucune optimisation de ce genre.

2–8.4 Dépendances inter-instructions

En EVA, les instructions sont exécutées les unes après les autres. Toutefois, le programmeur peut indiquer que deux instructions ne sont pas dépendantes en les séparant par le séparateur . . (point-point). Plusieurs instructions consécutives peuvent être séparées par des point–points, indiquant ainsi que ces instructions sont toutes indépendantes.

Par l'introduction d'un séparateur . . entre deux instructions, le programmeur indique que si un élément de vecteur est référencé en écriture par une des deux instructions, il ne l'est qu'une fois et il n'est pas référencé en lecture dans aucune des deux instructions. Nous précisons la sémantique de ce séparateur dans le cas général à la sous-section suivante.

Cette spécification de non-dépendance est prise en compte par le compilateur (ou le traducteur) pour déclencher les instructions en parallèle. De plus cette spécification autorise la fusion des boucles des instructions vectorielles.

Cette exécution en parallèle ne peut se faire qu'au niveau des instructions élémentaires EVA; il n'existe pas de parallélisme au niveau des tâches (appel de fonction ou bloc d'instructions).

2–8.5 Composition des dépendances entre instructions et des dépendances entre parties droite et gauche d'affectations

Nous donnons ici le sens précis des spécifications de non-dépendances entre instructions et de l'utilisation des affectations EVA = et : = ainsi que de la combinaison des deux.

Nous considérons une relation de précédence sur les objets référençant des éléments de vecteur (les *référants*). Cette relation, notée du signe <, est définie ainsi : soient deux instructions *S1* et *S2* telles que *S1* précède *S2* dans le flot d'instruction et que *S1* n'est pas séparée de *S2* par le séparateur `..`, alors tout référant appartenant à *S1* précède tout référant de *S2*. Pour une instruction d'affectation =*S*, tout référant de la partie droite de *S* précède tout référant de la partie gauche de *S*. Pour la séquence d'instruction

```
a = b + c ..
d [e] := f
g = h * i
```

on a : $b = c < a = f = d = e = h = i < g$

On considère alors les ensembles de référants égaux. Pour tout ensemble, on a la propriété suivante : un élément de vecteur référencé en écriture par un référant de l'ensemble ne l'est que par ce référant et n'est pas référencé en lecture aucun autre référant de l'ensemble.

Sur l'exemple précédent, le programmeur assure donc que (Ref (r) désigne l'ensemble des éléments référencés par le référant r)

$$\text{Ref}(a) \cap (\text{Ref}(d) \cup \text{Ref}(f) \cup \text{Ref}(e) \cup \text{Ref}(h) \cup \text{Ref}(i)) = \emptyset,$$
et $$\text{Ref}(d) \cap (\text{Ref}(a) \cup \text{Ref}(f) \cup \text{Ref}(e) \cup \text{Ref}(h) \cup \text{Ref}(i)) = \emptyset.$$

2–9 Les structures de contrôle

EVA est un langage vectoriel explicite. En aucun cas une structure de contrôle scalaire ne produira un code vectoriel. EVA propose une structure de contrôle spécifique au traitement vectoriel : le **`with`**. Nous rappelons les structures de contrôle qui ont été proposées par d'autre langage vectoriels, puis introduirons le **`with`** EVA. Nous expliquerons ensuite pourquoi nous nous sommes restreints à l'introduction du seul constructeur **`with`**.

2–9.1 Les structures de contrôle de la programmation vectorielle : extension des structures de contrôle scalaires

De nombreux langages vectoriels proposent une construction **`WHERE`** équivalente à un **`if`** vectoriel (se référer à la première partie de ce document). Certains langages étendent les structures de contrôle scalaires à des opérandes vectoriels. C'est le cas des langages ACTUS et POMPC. En ACTUS par exemple, les boucles PASCAL **`while`** dont le prédicat est un vecteur sont valides. Une boucle ACTUS telle

```
while a [1:50] < b [1:50] do
      a [#] := a [#] + 1
```

est exécutée tant qu'il existe un élément de *a* inférieur à *b*. Une itération de la boucle incrémente de 1 les valeurs de *a* inférieures à *b*. On pourrait écrire en ACTUS (sans exprimer le parallélisme)

```
while any (a [1:50] < b [1:50]) do
      for i := 1 to 50 do
      if a [i] < b [i] then
            a [i] := a [i] + 1
```

avec *any* une fonction retournant faux si et seulement si son opérande tableau ne possède aucun élément vrai. POMPC et MPL ont étendu toutes les constructions de contrôle de flot scalaire du langage C à des opérandes vecteurs (se reporter au chapitre 1). Quelques subtilités concernant la sémantique de telles constructions sont présentées dans la section 2–9.3.

2–9.2 La structure de contrôle vectorielle `with` de EVA

2–9.2.1 Présentation du bloc `with`

Lors de la conception de EVA, nous nous sommes inspirés du constructeur **`where`** généralement adopté et avons proposé un constructeur de bloc, le **`with`**. Un bloc **`with`** est un bloc d'instructions précisant une *description par défaut* pour toutes les instructions d'affectation vectorielle du bloc. La description par défaut est, comme toutes les descriptions EVA, une liste de valeurs booléennes, un triplet ou une liste d'index. La construction **`with`** est donc plus générale que les constructions **`where`** habituelles qui limitent la description à une liste de valeurs booléennes. Toute instruction est autorisée dans un bloc **`with`**. Cependant, seules les instructions d'affectation vectorielles sont touchées par la description par défaut; la sémantique des autres instructions est inchangée. Une instruction d'affectation vectorielle est une instruction d'affectation (= ou : =) dont la partie gauche est un vecteur.

Un bloc **`with`** est construit sur le modèle de l'exemple suivant :

```
vect real v1, v2, v3, w1, w2, w3
...
with [descr_expr]
      v1 = v2 [1:[2]*] + v3
      w1 = w2 = w3
      if (any (v1 > 0))
            puts ("positive value(s)")
            v1 += 1
      end
end
```

qui est équivalent à

```
{
      temp := descr_expr
      % temp 'length % {
            vl [temp] = (v2 [1:[2]*] + v3) [temp]
            wl [temp] = (w2 = w3) [temp]
            if (any (vl > 0))
                  puts ("positive value(s)")
                  vl [temp] += 1
            end
      }
}
```

La variable *temp* est une variable vecteur booléenne si l'expression *descr_expr* est booléenne, une variable vecteur d'entiers sinon. Les quelques points suivants appellent un commentaire :

- La description par défaut est appliquée à la partie gauche et droite d'une instruction d'affectation vectorielle si celle-ci est vecteur. Toutes les instructions d'affectation vectorielle sont traitées ainsi, qu'elles soient ou non incluses dans une autre structure de contrôle (`if` par exemple).
- Les instructions qui ne sont pas des affectations vectorielles sont autorisées dans un bloc **`with`**. Leur sémantique est inchangée qu'elles soient dans un bloc **`with`** ou non.

2–9.2.2 Imbrication des blocs `with`

L'imbrication des blocs **`with`** est possible en EVA. La description par défaut du bloc interne est le résultat de la description de l'expression de description du bloc **`with`** englobant par la description par défaut du bloc interne. Exemple :

```
with [pair]
      with [first_3]
            vl = v2 [1:[2]*] + v3
      end
end
```

est équivalent à

```
{     temp1 := pair
      % temp1 'length % {
            temp2 := pair [first_3]
            % temp2 'length % {
                  vl [temp2] = (v2 [1:[2]*] + v3) [temp2]
            }
      }
}
```

2–9.2.3 L'attribut `'with`

L'attribut **`'with`** ne peut être utilisé que dans un bloc **`with`**. La valeur retournée par cet attribut est la description par défaut applicable à l'endroit d'utilisation de l'attribut.

2–9.2.4 Le bloc `orwith`

Deux blocs **`with`** successifs peuvent être unifiés par un constructeur **`orwith`**. Par exemple

```
with [1 :[2] N]
      a = 3
orwith [2 :[2] N]
      b = 4
end
```

est équivalent à

```
with [1 :[2] N]
      a = 3
end with [2 :[2] N]
      b = 4
end
```

L'expression de description du **`orwith`** devient l'expression de description par défaut. L'expression de description du **`orwith`** peut référencer l'expression de description du bloc **`with`** (ou **`orwith`**) précédent par l'attribut **`'with`**; sa validité s'étendant jusqu'à la fin de l'expression par défaut du **`orwith`**. On peut donc écrire

```
with [a >= 0]
      a +:= 1
      with [a <= 10]
            a +:= 10
      end
orwith [not 'with]
      a +:= 2
end
```

ou

```
#define elsewith orwith [not 'with]
with [a >= 0]
      a +:= 1
      with [a <= 10]
            a +:= 10
      end
elsewith
            a +:= 2
      end
```

et ainsi retrouver l'équivalence du **`elsewhere`** de FORTRAN 90.

2–9.2.5 Expression de description par défaut instanciable

La sémantique d'un bloc **`with`** est légèrement différente si l'expression de description par défaut est une expression instanciable. Comme dans le cas général, toute affectation vectorielle est contrôlée par la description par défaut. Cette expression instanciable est évaluée/instanciée pour chaque description par défaut. Le gain de performances pouvant être obtenu par une telle construction est alors minime. Toutefois, toute sous-expression non-instanciable de l'expression instanciable est évaluée préalablement à l'exécution du bloc **`with`**.

2–9.3 D'autres structures vectorielles en EVA ?

Le **`with`** EVA est une extension du **`where`**, **`if`** vectoriel. D'autres langages généralisent les constructeurs de boucle à des opérandes vectorielles. EVA ne propose qu'un constructeur, le **`with`**. Nous expliquons ici les raisons de ce choix.

Nous prenons l'exemple d'une boucle **while** vectorielle. Nous donnons ici les diverses sémantiques qui peuvent être "naturellement" associées à une telle boucle. Nous les exprimerons en EVA. La boucle

```
while (Vbool)
      ...
end
```

est contrôlée par une expression à résultat booléen *Vbool*. Dans le corps de la boucle, des instructions vectorielles référencent des vecteurs. Un premier sens donné à la boucle est

```
/* equivalence 1 */
while (all (Vbool))
      ...
end
```

La boucle est itérée tant que tous les éléments de l'expression *Vbool* sont vrais; la sémantique des instructions composant la boucle est inchangée. On peut imaginer itérer tant qu'il existe un élément vrai :

```
/* equivalence 2 */
while (any (Vbool))
      ...
end
```

Les accès des vecteurs au sein du bloc **while** vectoriel peuvent être contrôlés par la valeur de *Vbool*; cette sémantique est celle de la boucle **whilesomewhere** de POMPC (chapitre 1) :

```
/* equivalence 3 */
while (any (Vbool))
      with [Vbool]
            ...
      end
end
```

Finalement, un élément non sélectionné pour une itération ne peut plus l'être pour les itérations suivantes; c'est la sémantique choisie pour la boucle **while** vectorielle de MPL (chapitre 1) :

```
/* equivalence 4 */
Vdescr := Vbool
while ((any (Vdescr))
      Vdescr &:= Vbool
      with [Vdescr]
            ...
      end
end
```

Nous avons donc présenté 4 alternatives à la sémantique d'une boucle **while** vectorielle. Si la première alternative est écartée (car offrant une construction bien "pauvre"), les trois autres semblent acceptables. L'introduction d'un constructeur dont la sémantique n'est pas clairement et aisément identifiable et dont les diverses équivalences sont facilement exprimables en EVA ne nous a donc pas paru indispensable.

2–10 Fonctions et passage de paramètres vecteurs

EVA est un langage procédural. Un programme EVA est un ensemble de modules. Un module est une unité de compilation. On distingue les modules EVA et les modules externes (C ou FORTRAN par exemple). Un module EVA est une suite de définitions de variables globales et de fonctions. Nous nous intéressons ici aux fonctions EVA. Les caractéristiques de ces fonctions sont en de nombreux points similaires aux fonctions des langages habituels. Cette section présente les seules particularités de EVA.

Une fonction EVA est déclarée externe, globale ou locale. Les fonctions externes sont des fonctions définies dans un autre module et qui sont visibles dans le module. Les fonctions globales et locales sont définies dans le module. Seules les fonctions globales sont "exportées" donc appelables depuis les autres modules. Les fonctions locales (mot clé `local`) ne sont utilisables que dans le module au sein duquel elles sont définies. Cela est un avantage certain pour le programmeur. De plus, lors de la compilation il est possible d'analyser le code d'une fonction locale pour mettre en place des optimisations. Par exemple déterminer précisément les caractéristiques des paramètres (attributs des vecteurs par exemple; cf. la section 2–14.3 pour plus de détails) et générer différentes versions pour différents appels.

2–10.1 Définition de fonctions EVA

Une fonction EVA peut retourner une valeur. Le type et le constructeur (scalaire ou vecteur) de cette valeur est déterminé par la définition de la fonction. Une fonction accepte une liste de paramètres en entrée; chaque paramètre est scalaire ou vecteur. Le passage des paramètres est discuté dans les sections suivantes.

2–10.2 Allocation des vecteurs locaux à une fonction

Une fonction peut définir des objets locaux. Ces objets vecteurs et scalaires sont alloués automatiquement lors de chaque activation de la fonction et désalloués à chaque terminaison. Pour un vecteur, seule l'allocation de l'en-tête est automatique; l'allocation des zones d'allocation et de description est réalisée via des opérations d'association. Cependant, les zones ainsi allouées seront automatiquement libérées par EVA lors de la terminaison de la fonction. Notons aussi que les zones encore référencées par un autre vecteur ne seront pas libérées. Exemple :

```
vect real vglobal1, vglobal2
...
f ()
      vect real vlocal1, vlocal2, vlocal3
      vlocal1 <- 100
      vlocal2 <- 1000
      vglobal1 <- vlocal2
      vlocal3 <- vglobal2
end
```

Pour les trois vecteurs locaux, seule la zone d'allocation du vecteur *vlocal1* est effectivement libérée à la fin de la fonction. La zone d'allocation de *vlocal2* est encore référencée par le vecteur global *vglobal1* et

vlocal3 référence la zone d'allocation du vecteur *vglobal1*; ces deux zones ne seront donc pas libérées. Cette gestion des zones d'allocation est réalisée par un compteur de références associé à chaque zone; se reporter à la description de la représentation des vecteurs dans la seconde partie de ce chapitre pour plus de précisions.

2–10.3 Appel de fonction EVA—Passage de paramètres

Nous discutons ici des particularités de EVA en ce qui concerne les appels de fonctions et le mécanisme mis en place pour le passage des paramètres. Les paramètres effectifs d'une fonction sont des expressions EVA. Ces expressions sont évaluées. Le résultat de cette évaluation est passée à la fonction appelée.

Les passages de paramètres se font par adresse entre les fonctions EVA. Toutefois des passages par valeur ont été définis et sont réservés à l'appel de fonction C depuis EVA (se référer à la section 2–12.2, Interface avec C et FORTRAN).

Un paramètre formel vecteur sera associé à un en-tête de vecteur. Un tel en-tête est essentiellement constitué d'une référence sur la zone d'allocation et d'une référence sur la zone de description (figures de la page 82). (De plus amples détails sont fournis sur les en-têtes de vecteurs dans la seconde section de ce chapitre.) Cet en-tête est celui d'une variable vecteur paramètre effectif ou un en-tête *temporaire* créé pour la circonstance.

Nous détaillons en premier lieu le passage d'expressions non réduites à un nom de variable; nous traitons ensuite le passage des variables et des opérations réalisées lors du passage de variables déclarées constantes ou à association unique.

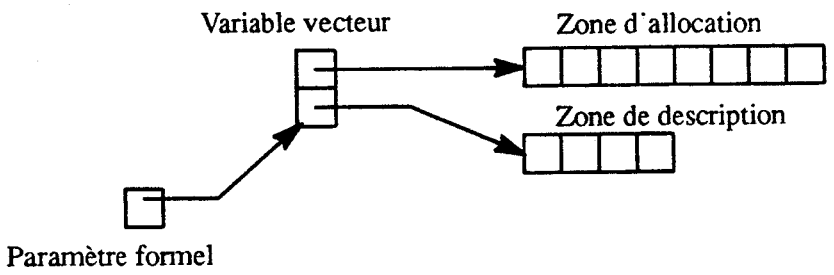
2–10.3.1 Passage d'expressions en paramètre

Une expression paramètre effectif non limité à un nom de variable est évaluée. Le résultat est stocké dans une zone temporaire. Pour une expression vecteur, un en-tête fictif référençant cette zone comme zone d'allocation et de zone de description vide sera passé à la fonction appelée et référencé via le paramètre formel. Le passage de vecteurs temporaires est ainsi unifié avec le passage de variables vecteurs. Les zones temporaires sont automatiquement libérées lors du retour de fonction.

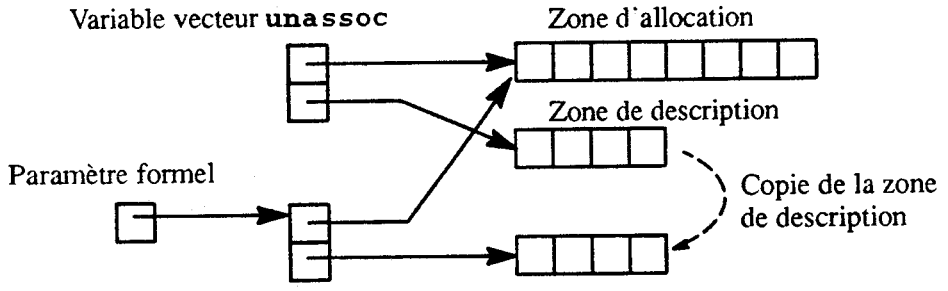
Le passage d'une expression réduite à un nom de vecteur décrit est un cas particulier de passage d'expression en paramètre; la description sera consommée et le paramètre formel ne partagera pas les zones d'allocation et de description du vecteur initial. Exemple :

```
maj (vecteur)
      vect real vecteur
      vecteur += 1
end
...
maj (v [1:5])
```

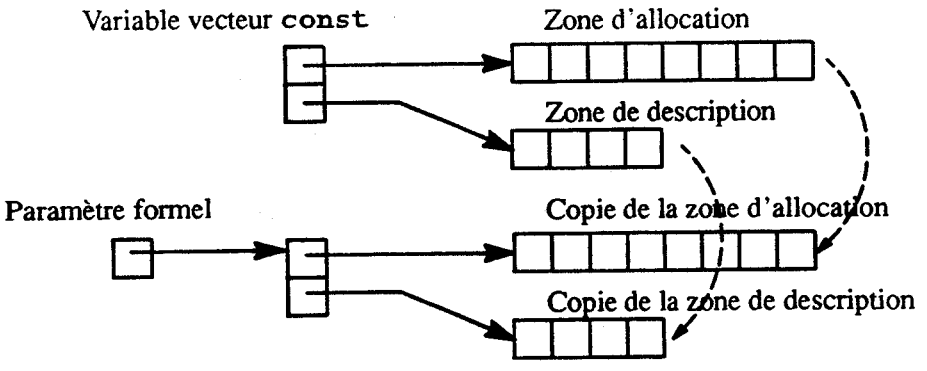
L'appel de la fonction *maj* est sans effet sur les valeurs de *v*. La mise à jour du vecteur décrit passé en paramètre est impossible de cette manière. Nous revenons sur ce problème à la section 2–10.3.5.



a—Passage d'une variable vecteur simple.



b—Passage d'une variable vecteur à association unique.



c—Passage d'une variable vecteur déclarée constante.

Figure 2–6. Passage de paramètres vecteurs.

2–10.3.2 Passage de variables simples en paramètre

Nous présentons ici le passage de paramètre variable vecteur. Les variables vecteurs sont traitées différemment selon leur classe de modification (variable, constante et à association unique). Nous présentons le passage de variables vecteurs simples.

Dans ce cas, le paramètre formel vecteur partagera l'en-tête du vecteur passé en paramètre (figure 2–6.a). Les conséquences sont claires : 1—les modifications des valeurs de la zone d'allocation du paramètre formel (par affectation) sont répercutées au paramètre effectif, 2—les modifications de l'en-tête du paramètre formel (par association) sont répercutées au paramètre effectif. Il est ainsi possible d'appeler une fonction qui réalise l'allocation de son paramètre vecteur :

```
allocation_bornee (vecteur, taille)
      vect real vecteur
      int taille
      if (taille < 1000)
            vecteur <- taille
      else
            vecteur <- 1000
      end
end
...
vect real V
int ma_taille
...
allocation_bornee (V, ma_taille)
...
```

2–10.3.3 Passage de variable vecteur à association unique

Les variables à association unique ne peuvent pas être associées en dehors de leur déclaration. Le passage d'une telle variable en paramètre garde cette propriété.

Une copie de l'en-tête du vecteur paramètre est réalisée. Cette copie référence la zone d'allocation du vecteur paramètre effectif et une copie de la zone de description de ce vecteur. Cette copie est passée à la fonction et référencée via le paramètre formel (figure 2–6.b). Toute affectation du paramètre formel est ainsi répercutée au paramètre effectif. A l'inverse, une association du paramètre formel n'est pas répercutée au paramètre effectif. Les caractéristiques **unassoc** du paramètre effectif sont ainsi conservées. Une fonction peut donc réaliser une initialisation d'un vecteur à association unique.

```
initialisation (vecteur)
      vect real vecteur
      vecteur = 0.0
end
...
unassoc vect real V <- 1000
...
initialisation (V)
...
```

2–10.3.4 Passage de variable déclarée constante en paramètre

Les variables vecteurs déclarées constantes ne peuvent être ni affectées, ni associées en dehors de leur déclaration. Un mécanisme de recopie est donc mis en place lors du passage de paramètre pour conserver l'intégrité de ces variables.

Considérons le passage en paramètre d'une variable vecteur déclarée constante. Une copie des zones d'allocation et de description de cette variable est réalisée. Ces copies sont référencées par un en-tête temporaire qui est passé à la fonction (figure 2–6.c). Ainsi, les opérations d'association et d'affectation sur le paramètre formel ne sont pas répercutées sur le paramètre effectif. Au retour de la fonction, les zones temporaires allouées pour les copies sont libérées. Le mécanisme est identique à celui mis en place pour le passage d'une expression vecteur non réduite à un nom de variable vecteur.

2–10.3.5 Passage de sous-vecteurs à une fonction

Nous avons déjà abordé le problème du passage d'une "tranche" d'un vecteur EVA à une fonction. Nous donnons les méthodes à utiliser pour réaliser une telle opération plus générale qui est la mise à jour d'un sous-ensemble des éléments d'un vecteur EVA par une fonction.

Rappelons que le passage d'une expression réduite à un nom de vecteur décrit est un cas particulier de passage d'expressions en paramètre. La description sera consommée et le paramètre formel ne partagera pas les zones d'allocation et de description du vecteur. Soit la fonction

```
maj (vecteur)
      vect real vecteur
      vecteur +:= 1
end
```

Les éléments d'un vecteur *V* sont incrémentés par

```
vect real V
maj (V)
```

A l'inverse, un appel semblable à

```
vect real V
vect int I
...
maj (V [I])
```

n'a aucun effet sur les éléments de *V*. La mise à jour d'un sous-ensemble des éléments est réalisée par une association préliminaire au passage de paramètre :

```
vect real V, VdeI
vect int I
...
VdeI <- V [I]
maj (VdeI)
```

Le vecteur *VdeI* est constitué des éléments de *V* décrit par la liste d'index *I*. La fonction *maj* incrémente donc ces éléments; nous obtenons le résultat désiré.

2–10.4 Retour d'une valeur vecteur

Une fonction EVA retourne une valeur scalaire ou un vecteur. Le retour de scalaire est traditionnel. Nous discutons ici du retour d'une valeur vecteur.

Seul un vecteur temporaire peut être retourné par une fonction EVA. Toutefois, au niveau du programmeur, tous les types de vecteurs peuvent être retournés. Ces objets seront transformés en vecteurs temporaires avant le retour effectif. La description implicite de la zone d'allocation par la zone de description d'un vecteur est ainsi consommée lorsque ce vecteur est retourné par une fonction.

```
vect real
      v <- 1 : 5
      vdecr <- v [b'01011']
vect real fnct ()
      ...
      return vdecr
end
```

La fonction *fnct* retourne un vecteur temporaire constitué des éléments 2, 4 et 5. Il est aussi possible de retourner un vecteur alloué localement à une fonction :

```
vect real allocation ()
      vect real vecteur_local <- ...
      ...
      return vecteur_local
end
```

Une copie du vecteur local est réalisée et retournée par la fonction. (Cette copie n'a pas à être effectivement réalisée si la zone d'allocation du vecteur local n'est partagée par aucun autre vecteur et que la zone de description est vide; il suffit de retourner la zone d'allocation comme vecteur temporaire résultat de la fonction.)

2–10.5 Les fonctions prédéfinies EVA

EVA fournit un ensemble de fonctions prédéfinies. On distingue

- les fonctions mathématiques (**abs**, **log**, etc...), trigonométriques (**sin**, **cos**, etc...), et de conversion (en une valeur réelle, entière, ..) qui acceptent des paramètres scalaires et vecteurs;
- les fonctions de manipulation de vecteur d'entiers en tant que chaîne de caractères;
- les fonctions de réduction (somme des éléments, valeur du maximum des éléments, ...) qui retournent une valeur scalaire à partir d'un opérande vecteur;
- les fonctions d'accès aux caractéristiques d'un opérande vecteur qui, par exemple, retournent une valeur booléenne indiquant si un vecteur est décrit, ou retournent le nombre d'éléments du vecteur.

Ces fonctions génèrent, pour la grande majorité, du code en ligne. La plupart des fonctions correspondent à une instruction du langage intermédiaire DEVIL.

2–11 Autres constructions et caractéristiques de EVA

2–11.1 Les multi-vecteurs

Un multi-vecteur est un objet autorisant la manipulation, la référence de plusieurs vecteurs. Ces objets n'ont été introduits dans le langage qu'à cette seule fin. Leur fonctionnalité est limitée à la définition (et éventuellement à l'allocation et l'initialisation) simultanée de plusieurs vecteurs ainsi qu'à la référence par un index entier d'un des vecteurs composant le multi-vecteur (via l'opérateur d'indiçage de multi-vecteurs `( )`). La taille, c'est-à-dire le nombre de vecteurs composant le multi-vecteur est fixe et doit être connue à la compilation. Les multi-vecteurs ne peuvent pas être passés par paramètre ni retournés par des fonctions. (Cependant, ces opérations sont possibles sur un des vecteurs formant un multi-vecteur.)

2–12 Interface avec les autres langages

EVA est un langage vectoriel. A ce titre EVA n'inclut pas certaines des caractéristiques des langages de programmation habituels tels les pointeurs ou la gestion des entrées/sorties (gérées par une bibliothèque spécialisée). Cependant une interface avec les autres langages (principalement C et FORTRAN) dote EVA de l'aspect universel de ces langages. Nous détaillons ici les aspects de EVA facilitant cette interface. Deux points sont concernés : les appels de fonction entre EVA et les autres langages et le partage de variable entre des modules EVA et des modules C ou FORTRAN.

L'accès à des modules et bibliothèques existants est une préoccupation importante des programmeurs utilisateurs de machines vectorielles. Il n'est pas envisageable de réécrire les nombreuses lignes de code existantes dans un nouveau langage.

2–12.1 Les variables comme interface

Il est possible de partager des données entre des modules EVA et des modules d'autres langages.

EVA définit des zones dites **common** pour l'utilisation partagée de variables entre deux modules. Les **common** avec des modules C ou EVA contiennent une variable, alors elle-même qualifiée de variable **common**. Les **common** avec un module FORTRAN peuvent contenir plusieurs zones. Une telle zone est référencée par un vecteur EVA comme une zone d'allocation et par FORTRAN comme un tableau.

Un fichier **`#include`** définit les types vecteurs EVA et l'interface à une bibliothèque de manipulation de vecteurs EVA depuis C (réduction des vecteurs à des tableaux, création de vecteurs à partir de tableaux, etc..).

2–12.2 Les fonctions comme interface

Les fonctions EVA peuvent être appelées depuis un autre langage. La prise en charge de cette caractéristique est quasi immédiate. Un module EVA peut déclarer des fonctions externes C ou

FORTRAN. Un mécanisme particulier est alors mis en place pour aider de telles déclarations et utilisations.

Exportation de fonctions EVA vers d'autres langages

Une fonction EVA non définie `local` est exportée par le module la définissant. Il est alors possible d'appeler la fonction en question depuis un autre module EVA, C ou FORTRAN. Toutefois, les paramètres de ces fonctions appelantes devront être passés par référence.

Appel de fonctions externes en EVA

Une déclaration **`extern`** autorise l'appel de fonctions définies dans un autre module. Ce module peut être un module EVA ou un module d'un autre langage. Comme ces autres langages mettent en place différents types de passage de paramètres et de retour de fonction, la syntaxe des déclarations de fonctions **`extern`** prend en compte ces particularités. Sans indication, une fonction **`extern`** passe ses paramètres par référence comme le font les fonctions EVA.

Il est possible, pour chaque paramètre d'une fonction déclarée **`extern`** de préciser s'il doit être passé par référence ou par valeur. Les appels depuis EVA prennent ensuite ces informations en compte. Le passage d'un paramètre vecteur par valeur réalise une copie des zones d'allocation et de description du vecteur et référence ces copies dans un en-tête qui est passé à la fonction. Rappelons que ce type de passage de paramètre ne peut avoir lieu entre fonctions EVA.

Chapitre 2 — Section 2

Implantation de EVA sur machine vectorielle pipeline

La première partie de ce chapitre présentait le langage EVA. Nous avons décrit les constructions proposées au programmeur et justifié leur existence par des considérations de programmation au niveau du parallélisme de données. La proposition d'outils de programmation qui ne saurait être implantés de manière efficace est sans intérêt dans la cadre de la programmation vectorielle. Cette seconde section du chapitre présente l'implantation de EVA sur une machine vectorielle pipeline. Nous présenterons ce en quoi les constructions proposées par EVA permettent au programmeur d'exprimer les connaissances sur son programme et comment ces informations sont utilisées par le compilateur EVA pour améliorer le code produit. Nous présenterons le langage DEVIL (langage intermédiaire vectoriel utilisé par EVA) et la compilation des constructions EVA en DEVIL. Nous discuterons ensuite de l'obtention effective de performances intéressantes à partir de programmes EVA; nous nous appuierons sur l'implantation de DEVIL réalisée sur Cray Y-MP.

Dans la suite de ce chapitre, nous distinguons les termes *compilateur* et *traducteur*. Le compilateur est chargé de la traduction de code EVA en DEVIL. Le traducteur génère du langage machine ou de l'assembleur à partir du code DEVIL.

2–13 DEVIL—Un langage vectoriel intermédiaire

DEVIL est le langage intermédiaire utilisé par EVA. Nous présentons ici DEVIL et sa machine virtuelle associée MAD. La thèse de Philippe Preux [Preu91], ainsi que [Preu90] et [DMP90b] fournissent une description plus complète de DEVIL et MAD. Nous ne donnons ici qu'un aperçu de DEVIL et MAD.

Comme pour tous les langages intermédiaires, deux points justifient essentiellement que la compilation de EVA se fasse via l'intermédiaire de DEVIL :

1 Le développement de compilateurs pour différentes cibles est facilité. Seule la dernière phase de compilation (implantation de DEVIL) est à écrire pour porter EVA sur une nouvelle cible.

2 Les optimisations indépendantes de la cible peuvent être réalisées en dehors de toute implantation effective du compilateur, soit lors de la génération de code DEVIL depuis EVA, soit par un optimiseur indépendant produisant du "DEVIL optimisé" à partir d'un code DEVIL existant.

DEVIL pourrait n'être alors qu'un langage intermédiaire correspondant à une représentation interne entre deux phases du compilateur. Nous avons préféré donner une forme lisible aux programmes DEVIL. Nous distinguons ainsi nettement les deux phases de compilation de EVA en DEVIL et de traduction de DEVIL en assembleur ou langage machine de la cible. DEVIL est ainsi un langage intermédiaire potentiel pour d'autres langages vectoriels ou pour des systèmes de vectorisation de programmes scalaires.

2–13.1 MAD—La machine virtuelle de DEVIL

La machine MAD est constituée d'un processeur et d'une mémoire. Le processeur réalise les traitements; la mémoire contient le code et les données. Un certain nombre de registres définissent aussi l'état de la machine. La mémoire est divisée en segments. On distingue entre autre :

- *code*, le segment de code;
- *data*, le segment de données; il contient les données statiques (globales à toutes les fonctions d'un module) et les données rémanentes (locales à une fonction dont la valeur est gardée d'un appel à l'autre);
- *param_to_send* et *param_rec*, deux segments de paramètres : le segment des paramètres des fonctions appelées par la fonction courante et le segment des paramètres de la fonction courante;
- *local*, le segment des variables locales de la fonction courante;
- *seg_tmp*, le segment des valeurs temporaires;
- *vlg*, le segment-pile associé au registre VLG. Le registre VLG détermine la longueur sur laquelle une opération vectorielle est exécutée;
- *heap*, ce segment contient les zones d'allocation et de description allouées dynamiquement par DEVIL.
- des segments commons; ils contiennent les variables partagées par plusieurs modules.

Les adressages dans ces segments sont résumés à la figure 2–7.

On remarquera que la mémoire de MAD n'est pas hiérarchisée; l'implantation de cette mémoire sur la mémoire effective de la machine cible est à la charge du traducteur. La mémoire de MAD contient des scalaires et des vecteurs.

Le processeur traite les scalaires et les vecteurs. Les unités fonctionnelles existent en nombre virtuellement infini dans MAD. La gestion du parallélisme est donc reporté sur le traducteur; le compilateur est autorisé à déclencher des instructions en parallèle sans se soucier de la disponibilité des unités fonctionnelles.

Le modèle de fonctionnement défini pour MAD est indépendant de toute architecture existante. La projection de MAD sur toute une gamme de machines est donc envisageable : machine à pile, machine mémoire à mémoire, ou machine à registres; machine scalaire, machine vectorielle type pipeline, ou machine tableau; machine à mémoire partagée, voire machine à mémoire distribuée.

2–13.2 Aperçu du jeu d'instructions de DEVIL

Pour l'essentiel, DEVIL est un langage à trois adresses; la gestion des temporaires est construite sur le modèle des langages de triplets. Nous ne détaillons pas ici le jeu d'instructions de DEVIL, mais présentons les grandes familles d'instructions. Nous distinguons 4 familles.

- Les pseudo-instructions. Elles définissent des symboles.
- les instructions scalaires. Elles traitent des objets scalaires; elles sont réparties en deux groupes : les instructions opératoires (instructions arithmétiques et d'affectation) et les instructions de contrôle de la machine MAD (saut, appel de fonction, gestion du registre VLG).
- les instructions vectorielles. Elles traitent les vecteurs; elles sont réparties en trois groupes : les instructions calculatoires, les instructions d'accès et manipulation de vecteurs (affectation, gather/scatter, compress/extend, etc...), les instructions de construction de vecteur (association, allocation, initialisation).
- les instructions de réduction de vecteurs. Elles produisent un résultat scalaire à partir d'opérande(s) vecteur(s). Leur résultat dépend des valeurs du vecteur (somme des éléments par exemple) ou des caractéristiques du vecteur (nombre d'éléments par exemple).

A chaque opérande d'une instruction DEVIL sont associés des attributs. Les attributs de constructeur (scalaire ou vecteur) et de type (entier, flottant, etc...) doivent être précisés. (D'autres attributs optimiseront le code généré (cf. 2–14.3)).

Pour des raisons d'orthogonalité des instructions de DEVIL, celles-ci acceptent des opérandes vecteurs ou scalaires et ce de tous les types DEVIL. Une instruction ne peut aisément être rangée dans l'une des familles distinguées ci–dessus. Cependant, une utilisation particulière d'une instruction (i.e. une instruction à laquelle sont associés les attributs de ses opérandes) se range sans ambiguïté dans l'une des 4 familles.

Les instructions opérant sur des vecteurs sont généralement exécutées sur VLG éléments. La majorité des instructions peuvent produire un résultat temporaire (en plus d'un éventuel résultat dans un opérande cible; ce temporaire en est alors une copie). Dans le cas où le temporaire produit est un vecteur, il compte VLG éléments.

Les conversions de types sont prises en charge par DEVIL à partir de la spécification des types des opérandes source et cible. Le type d'une opération générant uniquement un résultat dans un temporaire est déterminé par DEVIL.

Les instructions opérant sur des vecteurs peuvent être décrites. Le rangement du résultat est alors réalisé sous le contrôle d'un descripteur (vecteur de booléens, liste d'index ou triplet) sélectionnant les éléments de l'opérande cible à affecter.

mode d'adressage	adressage	segment
adressage immédiat	`#`*valeur*	—
adressage littéral	`#"`*valeur littérale*`"`	—
adressage global	`glo@(`*déplacement*`)`	*data*
adressage local	`loc@(`*déplacement*`)`	*local*
adressage paramètre	`par@(`*déplacement*`)`	*param_rec*
adressage common	*nom du common*	commons
ou	*nom du common*`@(`*déplacement*`)`	
temporaire non ré-utilisé	`&`*valeur*	*seg_tmp*
temporaire ré-utilisé	`^`*valeur*	*seg_tmp*
étiquette	`"`*étiquette*	*code*

valeur	valeur entière constante positive
valeur littérale	valeur d'un littéral (chaîne de caractères)
déplacement	valeur entière constante positive (en mot mémoire)
nom du common	chaîne de caractères, nom du segment common
étiquette	chaîne de caractères

Figure 2–7. Les modes d'adressage de DEVIL.

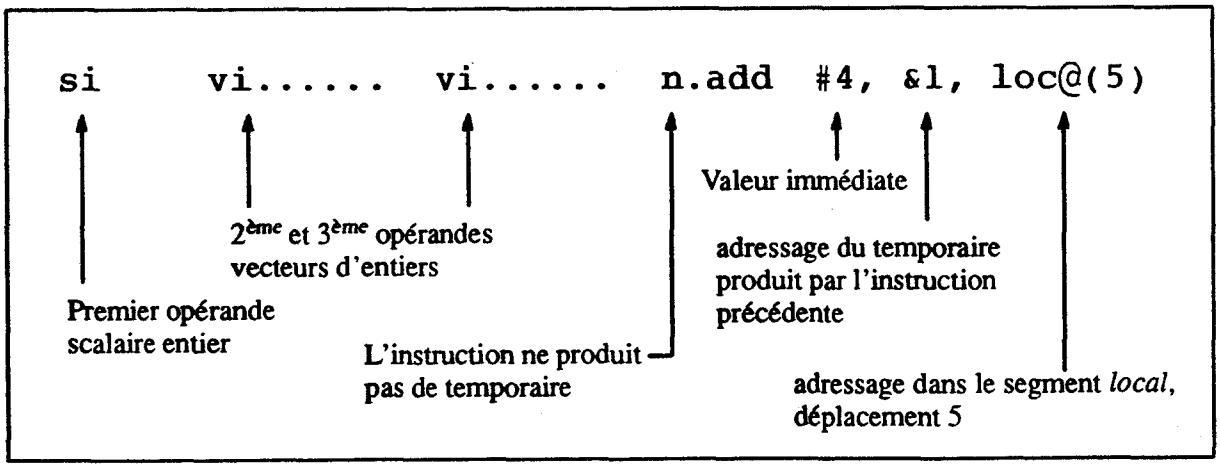


Figure 2–8. Format d'une instruction DEVIL.

2–13.3 Exemple de programme EVA/DEVIL

Nous présentons un exemple de programme DEVIL obtenu par le traduction d'un source EVA et les commentons. (La numérotation des lignes de codes est ajoutée pour les besoins de l'explication.)

2–13.3.1 Le programme EVA (figure 2–9)

Le module EVA est formé de deux fonctions (*main*, le point d'entrée du programme, et *Newton*).

Ligne 4. Déclaration d'une fonction externe retournant une valeur entière lue sur l'entrée standard dans son unique paramètre. (Bibliothèque EVA/DEVIL).

Lignes 8 à 18. Définition de la fonction *Newton*. Le vecteur *coeff* est local à la fonction; il est alloué dynamiquement (ligne 10) puis initialisé (lignes 12 et 13).

Ligne 15. Deux remarques :
1—La règle de la longueur par défaut s'applique; toutes les opérations sont donc réalisées sur la longueur de la partie gauche, soit *i*–2+1. Ainsi, la longueur de la partie droite n'a pas à être spécifiée (on évite une description).
2—L'opérateur d'affectation += force l'évaluation de la partie droite dans un temporaire préalablement à l'affectation à la partie gauche; ce passage par un temporaire protège ainsi les dépendances entre parties droite et gauche.

Ligne 17. L'instruction `return` retourne le vecteur *coeff* comme résultat de la fonction.

Lignes 20 à 27. La fonction *main* est le point d'entrée du programme. Les variables *degre* et *result* sont locales à cette fonction.

Ligne 25. La fonction *Newton* retourne un vecteur qui est associé au vecteur *result*. Le vecteur retourné est un vecteur temporaire. Il est constitué d'une séquence d'éléments rangés dans une zone allouée dynamiquement par la fonction. Cette séquence d'éléments devient la zone d'allocation de *result*; sa zone de description est vide.

2–13.3.2 Le code DEVIL correspondant (figure 2–10)

Nous présentons maintenant le résultat de la compilation de ce programme en DEVIL. Notons la structure d'une instruction DEVIL. Les premiers champs sont les attributs des opérandes; ils définissent principalement le type (entier, réel, ..) et le constructeur (scalaire, vecteur) des opérandes. Le code de l'opération est parfois précédé par un attribut `n..` Cela indique que l'instruction ne produit pas de temporaire. Dans le cas contraire, l'instruction produit une valeur temporaire. Suivent les opérandes; les différents adressages ont été résumés dans la figure 2–7. Le format d'une instruction DEVIL est repris dans la figure 2–8.

Ce programme DEVIL est divisé en 4 parties. La ligne 1 spécifie la taille du segment *data* (ici 0). Les lignes 2 à 31 correspondent à la fonction *Newton*, celles 32 à 43 à la fonction *main*. Les instructions des dernières lignes sont la zone d'initialisation des variables globales et rémanentes (segment *data*); dans cet exemple, il n'y a pas de variable dans ce segment *data*.

```
1  : -- Calcul des coefficients du binome de Newton,
2  : -- methode du triangle de pascal
3  :
4  : extern geti ()
5  :
6  : -- Retourne un vecteur contenant les coefficients du binome
7  : -- de Newton de degre n
8  : vect int Newton (n)
9  :      int n
10 :      vect int coeff <- n
11 :
12 :      coeff = 0
13 :      coeff [1] = 1
14 :      for i = 2, n
15 :           coeff [2:i] += coeff
16 :      endfor
17 :      return coeff
18 : end
19 :
20 : main ()
21 :      int degre
22 :      vect int result
23 :
24 :      geti (degre)
25 :      result <- Newton (degre)
26 :      -- etc...
27 : end
```

Figure 2–9. Exemple de programme EVA.

```
1  : ........  si                  n.const   size_static,  #0
2  : Newton:: Newton_loc, Newton_par
3  : vi......                      n.vinit   loc@(0)
4  : si        vi......            n.alloc   par@(0),  loc@(0)
5  : vie.n...                      n.pushvlg loc@(0)
6  : si        vie.n...            n.move    #"0",     loc@(0)
7  :                               n.popvlg
8  : si        si       vie.n...   n.scat    #"1",     #"1",     loc@(0)
9  : si        si                  n.move    #"2",     loc@(6)
10 : si        si                  n.move    par@(0),  loc@(7)
11 : LP0
12 : si        si                    gt      loc@(6),  loc@(7)
13 : sb        ........            n.branch  &1,       LE0
14 : si        si       si           sinit   #"2",     loc@(6),  #"1"
15 : visln...                      n.pushvlg ^1
16 : vie.n...  visln...              gath    loc@(0),  ^2
17 : vi..n...  vie.n...              add     &1,       loc@(0)
18 :                               n.wait
19 : vi..n...  visln... vie.n...   n.scat    &2,  &5,  loc@(0)
20 :                               n.popvlg
21 : si        si       si         n.add     loc@(6),  #1,  loc@(6)
22 : ........                      n.jmp     LP0
23 : LE0
24 : vie.n...                      n.pushvlg loc@(0)
25 : vie.n...                        move    loc@(0)
26 :                               n.popvlg
27 : vie.n...                      n.free    loc@(0)
28 : vi..n...                        return  &3
29 : ........  si                   n.const   Newton_loc,   #8
30 : ........  si                   n.const   Newton_par,   #1
31 :                               n.endfct
32 : main:: main_loc, main_par
33 : vi......                      n.vinit   loc@(1)
34 : si        si                  n.parout  loc@(0),  #0
35 : si                            n.call    geti
36 : si        si                  n.parout  loc@(0),  #0
37 : si                              call    Newton
38 : si        vi......            n.alloc   &1, loc@(1)
39 :                               n.wait
40 : vie.n...                      n.free    loc@(1)
41 : ........  si                   n.const   main_loc, #7
42 : ........  si                   n.const   main_par, #2
43 :                               n.endfct
44 :                               n.debistat
45 : si                            n.extern  geti
46 :                               n.finistat
```

Figure 2–10. Code DEVIL résultant de la compilation du programme de la figure 2–9.

La compilation réalisée ici suppose qu'une valeur scalaire est stockée sur un mot mémoire et un en-tête de vecteur sur 6 mots.

Fonction *Newton*

Ligne 2. Début de la fonction *Newton*. *Newton_loc* et *Newton_par* sont respectivement les tailles des segments *local* et *param_rec* de cette fonction; leur valeur est définie par la pseudo instruction **const** (lignes 29 et 30). Le segment *local* contient l'en-tête du vecteur *coeff* (**loc@(0)**), la variable scalaire de boucle *i* (**loc@(6)**) et une variable générée par le compilateur (**loc@(7)**) soit les 8 mots mémoire. Le segment *param_rec* contient le paramètre *n* (**par@(0)**) soit un mot.

coeff	**loc@(0)**
i	**loc@(6)**
variable compilateur	**loc@(7)**
n	**par@(0)**

Lignes 3 et 4. L'instruction **vinit** initialise le vecteur **loc@(0)**; l'instruction **alloc** associe ce vecteur avec une zone de taille **par@(0)** (association dynamique). (Toutefois, les attributs du vecteur **loc@(0)** indique que ce vecteur n'est pas allouée, l'instruction **alloc** ne produira donc pas de test de désallocation des zones d'allocation et de description.)

Lignes 5 à 7. L'opération utilise la règle de la longueur par défaut. L'instruction **pushvlg** empile la longueur du vecteur sur la pile des VLG. Le **move** transfert la valeur 1 sur les VLG premiers éléments du vecteur **loc@(0)**. Le **popvlg** dépile le sommet de la pile des VLG.

Ligne 8. L'instruction **scat** range la valeur de son premier opérande dans le vecteur troisième opérande; ce rangement est contrôlé par le second opérande. Dans le cas présent, le second opérande est scalaire; on ne range donc une valeur que dans un élément de la cible.

Lignes 9 à 13 et 21 à 23. Ces instructions correspondent à la gestion de la boucle **for** du programme EVA.

Ligne 14. L'instruction **sinit** produit un vecteur triplet à partir de trois valeurs scalaires.

Lignes 15 et 20. Ces deux instructions définissent un bloc vectoriel de longueur courante, la longueur du vecteur produit par l'instruction **sinit**. Ce bloc correspond au corps de la boucle **for** du programme EVA. Notons l'utilisation de l'adressage du temporaire produit par l'instruction précédente **^1** comme n'étant pas la dernière utilisation de ce temporaire. Notons aussi l'attribut **visln...** de ce même temporaire. Le **vi** indique que l'opérande est un vecteur d'entiers; le **s** signifie que ce vecteur d'entiers est un triplet; le **1** indique que le pas de ce triplet est 1; et le **n** indique que le vecteur ne possède pas de zone de description (c'est en fait le cas pour tous les vecteurs temporaires).

Lignes 16 à 19. L'instruction **gath** produit un vecteur temporaire résultat de la description de **loc@(0)** par le triplet **^2**. Le **add** incrémente ce vecteur temporaire. L'instruction **wait** garantit que les instructions précédentes sont terminées avant de déclencher l'instruction suivante. Cette instruction reflète la dépendance explicitée par l'opérateur += EVA. L'instruction **scat** peut ensuite ranger le vecteur résultat sous le contrôle du triplet dont c'est la dernière utilisation (adressage **&**).

Lignes 24 à 28. L'instruction **move** produit un vecteur temporaire contenant les valeurs du vecteur **loc@(0)**. La longueur de ce vecteur temporaire est celle de **loc@(0)** (instruction **pushvlg**). Ce

vecteur temporaire est le résultat de la fonction (instruction **return**). L'instruction **free** libère les zones (d'allocation et de description principalement) du vecteur local **loc@(0)**. (Cependant, une zone d'allocation qui serait partagée par un autre vecteur ne serait pas libérée.)

Fonction *Main*

Lignes 34 et 35. L'instruction **parout** place son premier opérande à l'offset indiqué par le second dans le segment *param_to_send* des paramètres à envoyer. L'instruction **call** appelle la fonction *geti*.

Lignes 36 à 38. L'appel de la fonction *Newton* retourne un vecteur temporaire; ce vecteur temporaire est le résultat de l'instruction **call**. Ce vecteur temporaire est associé au vecteur local **loc@(0)** (instruction **alloc**). On remarque que cette instruction référence un vecteur temporaire produit en dehors d'un bloc **pushvlg/popvlg**. Ce vecteur est en fait produit par l'instruction **return** de la fonction *Newton* (ligne 28) qui référence un temporaire produit dans un bloc **pushvlg/popvlg**.

2–14 Les objets manipulés par DEVIL et leur représentation

2–14.1 La représentation des vecteurs en DEVIL

Les vecteurs EVA sont manipulés en tant que tels en DEVIL. Un vecteur DEVIL est manipulé via son *en-tête*. Cet en-tête est composé de six champs : za, zi, type, zd, lgth et tzd. Nous décrivons ces six champs et présentons les informations contenues dans la zone d'informations.

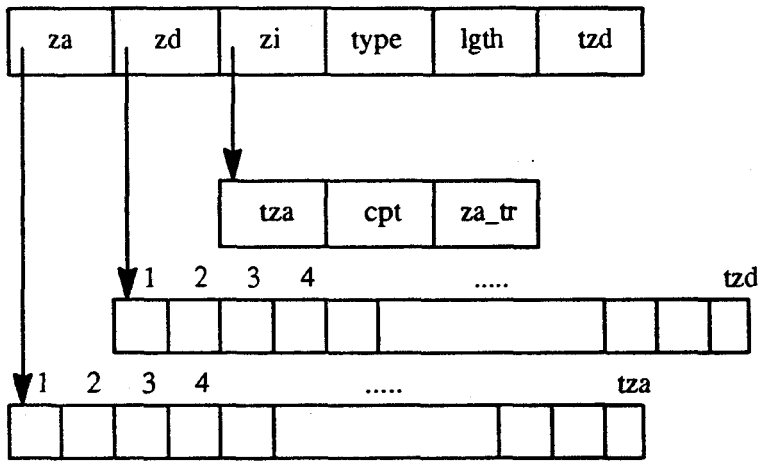
- **za**—zone d'allocation. Ce champ contient une référence sur la zone d'allocation. Cette zone d'allocation est une table de valeurs booléennes, entières ou réelles (simple ou double précision) selon le type des éléments du vecteur. Cette forme est nommée forme étendue du vecteur. Les vecteurs d'entiers peuvent aussi être codés sous forme de triplet. La zone d'allocation est alors constituée d'une table de trois valeurs entières : la borne inférieure, la borne supérieure et le pas.

- **zi**—zone d'informations. Ce champ est une référence sur une zone d'informations. Une zone d'informations est associée à toute zone d'allocation. Les vecteurs se partageant une même zone d'allocation se partagent la zone d'informations associée. Une zone d'informations est composée de :

    - **tza**—taille de la zone d'allocation. Ce champ contient le nombre d'éléments de la zone d'allocation;

    - **cpt**—compteur de références. Ce champ contient le nombre de vecteurs (en-têtes de vecteur) qui référencent la zone d'allocation associée à la zone d'informations;

    - **za_tr**—zone d'allocation triplet. Ce champ est un indicateur binaire indiquant, pour un vecteur d'entiers uniquement, si la zone d'allocation est étendue ou codée sous forme d'un triplet.



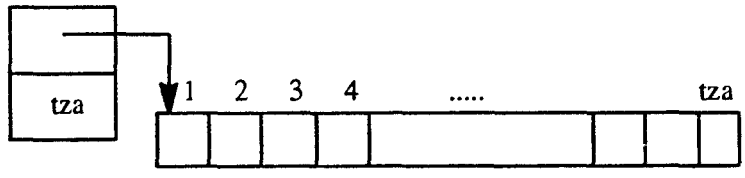
en-tête :

za	référence sur la zone d'allocation
zi	référence sur la zone d'informations
zd	référence sur la zone de description
type	type de la zone de description
lgth	longueur du vecteur
tzd	taille de la zone de description

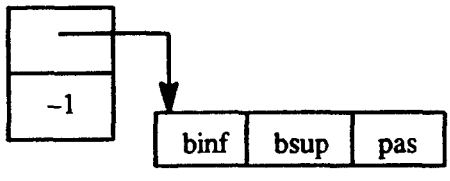
zone d'informations :

tza	taille de la zone d'allocation
cpt	compteur de références sur la zone d'allocation
za_tr	indicateur zone d'allocation étendue ou triplet

Figure 2–11. Représentation d'un vecteur EVA/DEVIL.



a—Vecteur temporaire sous forme étendue.



b—Vecteur temporaire sous forme de triplet.

tza	taille de la zone d'allocation
binf	borne inférieure
bsup	borne supérieure
pas	pas

Figure 2–12. Représentations d'un vecteur temporaire DEVIL.

- **type**—type de la zone de description. Ce champ indique le type de la zone de description du vecteur parmi : vide, liste d'index, liste de bits, triplet.
- **zd**—zone de description. Ce champ est une référence sur la zone de description du vecteur. Selon la valeur du champ type, zd contient
    - un pointeur nul si la valeur du champ type est vide;
    - une référence à une table d'entiers si le champ type est liste d'index;
    - une référence à une table de bits si le champ type est liste de bits; et
    - une référence à un triplet d'entiers si le champ type est triplet.
- **tzd**—taille de la zone de description. Ce champ contient le nombre d'éléments de la zone de description. Ce nombre d'éléments est
    - 0 s'il n'y a pas de zone de description;
    - le nombre d'éléments de la liste si la zone de description est une liste d'index ou de bits; et
    - la valeur du champ lgth si la la zone de description est un triplet.
- **lgth**—longueur. Ce champ contient le nombre effectif d'éléments du vecteur. Les éléments effectifs sont les éléments de la zone d'allocation sélectionnés par la zone de description. Selon la valeur du champ zd, ce champ vaut
    - la valeur du champ tza de la zone d'informations si le vecteur ne possède pas de zone de description;
    - la valeur du champ tzd si le descripteur est une liste d'index;
    - le nombre de bits à 1 de la zone de description si celle-ci est une liste de bits; et
    - la valeur de ((bsup – binf + pas) / pas) si la zone de description est un triplet, 0 si ce nombre est négatif. (Ceci pour les valeurs de pas strictement positives.)

Un tel en-tête de vecteur est représenté sur la figure 2–11.

2–14.2 Les vecteurs temporaires

Les vecteurs temporaires sont une notion importante en DEVIL. Les vecteurs temporaires existent en nombre virtuellement infini en DEVIL. Ces objets contiennent les résultats d'évaluation partielle des expressions EVA; des temporaires sont également produits par le compilateur EVA pour coder les blocs d'instructions `with` par exemple (cf. infra la traduction du bloc `with`). La majorité des instructions DEVIL peuvent produire une valeur temporaire. La valeur temporaire est ensuite utilisée une ou plusieurs fois (une fois dans la plupart des cas cependant) dans les instructions suivantes. Les temporaires DEVIL ne sont pas nommés; un temporaire est référencé par le numéro de l'instruction l'ayant produit. L'instruction EVA

```
x := a * (b * c - d /e)
```

se traduit en pseudo-DEVIL (*pseudo* car les modes d'adressages des objets non temporaires ne sont pas explicités et les attributs des opérandes sont absents)

```
      mul       b, c
      div       d, e
      sub       &1, &2
    n.mul       a, &1, x
```

La notation **&***x* référence la valeur du temporaire produit par la $x^{ième}$ instruction précédente. Le préfixe **n.** indique que l'instruction ne produit pas de temporaire. Les instructions à deux opérandes telles **mul**, **div**, **sub**, etc.. produisent un résultat dans une cible si celle-ci est précisée comme troisième argument. Une instruction peut produire un temporaire et un résultat dans une cible :

```
x := d := b + c
```

produit le code DEVIL suivant pour l'évaluation de l'expression à affecter à *x*

```
      mul       b, c, d
```

L'utilisation d'un temporaire par une référence **&** indique également que cette utilisation est la dernière du temporaire. Un adressage de temporaire par un **^** indique à l'inverse que le temporaire sera encore utilisé dans la suite du code. Cet adressage est rarement utilisé par le compilateur EVA; un exemple toutefois :

```
vect real v
int a, b

v [a*b] += 13.4
```

produit le code DEVIL

```
      mul       a, b
      gath      v, ^1
      add       &1, #"13.4"
    n.scat      &1, &3, v
```

l'expression *a* * *b* de description est évaluée. Son résultat est utilisé deux fois.

Les vecteurs temporaires DEVIL ne peuvent pas être cible d'une instruction. Cette gestion des vecteurs temporaires comme objets à assignation unique facilite l'allocation de ceux-ci dans des registres ou des caches en évitant les problèmes engendrés par un maintien de cohérence de mise à jour entre un cache ou un registre et une mémoire centrale. De plus elle autorise une récupération des zones allouées aux vecteurs temporaires comme zone d'allocation ou de description de vecteurs.

Représentation des vecteurs temporaires DEVIL

La forme des temporaires vecteurs DEVIL n'est pas aussi riche que celle des vecteurs EVA/DEVIL. Un vecteur temporaire n'est pas associé à un en-tête. Il ne possède pas de zone de description : un vecteur temporaire est constitué d'une suite de valeurs contiguës en mémoire. Deux types de vecteurs temporaires sont toutefois distingués : les vecteurs temporaires étendus et les vecteurs temporaires triplets. Un vecteur

temporaire étendu est représenté par la suite de ces éléments et le nombre de valeurs de cette suite. Un vecteur temporaire triplet est représenté par trois entiers et une valeur –1. Cette valeur est utilisée pour distinguer les deux formes de vecteurs temporaires, le type de vecteur temporaire pouvant parfois ne pas être connu avant l'exécution (figure 2–12).

2–14.3 Les attributs de vecteurs

A chaque opérande de toute instruction DEVIL est associé une série d'attributs. Ces attributs précisent les caractéristiques de l'opérande. Ces attributs sont générés par le compilateur. Il est nécessaire de préciser, pour chaque opérande son type et son constructeur (scalaire ou vecteur). Pour les vecteurs, des attributs supplémentaires autorisent la prise en compte de caractéristiques dès la compilation et limitent ainsi les alternatives à envisager lors de l'exécution. Ces attributs facultatifs indiquent pour la zone d'allocation du vecteur (dans le cas d'un vecteur d'entier seulement) :

- Si la zone d'allocation est étendue ou si elle est un triplet (ou si cette information est inconnue).
- Dans le cas où la zone d'allocation est un triplet, si la valeur de son pas est 1, différente de 1 (ou si cette information est inconnue).

Pour la zone de description du vecteur (dans tous les cas de vecteurs), les attributs indiquent :

- Si la zone de description est absente, ou présente (ou si cette information est inconnue).
- Dans le cas où la zone de description n'est pas vide, si elle est constituée d'une liste d'index, d'une séquence de booléens (ou si cette information est inconnue).
- Dans le cas où la zone de description est une liste d'index, si elle étendue ou si elle est un triplet (ou si cette information est inconnue).
- Dans le cas où la zone de description est un triplet, si la valeur de son pas est 1, différente de 1 (ou si cette information est inconnue).

Dans les cas favorables, c'est-à-dire lorsque tous les attributs sont spécifiés, le code produit par le traducteur est optimal. La non spécification d'un attribut oblige un test à l'exécution pour connaître la caractéristique correspondante et entraîne la génération des différents codes correspondants aux différentes alternatives. Dans le code EVA

```
vect int v
if (expr)
      v <- 1 :[2] 200
else
      v <- 100
end
... v ...
```

suite à l'alternative `if`, il est impossible de savoir si la zone d'allocation du vecteur *v* est étendue ou non. Le traducteur devra donc générer un test et le code correspondant aux deux cas.

2–15 Compilation de EVA en DEVIL

Nous présentons ici la première phase de la compilation d'un programme EVA, la production de code DEVIL par le compilateur.

Nous présentons rapidement la structure du compilateur tel qu'il fut réalisé (2–15.1). Nous nous attacherons surtout à présenter la compilation en DEVIL des structures de EVA non existantes en DEVIL. Enfin nous examinerons les informations qui sont nécessaires à DEVIL et qui ne sont pas exprimées directement en EVA (2–15.10).

2–15.1 Structure du compilateur

Précisons que la compilation de EVA en DEVIL est indépendante de la machine cible. Seul un fichier de configuration définit la taille des objets (scalaire et en-tête de vecteur) dans les segments de MAD.

Le compilateur EVA est bâti autour d'une approche de traduction dirigée par la syntaxe. Nous avons utilisé les outils `lex` et `yacc` d'Unix. La compilation est réalisée instruction par instruction. `yacc` crée une représentation arborescente qui est ensuite balayée pour produire le code DEVIL. Cette approche de transition par un arbre avait été choisie pour des raisons de souplesse de mise au point, les langages EVA et DEVIL n'étant pas encore figés lors de l'écriture du compilateur.

2–15.2 Les constructions EVA absentes en DEVIL

Nous listons ici les constructions et caractéristiques de EVA qui ne se retrouvent pas directement en DEVIL. La raison majeure de l'absence de certaines constructions en DEVIL est leur indépendance totale vis-à-vis de la machine cible. Ces constructions peuvent être consommées au niveau du compilateur EVA. DEVIL est en ce sens un langage "primitif" autorisant l'expression d'algorithmes vectoriels.

- Les objets DEVIL ne sont pas nommés mais référencés par leur adresse dans un des segments de MAD.
- Les opérandes des instructions DEVIL sont des objets et non des expressions.
- Aucune information de forme et de dimension n'est associée à un vecteur en DEVIL.
- Conséquence du précédent point, les objets instanciables ne sont pas définissables au niveau DEVIL.
- Les structures de contrôle `for`, `while`, etc... n'existent pas en DEVIL.
- La structure `with` n'existe pas en DEVIL.
- Les différents opérateurs d'affectation (= et : = mais aussi *op*=) n'existent pas en DEVIL sous la même forme qu'en EVA.

- Les opérateurs courts-circuits **andthen** et **orelse** n'existent pas en DEVIL.
- Le déclenchement des instructions DEVIL n'est pas séquentialisé mais réalisé par bloc.

Nous détaillons la génération des constructions les plus significatives dans les sections suivantes.

2–15.3 Allocation des objets EVA dans les segments de MAD

L'allocation des objets EVA est "statique"; il n'existe pas de création dynamique d'objets EVA. Il n'existe pas d'instructions d'allocation dynamique d'objets en DEVIL. Le terme objet référence ici un objet scalaire ou un en-tête de vecteur. Cependant DEVIL autorise, via les instructions de construction et d'allocation de vecteurs, la création dynamique de zones d'allocation et de description.

2–15.3.1 Allocation des objets

En DEVIL, les objets sont référencés par leur adresse dans un segment de MAD. L'allocation des objets EVA dans ces segments est donc à la charge du compilateur EVA. Cette allocation se fait à des adresses successives lors de chaque déclaration de variable dans le source EVA. Certaines variables sont automatiquement générées par le compilateur pour son usage propre, par exemple pour la gestion des boucles ou des objets instanciables (cf. infra).

La correspondance entre les différentes classes d'allocation des variables EVA et les segments de MAD est immédiate. A chaque variable externe est associée un segment *common*. Les variables globales à un module sont allouées dans le segment *data*. Les variables automatiques sont allouées dans le segment *local* et désallouées dès la fin du bloc dans lequel elles sont déclarées. Le segment *param_rec* contient les paramètres déclarés de la fonction dans l'ordre de la liste de ceux-ci dans la définition de la fonction. Le segment *param_to_send* n'est utilisé qu'au travers des instructions DEVIL de passage de paramètre; l'allocation n'est pas identifiée par EVA.

2–15.3.2 Allocation des zones de vecteurs

L'allocation des zones de vecteur est dans le cas général réalisée par les instructions DEVIL **assoc**, **alloc**, et **dassoc**. Les zones d'allocation, de description et d'information des vecteurs sont alors allouées (dynamiquement) dans le segment *heap*. Cette allocation dynamique est la seule prise en compte automatiquement par DEVIL. Les allocations de zones de vecteur dans la pile (segment *local*), dans la zone statique (segment *data*), ou dans un segment common doivent être réalisées par le compilateur EVA.

Instructions DEVIL de construction et allocation de vecteurs

Les instructions **alloc**, **assoc**, et **dassoc** réalisent les allocations de zones d'allocation, de description et d'informations des vecteurs. Ces zones sont allouées (dynamiquement) dans le segment *heap*. Ces trois instructions servent de traduction à l'opérateur d'association <- de EVA.

L'instruction **sinit** produit un vecteur triplet à partir de trois valeurs scalaires. L'instruction **sinitvlg** produit un vecteur triplet de VLG éléments à partir de deux scalaires, la borne inférieure et le pas; la valeur de la borne supérieure est déduite du nombre d'éléments à produire.

L'instruction **setza** crée un vecteur à partir de l'adresse des deux zones d'allocation et d'informations description. Aucune allocation n'est donc réalisée pour ces zones. Elles doivent avoir été préalablement allouées. Cette instruction autorise l'allocation explicite des zones d'allocation dans les segments *local* ou *common* par exemple. Les vecteurs ainsi alloués ne possèdent pas de zone de description.

Allocation dynamique par DEVIL

Nous présentons ici le cas général de génération de l'instruction d'association EVA. Trois instructions DEVIL sont utilisées suivant le type d'association (cf. la section 2–3.2, page 89 pour une description des différents types d'association). Comme cela est prévu en EVA, les trois instructions DEVIL libèrent les anciennes zones de vecteurs préalablement à toute nouvelle association.

L'instruction DEVIL **alloc** réalise l'allocation dynamique de zones d'allocation (et des zones d'informations associées) dans le segment *heap*; sa fonctionnalité est celle de l'association dynamique scalaire EVA. L'instruction EVA

```
v <- 1000
```

est générée

```
n.alloc     #"1000", v
```

L'instruction DEVIL **assoc** réalise le partage de zone d'allocation d'un vecteur source par un vecteur cible. La zone de description du vecteur cible est obtenue depuis une copie (allouée dynamiquement) de la zone de description du vecteur source. Cette instruction est donc générée pour coder la première forme d'association de partage. L'association de *v* :

```
vect real w <-...
vect v <- w
```

produit le code DEVIL

```
n.assoc     w, v
```

L'instruction **assoc** code aussi les opérations d'associations dynamiques initialisées. Le vecteur source est alors un vecteur temporaire; la zone d'allocation du vecteur cible n'est pas allouée dynamiquement : on récupère la zone d'allocation du temporaire (si le temporaire est consommé par l'instruction **dassoc** : adressage **&** et non **^**).

```
vect v <- w + 1000
```

est générée

```
  add       w, #"1000"
n.assoc     &1, v
```

La seconde forme d'association de partage (opérande droite vecteur décrit) est réalisée via l'instruction DEVIL **dassoc**. Comme pour **assoc**, la zone d'allocation du vecteur cible est allouée dynamiquement; sa zone de description est obtenue par composition (description) de la zone de description du vecteur source et du descripteur.

```
vect v <- w [d + 10]
```

est générée

```
      add       d, #"10"
n.dassoc    w, &l, v
```

Allocation statique par EVA

Les allocations de zones de vecteurs réalisées par DEVIL sont toutes faites dynamiquement dans le segment *heap*. Les optimisations mises en place pour allouer des zones de vecteurs statistiquement dans d'autres segments sont réalisés par EVA. Plusieurs conditions sont nécessaires à la réalisation d'une telle allocation :

- La taille de la zone d'allocation à allouer doit être connue dès la compilation, et
- On doit garantir que le vecteur ainsi alloué ne sera plus associé. En effet une ré-association entraînerait une désallocation automatique des zones du vecteurs qui sont supposées être allouées dans le segment *heap*, ce qui n'est pas le cas. Seuls les vecteurs déclarés constants ou à association unique peuvent donc bénéficier de cette allocation statique.

Cette allocation statique EVA est traduite via l'instruction DEVIL `setza`. Les différents cas de figure (allocation en pile, en zone statique ou en common) sont gérés sur le même modèle; nous détaillons ici l'allocation d'un vecteur sur le segment *local*.

Supposons le segment *local* de taille *t* (l'adresse disponible suivante est donc `loc@(t)`); suite à la déclaration du vecteur local

```
unassoc vect real v <- 1000
```

l'allocation est réalisée par EVA tel que représenté à la figure 2–13. La taille du segment est alors *t* + *1009*. Nous supposons qu'un en-tête de vecteur est codé sur 6 mots (adresse de base `loc@(t)`), une zone d'informations sur 3 mots (adresse de base `loc@(t+6)`), et un élément de tableau réel sur 1 mot (adresse de base de la zone d'allocation `loc@(t+9)`).

La liaison entre l'en-tête du vecteur et les zones d'informations et d'allocation est réalisée par l'instruction DEVIL

```
n.setza    loc@(t+9), #1000, loc@(t+6), loc@(t)
```

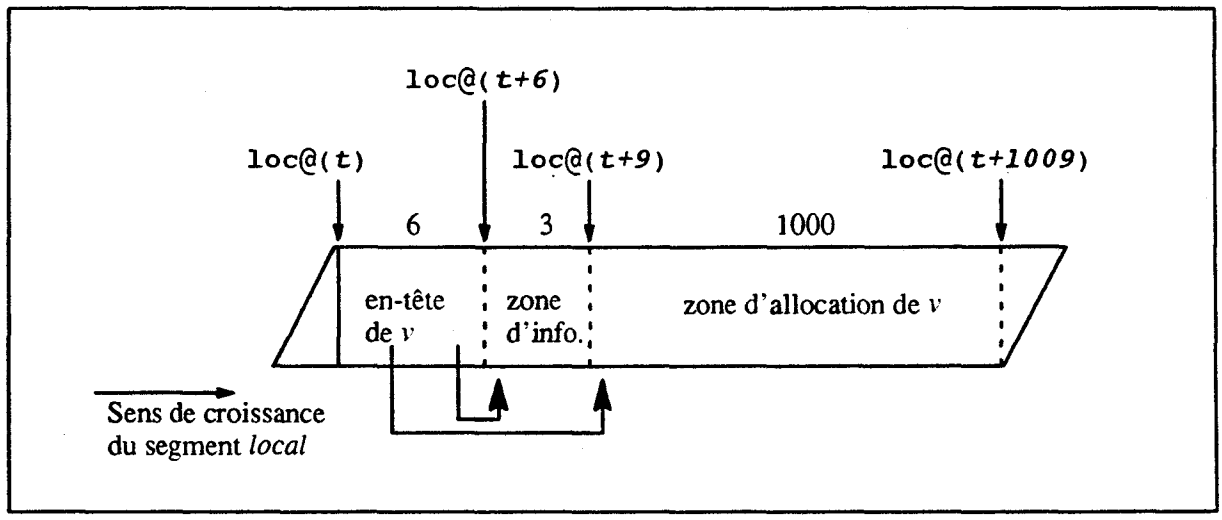


Figure 2–13. Allocation dans le segment *local* suite à l'instruction **setza**.

2–15.4 Délimitation de blocs vectoriels DEVIL

Le déclenchement des instructions DEVIL est réalisé par blocs. DEVIL définit les blocs *vectoriels* et des blocs *parallèles*. Un bloc vectoriel définit un ensemble d'instructions vectorielles traitant des vecteurs de même taille (instructions **pushvlg** et **popvlg**). Inversement, les blocs parallèles ne sont pas limités à une construction syntaxique. Nous présentons ici les blocs vectoriels; la définition des blocs parallèles et leur génération depuis EVA font l'objet de la section suivante.

Instructions DEVIL de gestion de la pile des VLG

Le segment *vlg* de MAD est géré sous forme d'une pile. Le sommet de la pile correspond à la longueur courante de traitement des vecteurs, VLG. Trois instructions sont relatives à cette pile. L'instruction **pushvlg** empile une nouvelle valeur de VLG. Cette valeur est celle de son opérande s'il est scalaire, ou la longueur de son opérande s'il est vecteur. L'instruction **popvlg** détruit le sommet de la pile des VLG. L'instruction **getvlg** retourne la valeur de VLG.

Production de blocs vectoriel depuis EVA

Un bloc vectoriel DEVIL est formé d'une suite d'instructions s'exécutant sur une même longueur. Il est limité par les instructions DEVIL de manipulation du registre VLG, **pushvlg** et **popvlg**.

EVA introduit la notion de segment pour spécifier les longueurs des opérations vectorielles (se référer à la section 2–7, page 98). Les suites d'opérations EVA réalisées sur une même longueur sont les suivantes :

- Les opérations d'un segment à longueur par défaut,
- Les opérations d'un segment à longueur explicite, et
- Chaque opération vectorielle d'un segment à longueur minimum.

EVA encadre donc chacune des suites d'instructions ainsi définies par des instructions **pushvlg** et **popvlg**. De plus le code nécessaire au calcul de la valeur de la longueur est produit. Exemples :

```
vect real u, v, w, x, y
int a, b
u := (v + w) * x                    n.pushvlg  u
                                      add      v, w
                                    n.mul      &1, x, u
                                    n.popvlg

% a + b % u := (v + w) * x            add      a, b
                                    n.pushvlg  &1
                                      add      v, w
                                    n.mul      &1, x, u
                                    n.popvlg

% < % u = (v + w) * x                nbelts   v
                                      nbelts   w
                                      min      &1, &2
                                    n.pushvlg  &1
                                      add      v, w
                                    n.popvlg
                                      nbelts   ^2
                                      nbelts   x
                                      min      &1, &2
                                    n.pushvlg  &1
                                      mul      &6, x
                                    n.popvlg
                                      nbelts   ^2
                                      nbelts   u
                                      min      &1, &2
                                    n.pushvlg  &1
                                    n.move     &6, u
                                    n.popvlg
```

On remarquera en particulier la pénalisation introduite par les segments à longueur minimum. Celle-ci est d'autant plus importante si l'on considère que de nombreuses optimisations sur la génération du code ne sont possibles qu'au sein d'un bloc vectoriel DEVIL. Nous discutons de ces optimisations ci-dessous dans le cadre des blocs parallèles.

2–15.5 Délimitation des blocs parallèles DEVIL

Les blocs vectoriels DEVIL sont partitionnés en blocs parallèles. Au sein d'un bloc vectoriel, un bloc parallèle est essentiellement limité par les instructions DEVIL de synchronisation `wait` (les instructions de rupture de séquence, appel et retour de fonction, saut et les instructions d'allocation de vecteurs coupent aussi les blocs parallèles).

Par définition, les instructions d'un bloc parallèle peuvent être déclenchées en même temps. Seules les dépendances de flots entre les instructions (utilisation d'un temporaire produit par une instruction précédente du bloc parallèle) peuvent amener un séquencement dans le déclenchement d'instructions.

Pour respecter la définition des blocs parallèles DEVIL, les dépendances existant au sein d'un segment de longueur EVA (donc un bloc vectoriel DEVIL) produisent donc une instruction de synchronisation `wait` :

- Une dépendance est supposée entre deux instructions EVA à moins qu'elles ne soient séparées par l'indicateur point-point `..` (cf. la section 2–8.4).

```
vect u,v w, x, y, z
% longueur % [
      u := v + w
      x := y * z
]
```

produit deux blocs parallèles :

```
n.pushvlg   longueur
n.add       v, w, u
n.wait
n.mul       y, z, x
n.popvlg
```

- De plus une affectation = ou *op*= laisse supposée une dépendance entre la partie droite et gauche de l'affectation, EVA génère donc une instruction DEVIL **`wait`** entre l'évaluation de la partie droite et le rangement dans la partie gauche (section 2–15.6 ci-dessus) :

```
vect u,v w
u = v + w
```

produit aussi deux blocs parallèles :

```
n.pushvlg   u
  add       v, w
n.wait
n.move      &2, u
n.popvlg
```

Les blocs parallèles non réduits à une opération sont donc obtenus par une construction de blocs à longueur spécifique EVA dont les instructions sont non-dépendantes les unes des autres; par exemple

```
vect real t, u, v, w, x, y, z
vect real i, j
int lgth
% lgth % [
      t := u + v ..
      w [i] := x ..
      y := z [j]
]
```

sera généré

```
n.pushvlg   lgth
n.add       u, v, t
n.scat      x, i, w
n.gath      z, j, y
n.popvlg
```

Les trois instructions EVA du même bloc de longueur se compilent en trois instructions DEVIL d'un même bloc parallèle.

2–15.6 Génération des affectations EVA

EVA dispose de deux affectations, le = et le :=. DEVIL ne propose le transfert d'informations qu'au travers de l'instruction **move**, de l'instruction de rangement contrôlé **scat** et des instructions acceptant un opérande cible. Nous nous intéressons ici uniquement à l'affectation de vecteur.

Instructions de rassemblement et éclatement en DEVIL

Les instructions DEVIL **gath** et **scat** réalisent les opérations de rassemblement et d'éclatement des éléments d'un vecteur. Ces rassemblements ou éclatements sont contrôlés par un descripteur qui est un vecteur d'entiers, un vecteur de booléens, un vecteur d'entier triplet, ou encore un scalaire entier. Dans ce dernier cas, les instructions réalisent un accès à un élément du vecteur.

Notons bien que les instructions DEVIL acceptent des opérandes vecteurs de type EVA/DEVIL incluant une zone de description. Une opération de gather ou de scatter, qualifiée d'implicite, peut alors être réalisée en dehors des seules instructions **gath** et **scat**.

2–15.6.1 Affectation :=

L'affectation := est générée en une instruction **move** DEVIL si la partie droite de l'affectation ne résulte pas d'une instruction pouvant produire son résultat dans une cible. Les codes EVA

```
vect real a, b                  et          vect real a, b, c
a := b                                      a:= b + c
```

sont générés (en pseudo-DEVIL)

```
n.pushvlg   a                                   n.pushvlg   a
n.move      b, a            et                  n.add       b, c, a
n.popvlg                                        n.popvlg
```

Une instruction EVA affectant un vecteur décrit est générée à l'aide d'un rangement contrôlé; par exemple

```
vect real a, b                                  vect real a, b, c
vect int i                      et              vect int i
a [i] := b                                      a [i] := b + c
```

produisent les codes

```
n.pushvlg   i                                   n.pushvlg   i
n.scat      b, i, a             et                add       b, c
n.popvlg                                        n.scat      &l, i, a
                                                n.popvlg
```

2–15.6.2 Affectation =

La sémantique de l'affectation = de EVA impose le passage par une zone temporaire. Cela est réalisé par la production d'un temporaire par l'évaluation de l'expression parte droite et le rangement de ce temporaire

dans la partie gauche. Les deux instructions ne pouvant être déclenchées en parallèle, elles sont séparées par une instruction DEVIL **wait**. (Se reporter à la génération des blocs parallèles DEVIL ci–dessus.) Le code EVA

```
vect real a, b, c
a = b + c
```

est généré (en pseudo-DEVIL)

```
n.pushvlg   a
  add       b, c
n.wait
n.move      &2, a
n.popvlg
```

La dépendance entre les parties droite et gauche, exprimée par le =, est prise en compte par une rupture de déclenchement des instructions entre l'évaluation de la partie gauche et son rangement dans la partie droite.

2–15.6.3 Les affectations *op*=

Les affectations ***op***= et ***op***: = de EVA sont des raccourcis pour le programmeur mais autorisent aussi, dans certains cas, l'optimisation du code généré. Certaines optimisations sont possibles bien que DEVIL ne présente pas d'instruction de ce type; le code EVA

```
vect real a, b
a +:= b
```

est généré

```
n.pushvlg   a
n.add       a, b, a
n.popvlg
```

Les optimisations sont obtenues par la réutilisation "explicite" d'une description.

```
vect real a, b
vect int i
a [i] +:= b          équivalent à          a [i] := a [i] + b
```

est généré

```
n.pushvlg   i
  move      i
  gath      a, ^1
  add       &1, b
n.scat      &1, &3, a
n.popvlg
```

Le réutilisation et la consommation d'un temporaire dans un même bloc **pushvlg/popvlg** d'instructions DEVIL sont la base d'une génération de code performant. Entre autre, elles autorisent l'allocation du temporaire uniquement en registre.

2–15.7 Génération relative aux informations de forme des vecteurs EVA

Les informations de forme et de dimension sont propres aux vecteurs EVA. On ne les retrouve pas dans les vecteurs DEVIL. Nous détaillons ici comment ces informations doivent être passées à DEVIL pour être prises en compte. Nous présentons les instructions DEVIL autorisant la projection multi-dimensionnelle des vecteurs et expliquons comment ces instructions sont générées depuis EVA.

Codage des informations de forme EVA

Les informations de forme des vecteurs EVA ne sont pas nécessairement évaluables lors de la compilation. Dans ce cas le code DEVIL pour l'évaluation de ces informations est produit lors de la déclaration de la variable. Un emplacement est alloué dans le segment *local* ou *data* (selon que le vecteur est déclaré dans une fonction ou non) pour stocker le résultat. Soit la fonction

```
f (vecteur, shift, taille)
      int shift, taille
      vect real vecteur [shift .. shift + taille]
...
end
```

Les déclarations produisent le code DEVIL suivant (la borne inférieure de *vecteur* est allouée en `loc@(0)`, la borne supérieure en `loc@(1)`):

```
n.move      shift, loc@(0)
n.add       shift, taille, loc@(1)
```

Ces informations sont ensuite utilisées pour construire les descripteurs opérant avec le vecteur *vecteur*.

Instruction DEVIL de production de descripteurs multi-dimensionnels

Les vecteurs DEVIL sont linéaires et aucune information de forme ne leur est associée. Les descripteurs opérandes des instructions **`gath`**, **`scat`**, ou **`dassoc`** doivent être mono-dimensionnels. Les instructions DEVIL **`dim`** et **`diml`** produisent des descripteurs linéaires à partir de deux descripteurs pour deux dimensions. Les descripteurs pour des espaces de dimension supérieure, disons *n*, sont construits à partir d'un descripteur linéaire produit pour *n–1* dimensions et du descripteur de la $n^{ième}$ dimension, donc par des appels successifs à **`dim`** et **`diml`**.

Les opérandes de l'instruction **`dim`** sont un vecteur descripteur de la dimension inférieure, le vecteur de description de la dernière dimension, les bornes inférieure et supérieure des indices dans la dernière dimension, et la borne inférieure des indices dans la dimension inférieure. Pour l'instruction **`diml`**, la valeur de l'opérande borne inférieure des indices de la dernière dimension est 1.

La composition de descripteurs par les instructions **`dim`** ne produit pas systématiquement un vecteur d'index. Les cas permettant de produire un triplet sont traités comme tels. Ces cas sont identifiés par un descripteur triplet et un descripteur scalaire :

```
vect real matrice [L, C]
int binf, bsup, pas, scal
... matrice [scal, binf :[pas] bsup] ...
... matrice [binf :[pas] bsup, scal] ...
```

Ces descriptions bi-dimensionnelles sont transformées en descriptions linéaires par un triplet. Nous supposons que les bornes supérieures des triplets ne référencent pas d'élément en dehors de la ligne ou la colonne (*bsup* ≤ *C* ou *bsup* ≤ *L*). Nous précisons aussi que les matrices sont rangées en mémoire par colonnes. Nous obtenons les équivalences suivantes :

```
... matrice [(binf-1)*L + scal :[pas*L] (bsup-1)*L + scal] ...
... matrice [(scal-1)*L + binf :[pas] (scal-1)*L + bsup] ...
```

Génération des descriptions multi-dimensionnelles EVA

Comme indiqué ci-dessus, les opérandes utilisés comme descripteur des vecteurs DEVIL doivent être linéaires; ceux-ci sont obtenus, si nécessaire, via les instructions DEVIL **dim** et **diml**. Soit le code EVA

```
vect real v [21 .. 40,  31 .. 50]
vect int i, j
...
v [i, j] = 12.3
```

le descripteur nécessaire à la réalisation de l'opération est construit par EVA :

```
    dim       I, J, #"21", #"40", #"31"
n.scat        #"12.3", &1, v
```

L'opération de scatter est alors réalisable, comme toutes les opérations sur les vecteurs multi-dimensionnels décrits, sur tous les éléments en parallèle; les deux dimensions sont accédées en parallèles. Les travaux de Tsuda et Kunedia autour du langage V-PASCAL [TsKu90] montrent des gains de performance obtenus par la transformation de boucles FOR PASCAL imbriquées en une boucle unique via un adressage vectoriel indirect (gather/scatter) [TsKu86], [Tsud88]. L'implantation des instructions DEVIL **dim** et **diml** est équivalente à la transformation présentée par Tsuda et Kunedia.

2–15.8 Gestion des variables instanciables au sein du compilateur EVA

Nous présentons ici comment les variables instanciables sont manipulées au sein du compilateur EVA et le code généré lors de leur utilisation.

Une variable instanciable est associée, au sein du compilateur EVA, à un arbre binaire représentant l'expression l'initialisant. Certaines feuilles de cet arbre correspondent à une primitive instanciable de EVA. Lors de l'utilisation de la variable dans une expression de description, le code est produit à partir de l'arbre et en référence au vecteur décrit; c'est-à-dire que les primitives instanciables sont remplacées par leur valeur pour le vecteur décrit. Exemple (repris de la section 2–6.2, page 97)

```
const int milieu = (upper - lower)/2
const vect int centre <- lower+1 : upper-1
vect real  v1, v10 [10]
...
v10 [milieu] = 121.0
v1 [centre] = 143.0
```

Le code DEVIL produit pour l'affectation de *v10* est

```
        sub       #"10", #1
        div       &1, #"2"
      n.scat      #"121.0", &1, v10
```

(Les optimisations telles la réduction ou la propagation des valeurs immédiates et littérales ne sont pas appliquées ici; nous présentons le code produit de manière générale.)

celui pour l'affectation du vecteur *assumed-length v1* est

```
-- lower + 1
        add       #1, #"1"
-- upper -1
        nbelts    v1
        sub       &1, #"1"
-- vecteur de description
        sinit     &5, &2, #1
-- affectation
      n.pushvlg   ^2
      n.scat      #"143.0", &3, v1
      n.popvlg
```

2–15.9 Compilation de la structure `with`

La structure **`with`** est une construction propre à EVA. Sa compilation en DEVIL est basée sur le principe de la ré-utilisation d'un temporaire résultat de l'évaluation de l'expression de description du bloc **`with`** et sur le fait que le bloc **`with`** forme un bloc à longueur spécifiée (voir l'équivalence d'un bloc **`with`** à la section 2–9.2.1). De plus les instructions décrites DEVIL sont utilisées. L'exemple

```
vect real a, b, c
vect int expr, i, j
with [expr]
      a := b + c ..
      d [i] := e
      f := g [j]
end
```

sera généré

```
      n.pushvlg   expr
        move      expr
      n.add       b, c, a [^1]
      n.scat      e, i, d [^2]
      n.wait
      n.gath      g, j, f [^4]
      n.reltmp    &5
      n.popvlg
```

L'instruction **`reltmp`** a pour seul rôle de libérer le temporaire résultat de l'expression de description du bloc **`with`**. Une autre génération pourrait se passer de cette instruction et référencer le temporaire par un **`&`** lors de sa dernière utilisation.

Une structure `with` autorise donc la génération de blocs parallèles importants, base de génération de code performant à partir de DEVIL.

2–15.10 Génération des attributs de vecteurs

Les informations d'attributs des vecteurs (section 2–14.3) doivent être passées à DEVIL alors qu'elles ne sont pas directement présentes dans le code EVA. Le compilateur EVA est donc chargé de les découvrir.

EVA gère, dans la table des symboles, une structure de données codant les attributs de chaque vecteur. Ces attributs sont passés à DEVIL à chaque utilisation du vecteur comme opérande. Ils sont mis à jour à chaque modification possible du vecteur par une instruction. On distingue donc :

1 L'initialisation des attributs.

- La déclaration d'un vecteur initialise à "vide" ses attributs.
- La déclaration d'un vecteur paramètre formel initialise à "inconnue" les valeurs de ses attributs. Ces valeurs devraient être, pour chaque appel, celles des attributs des paramètres effectifs. Mais il n'existe pas, en EVA, de mécanisme autorisant la précision d'une telle information.

2 La mise à jour des attributs.

- L'association d'un vecteur ré-initialise ses attributs en fonction des caractéristiques de la partie droite.
- L'affectation d'un vecteur modifie ses attributs en fonction des anciennes valeurs et des attributs de la partie droite. Cette modification recouvre l'extension des zones d'allocation représentée sous forme d'un triplet (section 2–2.4).
- Le passage en paramètre d'un vecteur positionne à "inconnue" la valeur de tous ses attributs au retour de la fonction.

  Cependant le passage de vecteurs déclarés constant ou à association unique limite cette ré-initialisation des attributs. Un vecteur à association unique ne pouvant être ré-associée, la plupart de ses attributs ne peuvent être modifiés. Seuls les attributs concernant l'extension ou non de la zone d'allocation peuvent être modifiés : la zone d'allocation triplet d'un vecteur est étendue par l'affectation d'une expression non triplet.

```
unassoc vect int a <- 1 : 1000          -- za triplet
a = 1|2|3|4                             -- za étendue
```

  Aucun des attributs d'un vecteur constant ne peut être modifié suite à sa déclaration. C'est à ce niveau que se situe un des avantages majeurs des vecteurs constants et à association unique.

Cette gestion des attributs est faite en fonction du flot d'exécution du programme. Par exemple, les attributs d'un vecteur à la sortie d'une alternative sont obtenus par composition des attributs avant l'alternative et de ceux produits au sein de l'alternative.

–Nous remarquons l'effet négatif des passages de paramètres vecteurs sur la spécification des attributs :

1 Du côté de la fonction appelée aucun attribut ne peut être précisé pour un vecteur paramètre, et

2 Du coté de la fonction appelante, au retour de l'exécution de la fonction appelée, les attributs des vecteurs paramètres sont indéterminés.

La résolution de ce problème est envisageable par 1—La mise en ligne des fonctions locales au module comportant des paramètres vecteurs, ou 2—L'ajout d'une spécification par le programmeur des caractéristiques des paramètres d'une fonction, ou 3—(généralisant le point précédent) L'intégration des informations contenues dans les attributs dans le type des vecteurs.

2–16 Implantation de DEVIL sur Cray Y-MP

Nous présentons ici l'implantation de DEVIL réalisée sur Cray Y-MP. Cette implantation fut réalisée par Philippe Preux au SCRI de Florida State University. Nous comparerons les performances obtenues avec celles d'autres langages.

Cette implantation génère du code CAL (Cray Assembly Language) pour un Cray Y-MP mono-processeur.

Deux parties sont distinguées dans le traducteur de DEVIL : le "squelette" qui regroupe les routines indépendantes de la machine cible, et les fonctions dépendantes de la machine cible qui recouvrent essentiellement la génération de code proprement dite.

Pour illustrer les codes CAL, nous utilisons un pseudo-assembleur qui est une forme "lisible" du CAL. Les fonctionnalités sont celles du CAL; cependant nous nous autorisons certains raccourcis, par exemple dans les adressages. Se reporter à l'annexe A pour une description de ce pseudo-assembleur.

Nous présentons la traduction des blocs vectoriels et parallèles, l'allocation des segments de MAD; nous insistons sur l'allocation des vecteurs temporaires DEVIL; nous donnons ensuite deux représentations des vecteurs de booléens, enfin nous présentons et discutons quelques performances obtenues.

2–16.1 Codage des blocs vectoriels et parallèles

La génération de code est réalisée bloc parallèle par bloc parallèle. Nous présentons ici la structure de la génération d'un tel bloc.

Un bloc vectoriel est formé d'instructions agissant toutes sur des vecteurs de même taille. Un tel bloc est partitionné en bloc(s) parallèle(s). Un bloc parallèle est composé d'instructions non dépendantes. Les deux conditions nécessaires à la fusion de boucles (même longueur et non dépendance) sont donc présentes pour toutes les instructions d'un bloc parallèle.

Toutes les instructions d'un bloc parallèle sont donc générées comme une boucle unique (qualifiée de boucle de strip-mining). Chaque itération de la boucle traite des vecteurs de taille égale à la taille des

registres vectoriels (VRS). La première boucle traite éventuellement moins d'éléments (valeur du registre VLG non multiple de VRS) par positionnement du registre contrôlant la longueur des opérandes vectoriels; le code EVA/DEVIL

```
% long % {                          n.pushvlg  long
     a := b + c ..                  n.add      b, c, a
     d := e * f                     n.mul      e, f, d
}                                   n.popvlg
```

est généré comme suit (en pseudo-assembleur); le registre *A1* contient l'index du premier élément à traiter dans une itération, les registres *S1* et *S2* contiennent respectivement le résultat et le reste de la division entière de *long* par VRS (*long* = *S1* * VRS + *S2*).

```
-- Une boucle unique pour le bloc parallèle
     MOVE       A1          0
     -- taille des vecteurs de la première itération
     MOVE       VL          S2
LOOP:
     -- n.add b, c, a
     VLOAD      V1          b [A1]
     VLOAD      V2          c [A1]
     VADD       V1          V2
     VSTORE     a [A1]      V1
     -- n.mul e, f, d
     VLOAD      V3          e [A1]
     VLOAD      V4          f [A1]
     VMUL       V3          V4
     VSTORE     d [A1]      V3
     -- taille des vecteurs pour les itérations suivantes
     ADD        A1          VL
     MOVE       VL          VRSnb
     -- gestion de la boucle
     DECR       S1
     JNZ        S1          LOOP
```

La gestion de la boucle est éventuellement supprimée si la taille du bloc parallèle (VLG) est connue à la compilation comme inférieure ou égale à VRS.

Les techniques habituelles de déroulage de boucles sont mises en place à ce niveau; deux conditions commandent cette mise en place. Le nombre de registres libres doit être suffisant, et le corps de la boucle doit être de taille raisonnable pour que le déroulage n'oblige pas des chargements inutiles du *buffer* d'instructions (très pénalisant, toute autre activité étant alors gelée).

Entre deux boucles correspondant à deux blocs parallèles, est insérée une instruction CAL CMR qui assure que tous les transferts mémoire déclenchés sont terminés avant de reprendre l'exécution du flot. Nous prenons ainsi en compte les éventuelles dépendances entre les deux blocs.

2–16.2 Allocation des segments de MAD

Nous présentons maintenant l'allocation des objets DEVIL. Nous distinguons l'allocation des vecteurs temporaires (section suivante) et celle des autres objets.

Les objets scalaires et les en-têtes de vecteurs sont alloués par EVA dans les segments *local* et *data* de MAD. Ces segments sont projetés directement sur la pile et la zone de donnée globale respectivement. Ces segments contiennent aussi les zones de vecteurs allouées directement par EVA car leur taille est connue à la compilation et parce qu'elles n'auront pas à être désallouées automatiquement par une instruction DEVIL (section 2-15.3.2).

Les zones de vecteurs sont, pour la plupart, allouées par les instructions DEVIL dans le segment *heap* de MAD; les allocations dans ce segment sont réalisées dynamiquement sur un tas.

2–16.3 Allocation des vecteurs temporaires

Nous discutons ici de l'allocation de vecteurs temporaires DEVIL et montrons comment celle-ci est facilitée par les constructions offertes par EVA et DEVIL.

Rappelons qu'un vecteur temporaire produit par une instruction DEVIL est constitué d'une suite de valeurs contiguës "en mémoire" (si l'on exclut les vecteurs temporaires sous forme de triplets). Rappelons aussi que le nombre d'éléments d'un vecteur temporaire est égal à la valeur du registre VLG lors de la création du vecteur.

Ces vecteurs temporaires doivent être alloués au mieux, c'est-à-dire si possible dans les registres vectoriels. Il est simple de déterminer si une telle allocation est possible en EVA/DEVIL.

- Si la longueur du vecteur est connue à la compilation comme inférieure ou égale à VRS (Vector Register Size, taille des registres vectoriels), le vecteur temporaire est alloué dans un registre vectoriel qui contient alors tous ses éléments; aucune allocation en mémoire n'est réalisée pour ce vecteur temporaire.

  La détection de la condition nécessaire à cette allocation en registre est directement liée à la valeur du registre VLG lors de la production du temporaire.

  Cette allocation peut être étendue à des vecteurs temporaires dont la taille est connue à la compilation comme inférieure à un "petit" multiple de VRS. Par exemple un vecteur de taille 128 est alloué dans deux registres de 64 éléments. Les opérations utilisant ce vecteur sont alors "dédoublées" pour les deux registres.

- Un autre cas plus général autorisant l'allocation en registre d'un vecteur temporaire concerne les vecteurs temporaires produits au sein d'un bloc parallèle DEVIL et consommés dans ce même bloc parallèle. Un tel vecteur temporaire peut être alloué dans un registre vectoriel, indépendamment de sa longueur et de celle des registres vectoriels de la machine; de même que dans le cas précédent, aucune allocation en mémoire n'est nécessaire pour ce vecteur temporaire.

  En effet, un bloc parallèle est une construction au sein de laquelle il n'existe pas de dépendance entre les différentes itérations de la boucle d'instructions produite pour ce bloc (cf. ci-dessus). Les valeurs produites pour un vecteur temporaire lors d'une itération sont donc consommées lors de cette même itération. Le vecteur temporaire est donc alloué par tranches correspondant à une itération dans un registre vectoriel unique. De plus, la durée de vie de ce registre s'étend de la création du temporaire à la dernière utilisation de ce temporaire; le registre naît et meurt à chaque itération.

La reconnaissance de tels cas de figure est immédiate en DEVIL (les blocs parallèles sont syntaxiquement identifiables, la création et la dernière utilisation d'un vecteur temporaire également); la production des informations utilisées par DEVIL depuis un programme EVA est elle aussi liée à des constructions syntaxiques (cf. la section 2–15.5).

Les vecteurs temporaires n'entrant dans aucun des deux groupes définis ci-dessus seront alloués en mémoire. Il s'agit de vecteurs temporaires produits dans un bloc parallèle et utilisés dans un autre bloc parallèle. Cette allocation est réalisée dynamiquement, avant le bloc parallèle dans lequel est produit le vecteur temporaire.

Le vecteur temporaire est alors manipulé selon le schéma suivant :

- L'instruction produisant le vecteur temporaire le produit par tranche, lors des itérations successives de la boucle d'instructions codant le bloc parallèle. Une tranche est produite dans un registre. Ce registre est utilisé dans le bloc parallèle pour accéder la valeur du temporaire.

- A la fin de chaque itération de ce bloc parallèle, la tranche du vecteur temporaire produite durant l'exécution du bloc est rangée (à la suite de celles produites par les itérations précédentes) dans la zone allouée pour le vecteur temporaire. Ce rangement peut d'ailleurs être effectué dès la rencontre de la dernière utilisation du vecteur temporaire dans le bloc parallèle; le registre utilisé par le temporaire est alors libéré.

- Dans les blocs parallèles suivants utilisant le vecteur temporaire, celui-ci est chargé par tranche dans un registre. Ce registre est utilisé au sein du bloc. Les vecteurs temporaires étant, par définition, à affectation unique en DEVIL, le registre n'a pas à être sauvegardé en mémoire à la fin du bloc. De plus, s'il est nécessaire de vider (*spiller*) des registres en mémoire, la valeur d'un tel registre n'a pas non plus besoin d'être réécrite en mémoire.

- Suite au bloc référençant la dernière utilisation du temporaire, la zone allouée en mémoire pour ce temporaire est libérée.

Nous donnons ici un exemple (se reporter à la section 2–15.5 pour la compilation de EVA en DEVIL).

```
% lg % {                                n.pushvlg   lg
      a = b + c   ..                      add       b, c
                                         n.wait
                                         n.move      &2, a
      d := e * f                         n.mul       e, f, d
}                                        n.popvlg
```

Le code DEVIL contient deux blocs parallèles, le vecteur temporaire résultant de l'évaluation de l'expression $b+c$ est utilisé dans un bloc parallèle qui n'est pas celui où il est créé. Le premier bloc parallèle est généré comme suit.

```
-- Première boucle, générant le temporaire
        -- Initialisations :
                A2 est l'index du premier élément à traiter dans une itération
                VL contient le nombre d'éléments à traiter lors de la première itération
                S1 contient le nombre d'itérations à effectuer
        -- Allocation de la zone pour le temporaire résultant de expr, son adresse est produite
dans le registre A1; on en garde une copie dans le registre A3
        CALL        A1          MALLOC      VLG
        MOVE        A3          A1
LOOP1:
        -- Production du temporaire : add b, c
        -- on produit le strip dans le registre V1; on le range en mémoire
        VLOAD       V1          b [A2]
        VLOAD       V2          c [A2]
        VADD        V1          V2
        VSTORE      A1          V1
        --
        ADD         A1          VL
        ADD         A2          VL
        MOVE        VL          VRS
        DECR        S1
        JNZ         S1          LOOP1
        CMR
```

Le second bloc parallèle utilise un temporaire produit précédemment, et contient la dernière utilisation de ce temporaire; il est généré comme suit

```
--Seconde boucle, utilisant le temporaire
        --Ré-initialisations de A2, VL, et S1
        --Initilialisation de A1, adresse de la zone temporaire, avec A3
        MOVE        A1          A3
LOOP2:
        --Chargement d'un strip du temporaire dans V1 pour n.move &2, a
        VLOAD       V1          A1
        VSTORE      a [A2]      V1
        --Réalisation de n.mul e, f, d
        VLOAD       V2          e [A2]
        VLOAD       V3          f [A2]
        VMUL        V2          V3
        VSTORE      d [A2]      V2
        --
        ADD         A1          VL
        ADD         A2          VL
        MOVE        VL          VRS
        DECR        S1
        JNZ         S1          LOOP2
        --Libération de la zone temporaire
        CALL        FREE        A3
```

2–16.4 Allocation des registres et génération de code

Nous discutons ici de l'allocation des registres et de la génération de code telles qu'elles sont réalisées au sein du traducteur DEVIL. La mise en place de ces deux mécanismes est simple et ne comporte pas de phase d'optimisation complexe.

Lors de la génération d'un bloc parallèle, une représentation intermédiaire des instructions du bloc est construite. Elle associe à chaque opérande son état à l'entrée dans le bloc. Les états possibles sont : en registre au début du bloc, en mémoire au début du bloc, ou produit par une instruction du bloc.

Un balayage de cette structure détermine ensuite les opérations à réaliser pour chaque instruction (chargement éventuel des opérandes de la mémoire vers les registres, etc...). A l'issue de cette phase, le nombre de registres nécessaires au codage du bloc est connu. C'est à ce niveau que sont mises en place les éventuelles opérations de vidage des registres en mémoire. On décide aussi d'optimisations telles le déroulage de boucles. Un dernier balayage alloue les registres et produit le code. L'allocation des registres vectoriels est réalisée par disponibilité; on utilise le registre le plus anciennement libéré dans le flot d'instructions générées. Cette heuristique ne correspond pas au cas idéal qui est d'allouer le registre le plus anciennement libéré dans le flot d'exécution, mais est généralement satisfaisante.

2–16.5 Représentations des vecteurs de booléens : *first-full* et *last-full*

Comme indiqué précédemment (section 2–2.3.5), les vecteurs de booléens sont identifiés comme tels en EVA. Cette information est conservée lors de la génération de code DEVIL. De plus, nous avons mis en place un codage particulièrement adapté à l'exécution des blocs parallèles tels qu'ils sont traduits depuis DEVIL. Nous exposons maintenant l'implantation de ces vecteurs de booléens.

Sur le Cray, nous codons chaque élément d'un vecteur de booléens sur un bit. Les mots mémoire du Cray sont de 64 bits. Un mot mémoire contient donc 64 éléments; le dernier mot contient éventuellement moins d'éléments. Considérons la boucle de strip-mining d'un bloc parallèle. Chaque itération de cette boucle traite VRS = 64 sur le Cray éléments sauf la première boucle qui en traite éventuellement moins (section 2–16.1).

Dans le cas général (longueur du vecteur de booléens non multiple de 64), les éléments d'un vecteur de booléens à traiter lors d'une itération sont sur deux mots successifs en mémoire. Nous proposons une implantation des vecteurs de booléens, qualifiée de *last-full*, qui range dans un même mot mémoire tous les éléments d'un vecteur de booléens devant être traités lors d'une itération. Le premier mot du vecteur est donc incomplètement utilisé, la première itération étant éventuellement réalisée sur une longueur de vecteur moindre.

Cette allocation last-full n'est possible que si l'on peut assurer, à la compilation, que toutes les utilisations du vecteur de bits seront réalisées sur une même longueur VLG. Cette condition est facilement identifiable pour les vecteurs temporaires. Il suffit que la création du vecteur temporaire et toutes ses utilisations soient faites au sein d'un même bloc vectoriel ou de plusieurs blocs vectoriels de même taille VLG; la découverte de cette condition n'est liée qu'à des structures syntaxiques de EVA/DEVIL.

Les vecteurs de bits qui ne peuvent être alloués last-full sont alloués de manière traditionnelle; ils sont qualifiés de *first-full*, le premier mot mémoire étant complètement utilisé. Chaque itération d'une boucle utilise alors des éléments du vecteur de bits rangés dans deux mots mémoire consécutifs.

2–16.6 Performances sur Cray Y-MP

Nous donnons ici quelques éléments de performance du code obtenu sur un processeur de Cray Y-MP/432. Cette section est articulée autour de différentes comparaisons. Une première série de codes compare les performances de EVA/DEVIL à celles obtenues par le code généré par le compilateur FORTRAN CFT. Nous fournissons ensuite l'influence de caractéristiques de EVA/DEVIL sur les performances, par exemple la structure des vecteurs ou la représentation des vecteurs de booléens.

Comparaisons avec FORTRAN

Nous comparons des programmes EVA/DEVIL avec leurs équivalents en FORTRAN CFT77. Deux versions des programmes FORTRAN sont comparées. Une version, qualifiée de implicite, est écrite en FORTRAN77 scalaire. Une seconde version utilise la notation explicite de FORTRAN CFT. Les programmes FORTRAN implicites sont vectorisés. Les programmes FORTRAN explicites et EVA/DEVIL n'ont pas à l'être. Les programmes EVA sont donnés dans les figures 2–14, 2–16 et 2–17. Il s'agit de

1—fonctions linéaires,
2—multiplication de complexes,
3—extraits du code *hydro*,
4—affectation masquée, et
5—instructions décrites (bloc `with/where`)

Les vecteurs EVA/DEVIL sont des vecteurs alloués sur la pile (déclarés à association unique et de taille connue à la compilation). Les programmes FORTRAN utilisent aussi des vecteurs locaux dont la taille est connue à la compilation.

Les codes correspondant aux programmes FORTRAN sont obtenus en activant toutes les optimisations du compilateur (version 4.0.0).

Les temps sont obtenus par la fonction `second ()` retournant le temps CPU écoulé depuis le début d'un programme. Les tests ont été réalisés sous une charge normale du Cray. Les résultats présentés sont issues de multiples exécutions de chaque code.

Nous comparons les temps d'exécution des différents programmes sur des vecteurs de taille *size* = 10 000 éléments. Les temps sont donnés ci-dessous en 10^{-6} secondes. Ils correspondent à la durée d'exécution des instructions pour FORTRAN et à la durée d'exécution des instructions et de création des vecteurs pour EVA/DEVIL.

Code 1 : fonctions linéaires

```
vect real x, y, z, t
real a, b
      y := a * x + b                -- ml
      z := (x * y + a) * b          -- mvl
      t := ((x * a) + y) * z        -- mvfl
```

Code 2 : multiplication de complexes

```
vect real  xr, xi, yr, yi, zr, zi
vect real x, y, z, t
real c, s
-- mcl
      % size % {
            zr := xr * yr - xi * yi ..
            zi := xr * yi + xi * yr
      }

-- mc2
      % size % {
            x := c * z - s * t ..
            y := s * z - c * t
      }
```

Code 3 : extrait du code *hydro*

```
vect real x, y, z
real q, r, t
vect real z10, z11

-- x [k] = q + y [k] * (r * z [k + 10] + t * z [k + 11])
      z10 <- z [10:*]
      z11 <- z [11:*]
      x := q + y * (r * z10 + t * z11)
```

Figure 2–14. Les premiers programmes EVA de la comparaison.

Langages	Codes					
	1 (m1)	1 (mv1)	1 (mvf1)	2 (mc1)	2(mc2)	3
CFT impl.	665	758	758			1927
CFT expl.	665	754	754			1927
EVA/DEVIL	665	753	753			1919

Figure 2-15. Temps d'exécution comparés des premiers tests pour les trois langages.

Les trois premières comparaisons correspondent à des séquences d'instructions de complexité croisante. Les temps obtenus pour ces trois comparaisons sont résumés à la figure 2-15. Nous ne notons pas de différence significative dans l'exécution des ces codes entre EVA/DEVIL et FORTRAN. Le schéma d'exécution est semblable pour les trois langages. Notons bien que pour l'extrait de code *hydro* (3), les accès aux éléments du vecteur *z* sont réalisés via deux vecteurs décrits, *z10* et *z11*. Le temps obtenu pour ce code inclut la création de ces deux vecteurs. Cette création de vecteur est donc amortie pour les traitements réalisés ici (vecteur de longueur 10 000).

Le code 4 est une instruction d'affectation masquée. Les programmes sources comparés sont donnés à la figure 2-16. Les résultats de cette comparaison sont aussi résumés sur cette figure.

Code EVA

```
vect real a, b, c
     c := (a == b) ? a ! b
```

Code FORTRAN implicite

```
      DO 10 i = 1, size
         IF (a (i) .EQ. b (i)) THEN
           c (i) = a (i)
         ELSE
           c (i) = b (i)
         ENDIF
10    CONTINUE
```

Langage	Temps d'exécution
CFT impl.	164
CFT expl.	137
EVA/DEVIL	137

Code FORTRAN explicite

```
c = CVMGT (a, b, a .EQ. b)
```

Figure 2-16. Code 4 : affectation masquée.

Les performances obtenus par EVA/DEVIL et FORTRAN explicite sont identiques. Cependant, le code FORTRAN doit être écrit à l'aide de la fonction intrinsèque `CVMGT` qui est spécifique au FORTRAN CFT, donc non-portable. De plus, l'appel de cette fonction doit préciser une expression de type `LOGICAL`. Une telle expression résulte d'un opérateur de comparaison (ici le `.EQ.`) ou d'une variable *logical*. Toutefois, l'utilisation d'une variable *logical* à ce niveau oblige à placer la fonction dans une boucle itérant sur les indices des tableaux. En effet, une variable *logical* représente 64 valeurs binaires (codées sur un mot de 64

bits); la représentation d'un nombre plus important de valeurs booléennes n'est pas possible avec de telles variables.

La faible performance de FORTRAN explicite s'explique par la représentation mémoire utilisée pour la valeur de l'expression booléenne. Une telle valeur est représentée comme un tableau d'entiers, c'est-à-dire qu'une valeur booléenne est codée sur un mot de 64 bits ! Cette représentation est la source de transferts mémoire plus importants et donc de performances moindres.

Le cinquième code consiste en une suite d'affectations vectorielles décrites par une même liste de valeurs booléennes. Les codes EVA et FORTRAN, ainsi que les temps mesurés sont présentés à la figure 2–17. Nous remarquons la non-portabilité du code FORTRAN explicite et la faible performance du code FORTRAN implicite. Cette dernière s'explique à nouveau par la représentation de la liste de valeurs booléennes en mémoire.

Code EVA

```
vect real a, b, c, d, e, f, x, y, z
     with [mask]      ..
       c := a + b     ..
       f := d + e     ..
       z := x + y
     end
```

Code FORTRAN implicite

```
      DO 10 i = 1, size
        IF (mask (i)) THEN
          c (i) = a (i) + b (i)
          f (i) = d (i) + e (i)
          z (i) = x (i) + y (i)
        ENDIF
10 CONTINUE
```

Code FORTRAN explicite

```
c = CVMGT (a + b, c, mask)
d = CVMGT (e + f, e, mask)
z = CVMGT (x + y, z, mask)
```

Langage	Temps d'exécution (*size*=100 000)
CFT impl.	9000
CFT expl.	4570
EVA/DEVIL	4550

Figure 2–17. Code 5 : bloc **with**.

En conclusion à ces quelques tests, nous insisterons sur la bonne qualité du code généré par DEVIL depuis EVA et l'expression claire des instructions au niveau EVA. EVA introduit une notion puissante de vecteur, dont nous avons vu que l'usage n'est pas pénalisant pour des vecteurs de grande taille (code *hydro* avec 10 000 éléments). Nous allons affiner ces mesures dans les paragraphes suivants.

Coût de la structure vecteur EVA/DEVIL

Nous tentons ici de déterminer le coût de la gestion de la structure de vecteur EVA/DEVIL. Pour une suite d'instructions données, ce coût est constant, quelle que soit la longueur de traitement des vecteurs. Nous

comparons donc les performances d'une instruction donnée pour des longueurs de traitement variables. L'instruction comparée est le code 3 (extrait de *hydro*). Ce code est placé dans une boucle de 1000 itérations pour que les mesures soient possibles sur des petites longueurs de vecteurs. La longueur de traitement varie de 10 à 3000 éléments. Les résultats d'exécution sont résumés à la figure 2–18.

Longueur	10	20	50	100	200	500
EVA-DEVIL	2204	2289	2745	3869	5813	12587
CFT	529	704	1050	2209	4240	9710
EVA-DEVIL/CFT	4,17	3,25	2,61	1,75	1,37	1,29
Longueur	1000	1500	2000	2500	3000	
EVA-DEVIL	21696	30172	41141	50482	60203	
CFT	19528	29598	41065	50357	60207	
EVA-DEVIL/CFT	1,11	1,02	1,00	1,00	1,00	

Figure 2–18. Coût de la gestion de la structure de vecteur EVA/DEVIL.

Pour EVA/DEVIL, les temps donnés correspondent à 1000 itérations (scalaires) d'une boucle comportant deux instructions de création d'un vecteur (*z10* et *z11*) par association du vecteur *z* et d'un triplet (cette opération réalise une allocation dynamique de la zone de description du triplet précédée d'une désallocation automatique de l'ancienne zone de description pour chaque itération de la boucle) et au calcul de *hydro* proprement dit. Les temps mesurés pour FORTRAN correspondent à l'exécution de 1000 itérations du calcul de *hydro*. Le coût de gestion de la structure de vecteur EVA/DEVIL est ainsi mesuré. Ce coût n'est significatif que pour des valeurs de la longueur inférieure à 200 pour cet exemple. A partir d'une longueur de 1000, la gestion de la structure de vecteur est amortie. A l'évidence, les traitements réalisés ici sur les vecteurs sont minimes. Pour des traitements plus importants, le coût relatif de la gestion de la structure vecteur diminue.

Conclusion

Les éléments de performance donnés ici sont pleinement satisfaisants. Les performances obtenues sont généralement comparables à celles des codes FORTRAN. Dans certaines cas, une représentation mémoire des vecteurs moins gourmande que celle de CFT, autorise une génération de code de meilleure qualité; on évite de nombreux transferts mémoire. La pénalisation induite par la structure de vecteur EVA/DEVIL disparaît vite quand la taille des vecteurs traités augmente. Cependant, dans cette évaluation, nous n'avons considéré que des vecteurs dont les attributs étaient pleinement spécifiés. Dans les cas où cette condition ne serait pas respectée, il nous faudrait introduire des tests à l'exécution préalablement aux traitements des vecteurs. Deux aspects mèneraient alors à une possible dégradation des performances : l'exécution de ces tests et des instructions de saut associées et l'augmentation de la taille du code, source d'éventuels rechargement des *buffers* d'instructions. Toutefois, la comparaison avec FORTRAN CFT ne doit pas seulement être faite en terme de performance, nous pouvons également comparer les possibilités d'expression dans le langage source et la complexité du processus de compilation.

2–17 Conclusion

Ce second chapitre a donc consisté en une présentation du langage vectoriel EVA, de sa compilation en DEVIL, langage vectoriel intermédiaire, et de son implantation sur machine vectorielle pipe-line type Cray Y-MP mono-processeur.

EVA est un langage vectoriel explicite; il laisse le programmeur exprimer son algorithme à un niveau de parallélisme compatible avec celui-ci (mais aussi avec celui des machines cibles). De plus, le langage EVA tente de fournir au programmeur des moyens d'expression des connaissances qu'il détient sur son programme. Cette approche autorise la mise en place simple de compilateurs ne nécessitant pas de phase d'optimisations avancées.

Au centre de EVA se trouve une structure de données particulièrement adaptée à la programmation vectorielle : le vecteur EVA. Un vecteur EVA est la réunion d'une zone d'allocation contenant les composantes du vecteur et d'une zone de description précisant la liste des éléments du vecteur parmi ces composantes. Les opérations vectorielles classiques de description sont donc intégrées dans les vecteurs EVA. Les vecteurs, sous cette forme, sont manipulés par toutes les instructions EVA. Une zone d'allocation est partageable entre plusieurs vecteurs. Ce partage de composantes définit des "sous-vecteurs"; ils référencent et mettent à jour les valeurs des composantes d'un même vecteur via des descripteurs différents. EVA intègre un opérateur, dit d'association, qui alloue et initialise les zones des vecteurs; il réalise aussi le lien entre un vecteur et ses zones d'allocation et de description. Nous définissons aussi des moyens d'expression et de contrôle simple des longueurs sur lesquelles traiter les vecteurs. Nous fournissons différents opérateurs d'affectation vectorielle exprimant ou non la dépendances entre les parties droite et gauche de l'affectation; la non dépendance peut aussi être exprimée entre deux instructions EVA. Nous proposons le **`with`**, un constructeur généralisant le **`where`**, habituelle extension vectorielle du **`if`**.

La compilation du langage EVA est assurée via le langage intermédiaire vectoriel DEVIL. Les instructions DEVIL traitent des vecteurs tels ceux définis dans EVA. Toutefois, associé à chaque opérande vectoriel DEVIL, des attributs générés par EVA précisent les caractéristiques du vecteur. Des blocs parallèles sont (syntaxiquement) identifiés en DEVIL; un bloc parallèle est une suite d'instructions qu'aucune dépendance, autre que des dépendances de flot, ne lie. La génération de DEVIL est axée autour des blocs parallèles; les instructions d'un bloc parallèle sont découpées ensemble en une boucle unique d'instructions vectorielles traitant des tranches de vecteurs de la taille des registres vectoriels (boucle de "strip-mining").

Le compilateur implanté sur Cray Y-MP produit, en général, des performances comparables à celles obtenues par les compilateurs vectorisants de la machine (C et FORTRAN). Sur certains points pour lesquels la structure de EVA/DEVIL est particulièrement adaptée, le gain en performance est sensible (génération de blocs d'instructions EVA à longueur explicite, utilisation du constructeur EVA **`with`**, spécification de la non-dépendance entre instructions EVA). Il est notablement intéressant de comparer la simplicité des compilateurs EVA et DEVIL et la qualité du code pouvant être produit.

Il reste quelques points délicats dont la prise en compte n'est pas pleinement satisfaisante en EVA/DEVIL. Il s'agit principalement du passage de vecteur en paramètre, dans leur forme EVA/DEVIL. L'information passant entre les deux fonctions appelantes et appelées est trop faible et les attributs de vecteurs restent de part et d'autre incomplètement spécifiés. Nous avons proposé (section 2–15.10) des idées pour la prise en compte de ces attributs entre les fonctions; nous reviendrons sur ce problème dans le cadre de LSD2.

Les langages EVA et DEVIL ont été conçus en visant les machines mono-processeurs vectorielles pipelines. De nouvelles constructions et instructions sont à définir pour élargir les cibles de EVA aux architectures qui tendent à se généraliser maintenant, à savoir les multi-processeurs pipelines vectoriels et les machines massivement parallèles. Pour les machines multi-processeurs, il nous faut soit intégrer des primitives de gestion du parallélisme spécifiques au traitement vectoriel, soit distinguer, par exemple au sein des blocs parallèles, des portions de codes pouvant former des tâches parallèles. Pour les machines massivement parallèles, il faut, pour le moins, prendre en compte la spécificité essentielle de ces machines, à savoir la distribution des données entre les processeurs élémentaires nécessitant la production de communications entre ces processeurs. Suivant notre démarche, il nous faut définir les informations nécessaires à cette prise en compte, proposer des outils au programmeur pour exprimer ces informations dans son programme, et prendre en compte ces spécifications lors de la génération de code. Une telle approche est entreprise au sein d'un nouveau langage : LSD2.

Chapitre 3

Le langage LSD2

L'évolution des machines à parallélisme de données tend à imposer les machines vectorielles pipelines multi-processeurs et les machines massivement parallèles. La prise en compte de ces machines, bien que prévue depuis EVA, ne fut pas mise en place faute de spécificité du langage vis-à–vis de ces machines. L'extension de EVA à ces classes de machines est un langage nommé LSD2. Notre approche pour la conception de LSD2 est identique à celle de EVA : nous fournissons au programmeur les moyens d'exprimer les connaissances qu'il détient pour aider la compilation et la production de code efficace. Cependant, contrairement à EVA, les constructions proposées par LSD2 s'intègrent à un langage existant; LSD2 est un langage "embedded". LSD2 propose une notion de type vecteur inspirée des vecteurs de EVA. Un objet vecteur regroupe toutes les spécifications et informations concernant ce vecteur. Une première présentation de ce type vecteur est fournie. Le langage LSD2 est développé au sein de l'environnement PARTNER qui regroupe aussi des langages intermédiaires et définit les passages entre ces langages. Une seconde section traite de ces différents niveaux de langages intermédiaires. La section suivante consiste en une présentation du langage LSD2 lui-même, nous détaillons alors les vecteurs LSD2 et leur manipulation au sein du langage. Parmi les spécifications incluses dans les vecteurs LSD2, se trouve l'expression de dépendances entre différents accès aux éléments d'un vecteur. Des dépendances qualifiées de partielles ont été exhibées. Nous présentons la génération de code depuis ces informations de dépendances partielles dans une dernière section; des résultats de performances d'une évaluation des dépendances partielles sont aussi fournis.

Une première description de PARTNER se trouve dans [DMP91a]. Une présentation de la programmation en LSD2 et des dépendances partielles entre parties droite et gauche d'une affectation vectorielle sont disponibles dans [DMP91b] et [DMP91c] respectivement.

Chapitre 3—Section 1

Le langage LSD2

3–1 Le vecteur LSD2

La principale construction proposée par LSD2 est le vecteur LSD2. Nous discutons ici des caractéristiques de ces vecteurs et donnons un aperçu des spécifications regroupées au sein d'un vecteur. La définition des vecteurs LSD2 est inspirée des vecteurs du langage EVA; nous retrouvons les deux zones d'allocation et de description. Cependant, un vecteur LSD2 regroupe tous les vecteurs EVA partageant la même zone d'allocation. De plus, les descriptions (nommées *filtres*) ne sont plus représentées en mémoire, mais uniquement utilisées pour réaliser les accès aux éléments du vecteur. La forme des vecteurs LSD2 est plus riche; un vecteur LSD2 contient d'autres informations et caractéristiques que nous nommons *spécifications*. Un vecteur LSD2 est donc constitué de (figure 3–1) :

- Une zone d'allocation. Cette zone d'allocation regroupe les éléments du vecteur. Une telle zone peut être comparée à la zone d'allocation d'un vecteur EVA ou à un tableau FORTRAN.
- Des filtres d'accès aux éléments du vecteur. Les filtres sont le moyen d'accéder à des ensembles de composantes parmi les éléments de la zone d'allocation du vecteur. Il n'existe pas d'opérateur de description vectorielle en LSD2; tous les accès aux éléments des vecteurs sont réalisés via les filtres. Un filtre est une description par un scalaire entier, ou par une séquence d'index, ou par une séquence de bits, ou par un triplet *borne inférieure, borne supérieure, pas*. Les filtres LSD2 incluent aussi des opérations d'accès aux éléments de vecteur par réduction et diffusion.
- Des spécifications de dépendances entre les filtres. Ces spécifications caractérisent la dépendance ou la non dépendance entre les filtres du vecteur. De plus nous avons mis en évidence des dépendances partielles entre deux filtres; elles sont aussi exprimées par ces spécifications.
- Des spécifications de distribution et d'alignement. Ces spécifications indiquent des relations fortes entre des groupes d'éléments du vecteur. Sur une machine massivement parallèle, elles

détermineront l'allocation des éléments du vecteur sur les processeurs élémentaires en vue de minimiser les communications inter-processeurs.

- Des spécifications de projection associées à chaque dimension d'un vecteur multi-dimensionnel. Une telle spécification associe un poids à chaque dimension d'un vecteur multi-dimensionnel. Le programmeur indique par ces poids, un ordre dans l'intérêt de considérer telle ou telle dimension comme effectivement parallèle dans l'implantation. (En effet pour une implantation sur une machine donnée dont le parallélisme est inférieur au nombre de dimensions du vecteur, certaines dimensions de ce vecteur ne peuvent être traitées en parallèle.)

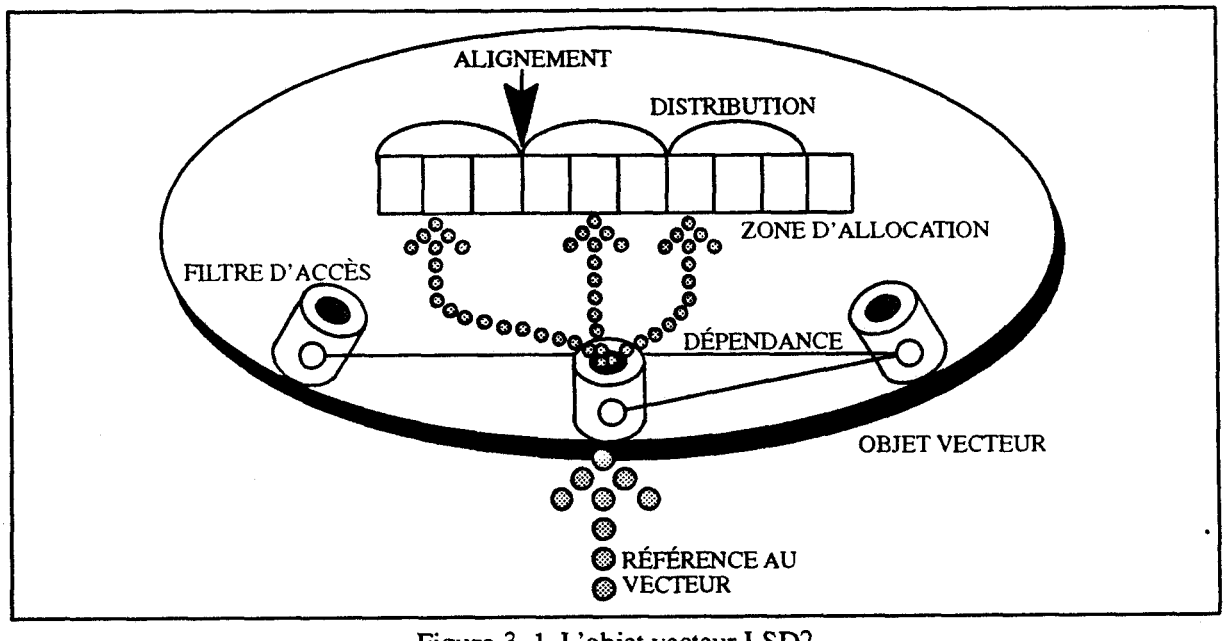


Figure 3–1. L'objet vecteur LSD2

Les spécifications sont optionnelles. Plusieurs avantages sont liés à cette caractéristique et au fait que les spécifications soient encapsulées dans le vecteur.

Les spécifications sont le moyen pour le programmeur d'indiquer des informations connues de lui sur les données que manipule son programme. En l'absence de ces informations, le programme est valide et correct. Ces spécifications sont utilisées par le compilateur pour améliorer la qualité du code produit. Nous séparons ainsi l'expression de l'algorithme de la phase de recherches de performances. Cette séparation facilite la programmation et améliore la lisibilité des programmes. Les algorithmes sont implantés en premier lieu. Ensuite, l'expression ou le raffinement des spécifications augmente l'efficacité du programme.

L'encapsulation des spécifications dans les objets vecteurs autorise un typage fort des vecteurs LSD2. En LSD2, les vecteurs peuvent être passés en paramètres aux fonctions. Le typage fort est alors un avantage certain. Dans une fonction, les caractéristiques des vecteurs reçus en paramètre sont ainsi entièrement connues. Aucune opération ou optimisation n'est à mettre en place pour assurer la justesse et l'efficacité du code produit, chose impossible en l'absence de typage fort.

L'allocation en mémoire des éléments d'un vecteur est dépendante des spécifications de distribution et d'alignement. Considérons le passage d'un vecteur en paramètre et supposons que les spécifications ne fassent pas partie du type du vecteur. La fonction appelée ne connaît alors pas l'allocation des éléments du vecteur. Plusieurs solutions sont envisageables pour remédier à cette situation.

- Une analyse inter-procédurale, éventuellement associée à la mise en ligne de la fonction appelée, autorise la prise en compte des spécifications des vecteurs entre fonctions appelante et appelée et la génération de code en fonction de ces informations. Une telle analyse suppose un ensemble d'outils de gestion de code source.

- Le passage par une forme normale de vecteur entre fonction appelante et appelée ne demande aucune analyse. Il est réalisé sur le schéma suivant : le paramètre effectif est copié selon une forme normale, cette copie est référencée par le paramètre effectif de la fonction qui, éventuellement, réalise encore une copie pour mettre le vecteur sous la forme indiquée par les spécifications de la déclaration de ce paramètre. La même opération est réalisée en sens inverse lors de la fin de l'exécution de la fonction. La pénalisation au niveau des performances est évidente. Elle est d'autant plus forte sur une machine massivement parallèle que la copie entre un vecteur alloué en fonction de spécifications de distribution et d'alignement et une forme normale (mettons l'allocation cyclique d'un élément de vecteur par PE) entraîne des communications inter-processeurs.

Les caractéristiques et avantages des vecteurs LSD2 sont résumés ainsi : notion puissante de structure de données bâtie sur le modèle des vecteurs EVA (zone d'allocation et filtres), expression de spécifications autorisant la production de code efficace, encapsulation de ces spécifications dans les objets vecteurs autorisant un typage fort, séparation entre l'expression des spécifications et l'expression de l'algorithme facilitant l'écriture des programmes.

3–2 L'environnement de production de programmes à parallélisme de données PARTNER

Nous présentons ici PARTNER. PARTNER est un environnement de production de programmes à parallélisme de données. Les deux modèles d'architectures à parallélisme de données (machines massivement parallèles et machines vectorielles pipelines) sont simultanément pris en compte au sein de PARTNER.

PARTNER est constitué de langages d'expression du parallélisme de données de différents niveaux et d'outils de traduction entre ces niveaux. Nous distinguons :

- LIVP et LIMP sont deux langages machines virtuelles respectivement pour les machines vectorielles pipelines multi-processeurs et les machines massivement parallèles. C'est à ce seul niveau que les deux types de machines sont distingués.

- LOLLA est un langage intermédiaire intégrant toutes les caractéristiques nécessaires à la projection sur les deux types de machines via LIVP et LIMP.

- LSD2 est un langage de haut niveau dédié à la manipulation explicite de vecteurs. Les programmes LSD2 produisent du code LOLLA.

La figure 1 indique les traductions à réaliser entre ces trois niveaux de langages.

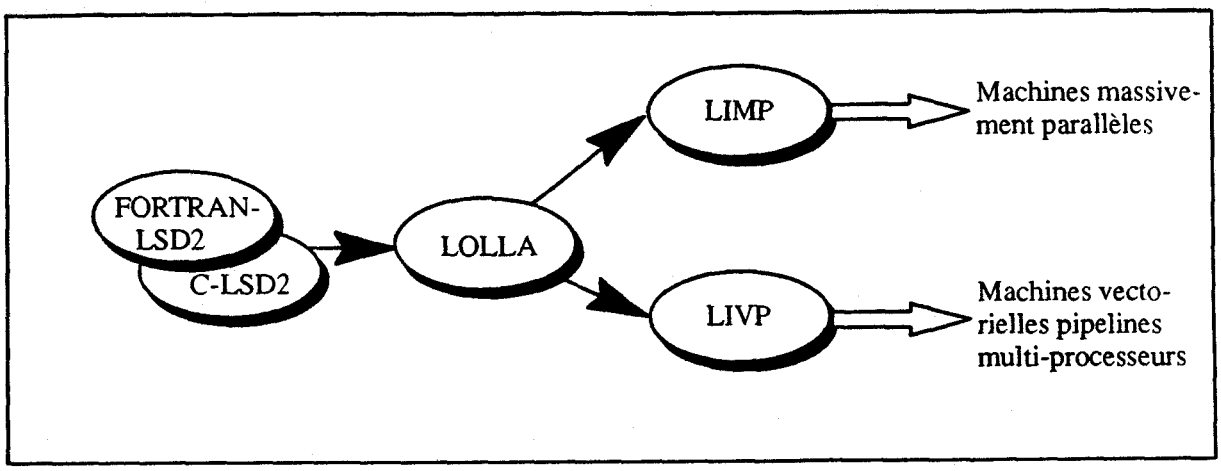


Figure 3–2. L'environnement PARTNER.

3–2.1 Les langages machines virtuelles LIMP et LIVP

Pour les machines pipelines vectorielles comme pour les machines massivement parallèles, notre approche est basée sur la définition de deux machines virtuelles. Nous associons à chacune d'elle un langage machine virtuelle, respectivement LIVP et LIMP. Ces deux langages représentent le parallélisme de données de manière à tirer avantage des caractéristiques des machines ciblées. Les traductions de LOLLA en LIMP et LIVP sont basées sur des algorithmes spécifiques en fonction du langage ciblé. Par exemple :

- La génération de code pour la réalisation des communications inter-processeur est réalisée par la traduction de parallélisme de données exprimé en LOLLA dans des instructions LIMP de communications. Ces communications sont directement liées aux filtres et autres informations contenues dans les vecteurs LSD2. Ce schéma de génération est spécifique à LIMP; il est inapproprié pour les machines vectorielles pipelines.
- Le jeux d'instructions LIVP inclut des opérations de gather/scatter et d'accès par pas. Ces opérations sont générées à partir des opérations de description du langage de haut niveau exprimée via les filtres des vecteurs LSD2.

3–2.2 Le langage de bas niveau LOLLA

LOLLA est un langage de bas niveau autorisant la projection sur machines massivement parallèles et vectorielles pipelines de langages à parallélisme de données de haut niveau. La vocation de LOLLA est d'être un langage point d'entrée d'un système de compilation pour machines à parallélisme de données. La projection d'un langage sur un ensemble de machines est réalisée par la traduction de ce langage en LOLLA. Deux nécessités antinomiques ont donc guidé la conception de LOLLA :

- LOLLA se doit d'être d'un niveau le plus élevé possible pour diminuer le coût de cette phase de traduction depuis les langages de haut niveau.
- A l'inverse, LOLLA doit être d'un niveau relativement bas pour autoriser la projection de tous les langages pouvant potentiellement l'utiliser.

Bien qu'en marge de nos travaux actuels, la projection de langages tel FORTRAN 90 (éventuellement complété d'instructions de distribution et d'alignement des tableaux du type de celle existante en FORTRAN D) doit rester possible sur LOLLA (figure 3–3). Ce point fut un soucis permanent lors de la conception de LOLLA.

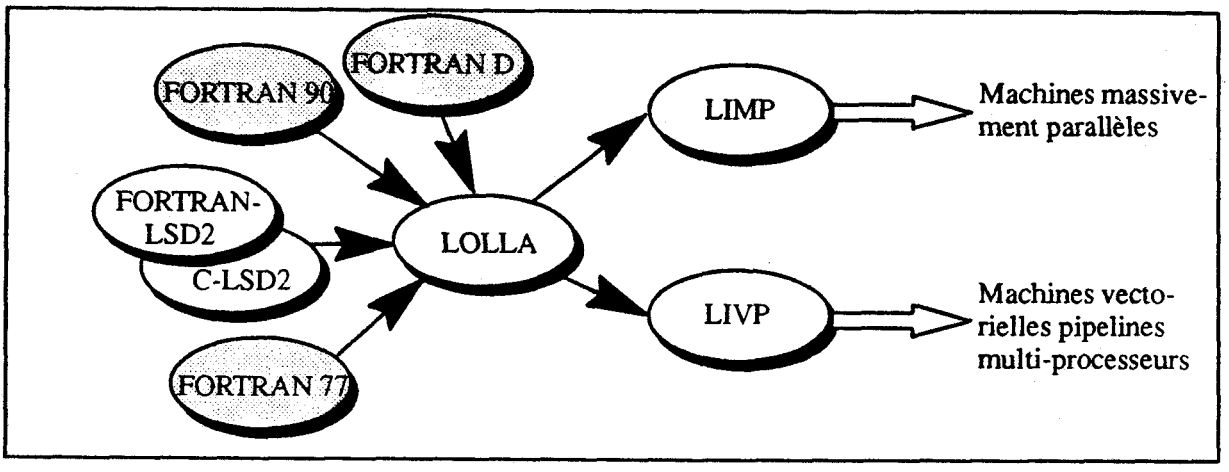


Figure 3–3. Projection de langages de haut niveau via LOLLA.

A partir de ces considérations, nous avons conçu LOLLA sur deux caractéristiques essentielles :

- LOLLA est un langage (de bas niveau) à parallélisme de données. Les instructions manipulent des vecteurs.
- Les informations de distribution, d'alignement, de dépendances et autres spécifications incluses dans les vecteurs LSD2 ne sont pas consommées au niveau de LOLLA.

LOLLA manipule des objets vecteurs et scalaires tels que ceux définis en LSD2. Ces objets sont manipulés via leur nom; il n'y a pas d'allocation des objets sur une mémoire virtuelle à ce niveau. Ce nommage des objets oblige la mise en place d'une notion de visibilité des noms et donc l'introduction d'une notion de blocs syntaxiques.

Un programme LOLLA est représenté par un graphe de flots. Les nœuds de ce graphe sont des blocs d'instructions. A chaque nœud est associée une table des symboles locaux à ce nœud. Dans cette table, nous retrouvons aussi les caractéristiques des vecteurs LSD2, à savoir les distributions, poids, alignements, filtres, et dépendances entre les filtres utilisés dans le nœud.

Les instructions LOLLA sont représentées par un langage de triplets. Une instruction LOLLA produit un objet temporaire résultat de son exécution. Hormis ces temporaires, les instructions manipulent les vecteurs et les filtres tels qu'ils sont définis en LSD2.

Les instructions vectorielles LOLLA (c'est à dire les instructions manipulant et/ou produisant des vecteurs) ne peuvent apparaître qu'au sein d'un *bloc vectoriel*. Un bloc vectoriel définit un ensemble d'instructions traitant des vecteurs de même taille. Une notion de fonction et de passage de paramètres existe en LOLLA. Les constructions de blocs **WHERE** et leur extension LSD2 (cf. infra la définition de LSD2) sont reproduites en LOLLA.

Nous résumons les caractéristiques de LOLLA par la définition de LOLLA comme une forme simple de LSD2, autorisant aussi la projection de FORTRAN 90, dans laquelle les expressions sont sous une forme primitive, et autorisant la projection sur les différents types de machines à parallélisme de données.

3–3 Le langage LSD2

Nous présentons dans cette section les constructions de LSD2 et les raisons qui ont motivé la mise en place de celles-ci. Notre approche est de fournir avec LSD2 un langage de haut niveau dans lequel il soit possible au programmeur d'exprimer des informations nécessaires à la production de code efficace. Une fois les informations pertinentes définies, nous ne nous attachons pas à construire des méthodes de découverte automatique de ces informations, mais à déterminer des outils d'expression de ces informations.

Les constructions proposées par LSD2 expriment le parallélisme des données. Nous les avons définies indépendamment de tout langage et de toute architecture spécifique. Le programmeur exprime son algorithme au travers des constructions de haut niveau traduisant le parallélisme des données. Ces deux aspects sont des avantages certains lors de développement de programmes.

LSD2 est un langage "embedded"; l'utilisation des constructions est réalisée de manière similaire au sein d'un langage hôte au choix du programmeur (cependant les langages C et FORTRAN sont les premiers candidats pour cet emploi). Un autre avantage de cet aspect est la possibilité de réécriture progressive, en LSD2, de programmes C ou FORTRAN existants. En effet, les programmes C et FORTRAN sont des programmes LSD2 valides. L'introduction graduelle de constructions LSD2 est donc facilitée.

Les constructions LSD2 expriment le parallélisme de données; elles sont indépendantes de la machine cible et du type de cette machine (massivement parallèle ou pipeline vectorielle). La projection sur la machine cible via LOLLA et LIMP ou LIVP est la seule à prendre en compte les caractéristiques et spécificités de la machine cible. Par exemple, la programmation en LSD2 n'inclut pas de primitives de communications entre les processeurs d'une machine massivement parallèle; ces instructions sont retrouvées depuis les opérations de descriptions vectorielles induites par l'utilisation de filtres de vecteurs.

LSD2 est un jeu de déclarations, de primitives et d'expressions utilisables dans un langage hôte. Les déclarations de vecteurs et de types vecteurs sont incluses dans la partie déclarative du langage hôte (ou dans celle d'un bloc d'instructions LSD2). Les expressions LSD2 sont utilisées dans les instructions du langage hôte.

Le reste de cette section détaille les constructions proposées par LSD2. Il s'agit principalement de définitions de types vecteurs LSD2, de déclarations et utilisations de vecteurs et de regroupement d'instructions en bloc.

3–3.1 Définitions de vecteurs en LSD2

Les constructions, instructions et opérations LSD2 sont bâties autour de la notion de vecteur LSD2 présentée à la section 3–1. Cette section détaille la définition de vecteur en LSD2.

Différents types de vecteurs sont distingués en LSD2 en fonction de leur degré de dynamicité. Nous commençons par la description de déclaration de vecteurs statiques, qui sont des vecteurs dont les caractéristiques sont déterminées à la compilation et ne varient pas lors de l'exécution; nous présentons ensuite les vecteurs automatiques et les vecteurs dynamiques; nous introduisons les types vecteurs LSD2.

3–3.1.1 Les spécifications incluses dans les vecteurs LSD2

Nous donnons ici la liste des spécifications pouvant être incluses dans un vecteur LSD2. Un vecteur LSD2 est défini par la réunion d'un ensemble de triplets formés

- d'une taille,
- d'une distribution, et
- d'un alignement.

Les vecteurs sont multi-dimensionnels, ils sont donc caractérisés par

- leur nombre de dimensions.

Ce nombre correspond au nombre de triplets. Chaque triplet définit les caractéristiques du vecteur selon une dimension donnée. De plus, le vecteur est aussi caractérisé par

- une projection.

Pour un vecteur donné, ces informations sont nécessaires à son allocation. Elles sont dans la majorité des cas fournies lors de la déclaration du vecteur. Des valeurs par défaut existent pour les informations de distribution, d'alignement et de projection. Les vecteurs déclarés avec ces spécifications par défaut sont qualifiés de *canoniques*.

Toute utilisation d'un vecteur oblige aussi la connaissance des informations de distribution, d'alignement, et de projection. L'information de taille n'est pas toujours nécessaire. Par exemple, l'accès aux 20 premiers éléments d'un vecteur n'utilise pas l'information de taille.

De plus, les informations suivantes peuvent être associées à un vecteur :

- les filtres d'accès aux éléments du vecteur, et
- les spécifications de dépendances entre les filtres.

Ces informations sont utilisées par le compilateur lors de la génération de code pour, d'une part réaliser les accès aux éléments du vecteur, et d'autre part améliorer la qualité du code généré.

3–3.1.2 Vecteurs statiques

La déclaration d'un vecteur associe un certain nombre de caractéristiques à un nom. Dans le cadre des vecteurs statiques, elle autorise aussi la réalisation de l'allocation du vecteur. Les informations devant être précisées sont celles caractérisant un vecteur, à savoir :

- le type des éléments du vecteur, et
- la taille du vecteur, c'est-à-dire la taille du vecteur selon chacune de ses dimensions.

D'autres informations sont aussi éventuellement associées à un vecteur, elles sont plus spécifiques à LSD2. Ce sont

- les spécifications de distribution, d'alignement et de projection.

Si la déclaration ne comporte pas de telles spécifications, les valeurs par défaut sont utilisées.

La déclaration d'un vecteur statique nécessite une expression statique, c'est-à-dire dont la valeur peut être déterminée lors de la compilation, de toutes ces informations. Considérons par exemple la déclaration du vecteur *VSTATIC1*

```
VECTOR REAL VSTATIC1
      SIZE        1000
      DISTRIBUTION 2
END
```

Elle définit la taille du vecteur mono-dimensionnel à 1000 et sa distribution de 2. La valeur par défaut est utilisée pour la spécification d'alignement. La déclaration d'un vecteur multi-dimensionnel spécifie des listes de spécifications :

```
VECTOR REAL VSTATIC2
      SIZE        1000, 2560
      DISTRIBUTION -, 2
END
```

les valeurs par défaut sont référencées par le signe moins -.

De manière évidente, l'allocation des vecteurs statiques peut être définie dès la compilation, les valeurs de toutes les informations étant connues dès cette étape.

3–3.1.3 Déclaration de filtre d'accès

Une déclaration associe éventuellement des filtres d'accès et des spécifications de dépendances entre les filtres au vecteur. Nous ne détaillons pas les différents types d'opérations réalisables au travers les filtres de vecteurs (se référer à la section NO TAG), mais donnons ici la forme des déclarations de filtres en LSD2. Par exemple, la déclaration de vecteur

```
VECTOR REAL VFILTRE1
      SIZE        1000
      FILTER      IMPAIR      [1 : 1000 : 2]
      FILTRE      PAIR        [2 : 1000 : 2]
      NODEPENDENCE IMPAIR, PAIR
      DISTRIBUTION 2
END
```

définit un vecteur de 1000 éléments contenant deux filtres d'accès *PAIR* et *IMPAIR* référençant respectivement les éléments d'indice pair et impair du vecteur. La spécification de non-dépendance indique qu'il n'existe pas d'élément du vecteur pouvant être référencé au travers les deux filtres.

Un filtre de vecteur est obtenu par l'application d'une description au vecteur. Il est aussi possible d'appliquer plusieurs descriptions ou d'appliquer une description à un autre filtre du vecteur. Par exemple,

```
INTEGER INDEX

VECTOR REAL VFILTRE2
      SIZE        1000
      FILTER      IMPAIR      [1 : 1000 : 2]
      FILTER      PAIR        [2 : 1000 : 2]
      FILTER      ALEAPAIR    [2 : 1000: 2] [INDEX]
      FILTER      ALEAIMPAIR  IMPAIR [INDEX]
END
```

Le filtre *ALEAPAIR* référence le *INDEX*^{ième} élément de rang pair, *ALEAIMPAIR* celui de rang impair.

Les filtres d'accès sont paramétrables. Leur utilisation sera alors elle aussi paramétrée. Par exemple

```
FILTER      FIRST (J)          [1 : J]
FILTER      TRANCHE (I)        [I : I + 20]
```

Cette construction est principalement syntaxique, en particulier, le type des paramètres n'est pas défini lors de la déclaration.

Les spécifications de dépendances peuvent être conditionnelles. Une telle spécification exprime une condition sur les paramètres des filtres sous laquelle la dépendance est effective. Par exemple,

```
IF (I .GT. J) NODEPENDANCE  TRANCHE (I), FIRST (J)
```

Une telle spécification conditionnelle de dépendance oblige, lors de la compilation d'une instruction utilisant la spécification, à la génération de deux codes. L'exécution du code généré pour le cas où la spécification est juste ou de celui pour le cas où elle n'est pas juste est déterminée par la valeur de la condition préalablement à cette exécution.

3–3.1.4 Vecteurs automatiques

A l'inverse des vecteurs statiques, les valeurs des informations associées à un vecteur automatique ne sont pas connues avant l'exécution. Par exemple, la taille d'un tel vecteur dépend de la valeur d'une variable ou de celle d'un paramètre.

```
INTEGER N
...
VECTOR REAL VAUTO1
      SIZE        N
      FILTER      IMPAIR      [1 : N : 2]
      FILTER      PAIR        [2 : N : 2]
      NODEPENDENCE IMPAIR, PAIR
END
```

La valeur, lors de la déclaration du vecteur, de la variable *N* détermine la taille du vecteur et les caractéristiques précises des filtres *PAIR* et *IMPAIR*. Donnons un autre exemple :

```
INTEGER DIST
...
VECTOR REAL VAUTO2
      SIZE        1000
      FILTER      IMPAIR      [1 : 1000 : 2]
      FILTER      PAIR        [2 : 1000 : 2]
      NODEPENDENCE IMPAIR, PAIR
      DISTRIBUTION DIST
END
```

La distribution du vecteur n'est définie qu'à l'exécution, en fonction de la valeur de la variable *DIST*.

Sur une machine massivement parallèle, l'allocation de ces vecteurs ne peut être définie dès la compilation. Ces vecteurs seront donc alloués *automatiquement*, à l'exécution, sur un tas. Ces vecteurs seront aussi automatiquement désalloués après leur dernière utilisation.

Deux aspects contribuent à diminuer les performances lors de l'utilisation de tels vecteurs. Le coût de l'allocation dynamique ne peut être négligé. Mais de plus, sur une machine massivement parallèle par exemple, les références aux éléments du vecteur dépendent des spécifications de distribution, alignement et projection. La non connaissance de ces informations dès la compilation transfère la réalisation d'une partie du "calcul d'adresses" à l'exécution.

3-3.1.5 Vecteurs dynamiques

Les vecteurs LSD2 dont les caractéristiques ne sont ni connues lors de la compilation, ni déterminées lors de leur déclaration sont dynamiques. En effet, la taille d'un vecteur et les spécifications de distribution, alignement et projection peuvent ne pas être précisées lors de la déclaration du vecteur, elles le seront dans le code.

En LSD2, les vecteurs dynamiques doivent être identifiés comme tels lors de leur déclaration. La spécification d'une des informations de taille, de distribution, d'alignement, ou de projection comme dynamique identifie un vecteur dynamique. Une telle spécification est définie comme dynamique par le mot clé **DYNAMIC**. Par exemple, le vecteur déclaré

```
VECTOR INTEGER VDYN1
      SIZE              DYNAMIC
      DISTRIBUTION      DYNAMIC
END
```

est défini avec une distribution et une taille non précisées. (Notons que l'alignement du vecteur est obtenu par la valeur par défaut.) Un tel vecteur n'est pas alloué. Il ne pourra l'être que lors de la définition de toutes ses spécifications. Cette définition est réalisée via l'affectation des champs *DISTRIBUTION*, *ALIGNEMENT*, *PROJECTION*, et *SIZE* du vecteur. Par exemple, on peut écrire

```
VDYN1. DISTRIBUTION = 2
VDYN1. SIZE = 1000
```

Ces affectations ne produisent aucune allocation. Celle-ci est réalisée explicitement par les primitives **ALLOC** et **REALLOC**, en fonction des valeurs affectées aux différents champs ou des anciennes valeurs des champs non affectés. Ces primitives associent une nouvelle zone d'allocation au vecteur. Par exemple

```
VDYN1. DISTRIBUTION = 2
VDYN1. SIZE = 1000
ALLOC (VDYN1)
```

Les valeurs des spécifications de distribution, alignement et projection ne peuvent pas être manipulées en LSD2. En effet, dans le cas général, ce sont des listes de valeurs entières ou des mots clés (**DISTRIBUTION BLOC** par exemple). L'affectation indépendante des différents champs du vecteur autorise ainsi une plus grande souplesse dans l'allocation dynamique :

```
VECTOR REAL VDYN2
      SIZE              DYNAMIC, DYNAMIC
      DISTRIBUTION      DYNAMIC, DYNAMIC
END
```

```
IF (EXPR) THEN
      VDYN. SIZE = 1000, 2560
ELSE
      VDYN. SIZE = 2000, 2560
      VDYN. DISTRIBUTION = 2, -
ENDIF
ALLOC (VDYN)
```

La primitive **REALLOC** assure, en plus de l'allocation, la conservation des anciennes valeurs des éléments du vecteur dans les premiers éléments de la nouvelle zone. Une telle opération peut nécessiter une recopie de l'ancienne zone ou même, sur une machine massivement parallèle et pour des vecteurs distribués de manière explicite, des communications entre les processeurs pour amener les anciennes valeurs à leur nouvel emplacement mémoire.

La désallocation des vecteurs de taille dynamique peut être réalisée par la primitive **FREE**. Cependant, la vie d'un vecteur LSD2 est déterminée à la compilation et il n'existe pas en LSD2 de moyen pour un objet de partager les éléments d'un vecteur. LSD2 assure donc une désallocation automatique des vecteurs en fin de de vie de ceux-ci. Un vecteur alloué dans une fonction est désalloué à la fin de la fonction, un vecteur alloué dans un bloc est désalloué à la fin du bloc. De plus, toute allocation dynamique inclut une désallocation automatique de l'ancienne zone d'allocation du vecteur.

3–3.1.6 Typage des spécifications

Les spécifications de distribution, d'alignement et de projection correspondent à des types d'objets en LSD2. De tels objets ne peuvent pas être manipulés explicitement par le programmeur car il n'existe pas d'opérateur sur ces objets. Ces objets sont utilisés au sein des déclarations de vecteurs et lors de l'affectation des champs pour l'allocation d'un vecteur dynamique (cf. ci-dessus).

Cependant, les valeurs des différents champs composant un vecteur sont accessibles au programmeur. Par exemple,

```
V. DISTRIBUTION
```

retourne la valeur de la distribution du vecteur *V*. On peut l'utiliser pour la déclaration d'un vecteur *W* ayant la même distribution (il doit avoir le même nombre de dimensions) :

```
VECTOR INTEGER W
      SIZE              1000, 300
      DISTRIBUTION      V. DISTRIBUTION
END
```

Il est aussi possible de passer ces valeurs en paramètre, et ainsi déclarer des vecteurs locaux de même distribution que tel ou tel vecteur. Par exemple

```
SUBROUTINE SBRTN (DIST)
VECTOR INTEGER LOCAL
      SIZE              1000, 300
      DISTRIBUTION      DIST
END
```

3–3.1.7 Type vecteur

La déclaration d'un type de vecteur LSD2 associe à un nom des informations nécessaires à la déclaration d'un vecteur LSD2. De tels objets sont, par exemple, utiles pour assurer la concordance de type lors de passage de vecteurs en paramètre.

```
VECTOR TYPE PAIR_IMPAIR1
      SIZE        1000
      FILTER      IMPAIR      [1 : 1000 : 2]
      FILTER      PAIR        [2 : 1000 : 2]
      NODEPENDENCE IMPAIR, PAIR
END
```

A la suite de quoi, on déclare le vecteur *PAIM1*

```
VECTOR PAIR_IMPAIR1 REAL PAIM1
```

Une déclaration de vecteur peut aussi raffiner les spécifications de dépendances et les filtres d'un type vecteur préalablement défini. Par exemple,

```
VECTOR PAIR_IMPAIR1 REAL PAIM2
      FILTER      FIRST500    [1 : 500]
      FILTER      LAST500     [501 : 1000]
      NODEPENDENCE      FIRST500, LAST500
END
```

Le vecteur *PAIM2* est composé des deux filtres définis dans le type *PAIR_IMPAIR1* et des filtres *FIRST500* et *LAST500*. Cette fonctionnalité est utile par exemple pour la déclaration locale d'un filtre dans une fonction ou un bloc d'instructions.

Un type vecteur peut être paramétré. Les informations de taille, les filtres et les spécifications peuvent alors dépendre de la valeur de ces paramètres; leur valeur est obtenue par la valeur du paramètre lors de la déclaration du vecteur. Par exemple

```
VECTOR TYPE PAIR_IMPAIR2 (TAILLE)
      SIZE        TAILLE
      FILTER      IMPAIR      [1 : TAILLE : 2]
      FILTER      PAIR        [2 : TAILLE : 2]
      NODEPENDENCE IMPAIR, PAIR
END
```

La déclaration de vecteur spécifie la valeur du paramètre :

```
INTEGER N
VECTOR PAIR_IMPAIR2 (1000) REAL PAIM3
VECTOR PAIR_IMPAIR2 (N) REAL PAIM4
```

Le vecteur *PAIM3* est statique, le vecteur *PAIM4* est automatique.

3–3.2 Les filtres de vecteur

Les opérations de descriptions, accès aux éléments d'un vecteur, sont obtenues, en LSD2, par l'utilisation de filtres de vecteurs. Ces filtres de vecteurs sont définis au sein des objets vecteurs, lors de la déclaration de ceux-ci.

Les descriptions retenues par LSD2 sont les trois descriptions, maintenant habituelles, par une liste de valeurs booléennes, par une liste d'index et par un triplet *borne inférieure*, *borne supérieure* et *pas*, représenté par trois valeurs scalaires. La spécification de ces valeurs est optionnelle, les valeurs par défaut sont 1 pour la borne inférieure et le pas, la taille de la dimension pour la borne supérieure. Par exemple

```
VECTOR REAL DIM2
      SIZE        10, 20
      FILTER      F1          [ 1:10:1, 3:5:1]
      FILTER      F2          [ :, 3:5]
END
```

Le filtre *F2* désigne les mêmes éléments que le filtre *F1*.

Un filtre comporte autant de descriptions que le vecteur de dimensions. Pour chaque dimension d'un filtre, les valeurs de description sont obtenues par des tableaux du langage hôte ou des vecteurs LSD2 mono-dimensionnels et de type entier ou booléen. Les opérateurs de décalage (`shift`) et de rotation (`rotate`) peuvent s'appliquer à ces valeurs de description. Par exemple :

```
VECTOR REAL DIM2
      SIZE        10, 20
      FILTER      F1          [ :, 3:5]
      FILTER      F3          F1 [ :, : shift 2]
END
```

LSD2 autorise aussi la description par une valeur scalaire. Une telle description change la forme du vecteur. Le filtre possède une dimension de moins que le vecteur.

LSD2 intègre dans les filtres d'autres constructions autorisant, elles aussi, le changement de la forme des vecteurs. Deux opérations sont distinguées :

- les descriptions de diffusion, et
- les descriptions de réduction.

Nous basons l'explication de ces deux types de descriptions sur des vecteurs, nommés *espace*. Les vecteurs de taille inférieure sont nommés *plan*. Ceci ne restreint en rien l'application de ces descriptions à des vecteurs tri-dimensionnels, mais introduit une terminologie pour simplifier l'exposé.

Un filtre d'un *espace* comprenant une diffusion (notée de la flèche ->) est une référence vers tous les *plans* de cet *espace*. Un tel filtre doit apparaître en partie gauche d'une affectation; il diffuse la valeur de la partie droite à tous ces *plans*. Considérons le vecteur *DIM2* déclaré

```
VECTOR REAL DIM2
      SIZE        5, 3
      FILTER      COLONNE1 (I)      [:, I]
      FILTER      COLONNES          [ :, ->]
      FILTER      LIGNES            [ ->, 2:3]
END

DIM2. COLONNES = [1:3]
DIM2. LIGNES = [1:2]
```

La première instruction, affecte toutes les colonnes de *DIM2* de la même valeur; les éléments de la ligne *i* ont pour valeur *i*. Cette première instruction produit le même résultat que les instructions suivantes (pas forcément le même code) :

```
      DO 10 I = 1, 3
         DIM2. COLONNE1 (I) = [1:3]
10 CONTINUE
```

La seconde instruction affecte les lignes 2 et 3 de *DIM2*. Les éléments de la ligne 2 ont pour valeur 1, ceux de la ligne 3 ont pour valeur 2.

A l'inverse de la diffusion, la présence d'une réduction dans un filtre d'*espace* réduit cet *espace* à un *plan* par composition de tous ses *plans*. Une opération (commutative et associative) est donc associée à la réduction. L'ordre de combinaison des *plans* n'est pas précisé. Le résultat d'une telle description est une valeur; un tel filtre doit apparaître en partie droite d'une affectation. Par exemple

```
REAL P
VECTOR REAL DIM3
      SIZE        5, 20, 3
      FILTER      SOMME       [ :, +, :]
      FILTER      PRODUIT     [ * : * : * ]
END

DIM2 = DIM3. SOMME
P = DIM3. PRODUIT
```

Le filtre *SOMME* est un vecteur deux dimensions dont les valeurs sont la somme des plans de *DIM3* selon son second axe. Le filtre *PRODUIT* retourne le produit de tous les éléments du vecteur.

De manière évidente, une description de diffusion est une *l-value*, elle ne peut qu'être affectée. Une description de réduction est une *r-value*, elle référence des valeurs. Un même filtre ne peut donc pas contenir une description de diffusion et une description de réduction.

3–3.3 Les spécifications de distribution, d'alignement et de projection

La manipulation de vecteurs sur une machine massivement parallèle est largement pénalisée par les communications entre les processeurs. Clairement, la recherche de performances sur ces machines est fortement liée à la recherche de schémas de distribution ("distribution" est ici utilisé dans le sens général de répartition) des éléments de vecteurs sur les différents processeurs afin de minimiser les communications entre les processeurs lors de la mise en place de l'algorithme. Cependant, il ne nous paraît pas acceptable que cette distribution des éléments se fasse sans le respect de certaines régles.

- Le coût des fonctions de calcul des adresses des éléments doivent rester "raisonnables". L'objectif est avant tout l'obtention de performances, rien ne sert de gagner d'un coté ce que l'on perd de l'autre. En particulier, il est indispensable que ces fonctions de calcul restent SIMD.
- Le codage de cette distribution doit rester "simple". En effet, dans notre approche, les informations de distribution associées à un vecteur sont passées en paramètre. Il est donc nécessaire qu'une représentation peu coûteuse existe pour ces informations.

- L'expression de cette distribution doit rester indépendante de la machine cible. En particulier, on ne peut supposer a priori connu le nombre de processeurs ou l'architecture du réseau d'interconnexions entre les processeurs.

Nous avons mis en place trois spécifications de distribution, d'alignement et de projection. A travers ces spécifications, le programmeur indique des liens forts entre des éléments ou des groupes d'éléments de vecteurs ou des ordres de priorité entre les diverses dimensions parallèles de ces données. La prise en compte de ces informations est assurée par le compilateur lors de l'allocation des vecteurs sur la mémoire des processeurs et ensuite lors du traitement de ces vecteurs.

La présentation de ces spécifications est délibérément axée sur l'implantation sur machines massivement parallèles. Cependant, la prise ne compte de ces informations pour les machines pipelines vectorielles n'est pas à exclure. Ces spécifications pourraient être utilisées pour allouer les vecteurs de manière à réduire les conflits d'accès mémoire. Sur une machine pipeline vectorielle multi-processeurs, la distribution pourrait indiquer une répartition des données entre les différents processeurs vectoriels. Ces perspectives sont l'objet de travaux en cours.

Nous présentons ces trois spécifications dans les sous-sections suivantes.

3–3.3.1 Distribution

La distribution d'un vecteur est une spécification qui, pour chacune des dimensions de ce vecteur, indique les traitements qui seront privilégiés par l'allocation des éléments du vecteur dans la mémoire distribuée d'une machine massivement parallèle. La spécification de distribution résulte en la définition de fonctions d'allocation. L'allocation d'un vecteur détermine, pour chacun des processeurs, les éléments qu'il détient, ou pour chaque élément, le processeur qui le détient.

En LSD2, la distribution spécifie le rapprochement d'éléments en groupes. Une telle spécification indique des liens forts entre les éléments du groupe, les échanges entre ces éléments seront importants. Sur une machine massivement parallèle, l'allocation de ces groupes d'éléments sur un même processeur est donc avantageuse; elle évite certaines communications entre les processeurs.

Dans la suite de cette section, nous présentons le modèle d'allocation sur les processeurs des éléments de vecteur en l'absence de distribution (distribution cyclique) et discutons ensuite d'une alternative à cette distribution (distribution par blocs). Nous proposons ensuite des distributions intermédiaires entre les deux extrêmes précédents. Enfin nous présentons des distributions conditionnelles dont l'application dépend de caractéristiques de la machine sur laquelle est exécuté le programme. La figure 3–4 reprend, pour un exemple, les conséquences des différentes distributions.

Nous supposons le vecteur de taille N, les éléments sont indicés par n, valeur de 1 à N. La machine est constituée de P processeurs, indicés par p de 0 à P–1. Nous présentons ici l'allocation pour des vecteurs mono-dimension. Rappelons qu'en LSD2 cette distribution peut être réalisée indépendamment pour toutes les dimensions d'un vecteur multi-dimensionnel.

Distribution cyclique

En l'absence de spécification de distribution, LSD2 alloue les éléments de manière cyclique. L'élément de vecteur d'indice n est alloué sur le processeur

(n–1) modulo P.

(Notons que dans le cas où la valeur P, nombre de processeurs est une puissance de 2, l'opération de modulo se réduit à une manipulation simple des bits de la représentation binaire de la valeur n.)

Les principaux avantages d'une telle allocation sont les suivants :

- les opérations réalisées sur des tranches d'éléments du vecteur peuvent l'être en parallèle, de tels éléments étant alloués sur des processeurs distincts.
- le numéro du processeur détenant un élément de vecteur donné est indépendant de N, la taille du vecteur. Lors d'opération de ré-allocation du vecteur, aucune migration des anciennes valeurs des éléments n'est nécessaire.

A l'inverse, la réalisation d'une opération de communication entre voisins peut nécessiter un rebouclage. Par exemple, les éléments de vecteur d'indice $P+1$ et $P+2$ sont respectivement alloués sur le dernier et le premier processeur. Un tel rebouclage est coûteux s'il n'est pas pris en compte par l'architecture de la machine.

Distribution par blocs

L'alternative à la distribution cyclique est d'allouer les éléments sur les processeurs par blocs. L'élément de vecteur d'indice n est alloué sur le processeur indicé par la division entière

$(n-1) / B$.

B est la taille d'un bloc, sa valeur est

$((N-1)/P) + 1$.

Les avantages et inconvénients de la distribution par blocs sont contraires à ceux de la distribution cyclique :

- les opérations de communications de voisinage à l'intérieur d'un bloc sont réalisées sans même générer de communications inter-processeurs.
- plus généralement, les communications de voisinages ne nécessitent jamais de rebouclage.

Inversement, le numéro du processeur détenant un élément d'indice donné est dépendant de N, la longueur du vecteur. Cette valeur doit donc être connue pour tout accès à un élément du vecteur à partir de son index. De plus, la ré-allocation d'un tel vecteur entraînera des migrations de valeurs entre les processeurs.

Du point de vue de l'efficacité du programme, le choix de l'une ou l'autre de ces deux allocations est lié aux opérations réalisées sur les éléments du vecteur. Une opération de communications entre des éléments proches aura avantage d'une distribution par blocs. Une opération traitant un sous-ensemble d'éléments contigus dans un vecteur est à l'inverse pénalisée par une distribution par blocs; le parallélisme de l'opération étant limité si plusieurs éléments sont alloués dans un même processeur.

En LSD2, la distribution est par défaut cyclique. Le programmeur spécifie une distribution par bloc avec la spécification de distribution

```
DISTRIBUTION BLOC
```

Raffinement des distributions, la distribution spécifique

Entre les deux types de distribution, cyclique et par blocs, il existe des distributions intermédiaires pouvant combiner les avantages de chacune des deux approches. Nous détaillons ici la distribution spécifique et présentons son expression en LSD2.

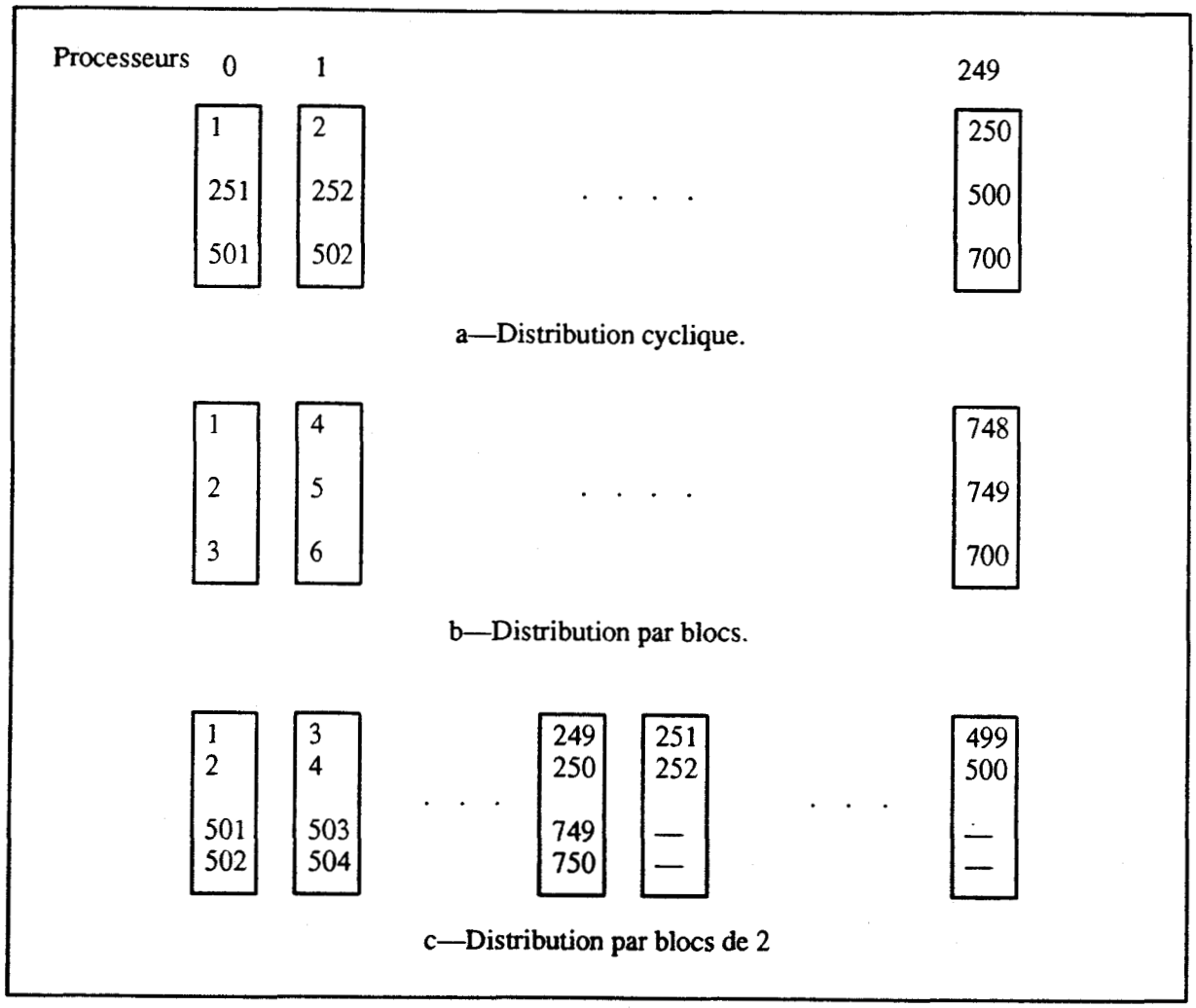


Figure 3–4. Distributions du vecteur *PAIM* sur 250 processeurs.

Nous introduisons cette distribution sur un exemple. Nous supposons une machine de P=250 processeurs. Considérons un vecteur LSD2 *PAIM* de N=750 éléments déclaré

```
VECTOR REAL PAIM
      SIZE        750
      FILTER      IMPAIR      [1 : 750 : 2]
      FILTER      PAIR        [2 : 750 : 2]
      NODEPENDENCE IMPAIR, PAIR
END
```

et l'opération faisant interagir les éléments d'indice impair et leur successeur

```
IMPAIR + PAIR
```

Par défaut, c'est-à-dire pour une distribution cyclique des éléments, un processeur donné contient les trois éléments du vecteur d'indice p+1, p+251, et p+501 (figure 3–4.a). La réalisation de l'opération nécessite donc une communication entre les couples de processeurs p=2*n et 2*n+1, pour n compris entre 0 et P/2–1=124.

Considérons maintenant une distribution par blocs.

```
VECTOR REAL PAIM
      SIZE        750
      FILTER      IMPAIR      [1 : 750 : 2]
      FILTER      PAIR        [2 : 750 : 2]
      NODEPENDENCE IMPAIR, PAIR
      DISTRIBUTION BLOC
END
```

Les blocs seront de taille 3. Le processeur p détient les éléments d'indice 3*p+1, 3*p+2, et 3*p+3 (figure 3–4.b). La réalisation de l'opération oblige une communication entre les couples de processeurs adjacents détenant, pour l'un, un élément d'indice pair, et pour l'autre, l'élément d'indice impair suivant, ou inversement.

Pourtant, il existe des allocations de l'ensemble des éléments de *PAIM* sur les processeurs de la machine telles qu'aucune communication ne soit nécessaire pour la réalisation de l'opération. Une telle allocation est par exemple spécifiée en LSD2 par une distribution de valeur 2 :

```
VECTOR REAL PAIM
      SIZE        750
      FILTER      IMPAIR      [1 : 750 : 2]
      FILTER      PAIR        [2 : 750 : 2]
      NODEPENDENCE IMPAIR, PAIR
      DISTRIBUTION     2
END
```

Une telle distribution est basée sur le regroupement des éléments du vecteur en blocs de taille 2. Ces blocs sont ensuite distribués de manière cyclique sur les processeurs (figure 3–4.c). Dans notre cas, la première moitié des processeurs détient deux blocs de deux éléments, la seconde moitié des processeurs détient un bloc de deux éléments.

Dans le cas général, une distribution spécifique est avantageuse pour les raisons suivantes :

- Les communications au sein des blocs ne produisent pas de communication entre les processeurs. La taille des blocs étant définie par le programmeur, un contrôle fin des communications est possible.
- Pour n donné, l'accès à l'élément n est indépendant de la taille N du vecteur.

Cependant, une telle distribution est aussi la cause de dégradations éventuelles :

- perte de place mémoire, et
- perte de parallélisme.

Dans notre exemple, l'allocation du vecteur *PAIM* selon la distribution de 2 nécessite 4 emplacements mémoire par processeurs. Les allocations cycliques et par blocs se suffisaient de 3 emplacements. Très liée à cette perte de mémoire, la perte de parallélisme est évidente lors d'opérations agissant sur toutes les valeurs du vecteur. Sur notre exemple, l'incrémentation des valeurs du vecteur, par exemple, nécessite quatre opérations par processeur pour une allocation spécifique de 2, elle n'en nécessite que 3 dans les deux cas traditionnels.

Par ailleurs, la distribution spécifique n'évite pas systématiquement le rebouclage des communications au voisins. Nous proposons de remédier à ce type de situations par une expression plus fine de la distribution. Ces distributions spécifiques sont qualifiées de conditionnelles.

Distributions spécifiques conditionnelles

Une distribution spécifique conditionnelle est caractérisée par une suite de distributions. Ces distributions ne sont réalisées que sous certaines conditions. Une distribution n'est réalisée que si les distributions antérieures produisent un rebouclage des communications locales. Par exemple, considérons la distribution

```
DISTRIBUTION 2 (3)
```

Elle indique une distribution de 2 impérative et une distribution conditionnelle de 3. Les éléments seront donc distribués par blocs de 2. Ces blocs de deux éléments seront conditionnellement distribués par blocs de 3. Cette distribution conditionnelle n'est effective que si la distribution antérieure induit un rebouclage des communications locales. Si l'on considère notre exemple du vecteur *PAIM*, cette seconde distribution aura bien lieu, le vecteur sera donc distribué par blocs de 6 éléments (de manière cyclique).

Plusieurs distributions conditionnelles peuvent être exprimées. La dernière distribution conditionnelle peut être une spécification `BLOC`. Les ensembles d'éléments obtenus par les distributions antérieures seront alors distribués par blocs (si nécessaire).

Les avantages liés à cette distribution sont le contrôle fin des communication locales, la possible absence de rebouclage. On notera aussi la conservation d'un parallélisme élevé, sur une machine comportant "beaucoup" de processeurs, par rapport à une distribution spécifique de taille plus importante.

La décision de distribution dépend des caractéristiques de la machine (en particulier de P, nombre de processeurs). Ces caractéristiques sont cachées du programmeur, cependant leur prise en compte est possible via l'aspect conditionnel de la distribution.

Notons qu'une fois les caractéristiques de la machine établies, le codage d'une telle distribution se fait sur un entier (la taille des blocs à distribuer de manière cyclique); l'aspect conditionnel n'a plus à être conservé. Dans notre environnement, la prise en compte de la distribution est conditionnelle jusqu'à son expression dans le langage intermédiaire LIMP. L'aspect conditionnel est consommé lors de la compilation de LIMP sur une machine donnée.

3–3.3.2 Alignement

La distribution est une caractéristique attachée à un vecteur. A l'inverse, l'alignement est une spécification qui autorise la prise en compte d'interactions entre les éléments de deux vecteurs.

Pour un vecteur, l'alignement exprime un déplacement de l'adresse de base dans un repère absolu et unique. Cette caractéristique d'expression par rapport à un repère absolu et unique autorise l'encapsulation de la valeur dans le vecteur.

L'interaction entre deux vecteurs est donc exprimée par un placement des vecteurs dans ce repère. Par exemple, considérons les vecteurs *PAIM* et *UPPER* déclarés comme suit

```
VECTOR REAL PAIM
      SIZE N
      FILTER      SUP          [N/2 : N]
      FILTER      IMPAIR       [1 : N : 2]
      FILTER      PAIR         [2 : N : 2]
      NODEPENDENCE IMPAIR, PAIR
      DISTRIBUTION 2
END

VECTOR REAL UPPER
      SIZE N/2
END
```

Supposons une instruction d'affectation des valeurs des derniers éléments du vecteur *PAIM* dans le vecteur *UPPER* :

```
UPPER = PAIM. SUP
```

Sur une machine massivement parallèle, en supposant une allocation "canonique" de *UPPER*, l'affectation de chaque élément demande une communication entre le processeur détenant la valeur de l'élément de *PAIM. SUP* et le processeur détenant l'élément dans lequel ranger cette valeur (figure 3–5).

L'expression d'un alignement et d'une distribution spécifiques pour le vecteur *UPPER* peut éliminer cette communication. Pour éviter toute communication, les éléments de *UPPER* doivent être alloués dans le même processeur que leur élément correspondant dans le vecteur *PAIM*. Pour ce faire, le vecteur *UPPER* est déclaré

```
VECTOR REAL UPPER
      SIZE              N/2
      DISTRIBUTION      2
      ALIGNEMENT        N/4
END
```

La même distribution que *PAIM* est appliquée. Le vecteur *UPPER* est donc composé de *N*/4 blocs de 2 éléments. La spécification d'alignement décale les paires d'éléments de *N*/4 adresses dans l'espace d'adressage virtuel référencé par le repère absolu. La réalisation de l'affectation à *PAIM* est réalisée sans communication (figure 3–6).

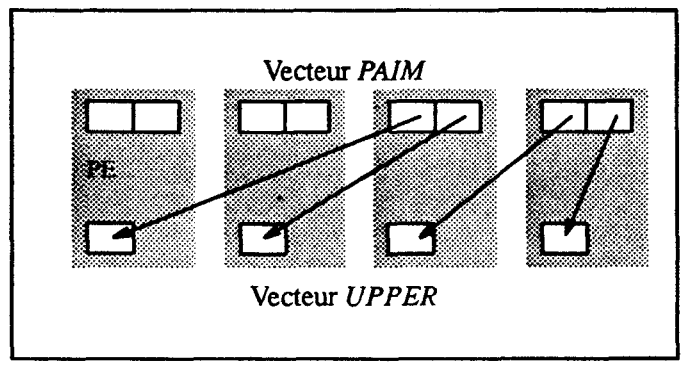


Figure 3–5. Communications inter-processeurs pour l'allocation canonique de *UPPER*.

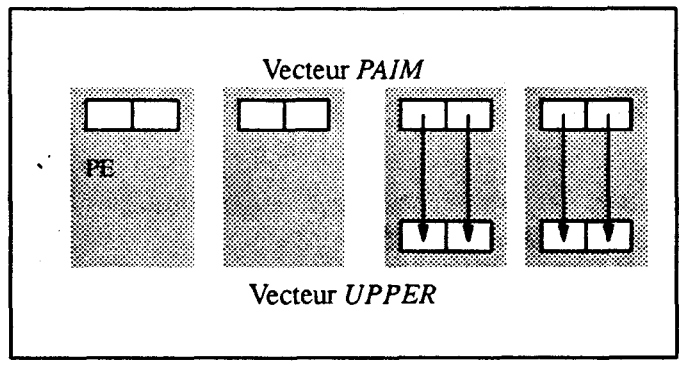


Figure 3–6. Conséquence de l'alignement et de la distribution des éléments de *UPPER*.

3–3.3.3 Projection

Les vecteurs LSD2 sont multi-dimensionnels. De plus, les éléments de chacune des dimensions d'un vecteur multi-dimensionnel peuvent être manipulés en parallèle. LSD2 ne limite pas le nombre de dimensions d'un vecteur multi-dimensionnels.

De manière évidente, le nombre de dimensions parallèles des machines est limité. Les machines vectorielles pipelines ne traitent que des vecteurs à une dimension parallèle. Le réseau d'interconnexions entre les processeurs des machines massivement parallèles est souvent une hyper-grille dont la dimension est restreinte.

La projection d'un nombre de dimensions parallèles d'un vecteur sur une machine possédant un nombre inférieur de dimensions parallèles n'est certainement pas unique. La méthode la plus simple est de réduire le nombre de dimensions parallèles du vecteur au nombre de dimensions parallèles de la machine par une projection non-parallèle (c'est-à-dire en mémoire) des dimensions supplémentaires.

Le choix des dimensions à projeter ainsi ne peut être fait sans la connaissance des caractéristiques de la machine. Le programmeur peut tout au plus indiquer une préférence sur les dimensions du vecteur qu'il

désire prioritairement traiter en parallèle. Une telle préférence peut être exprimée par une spécification de projection au sein de la déclaration d'un vecteur. Par exemple, dans la déclaration du vecteur trois dimensions *PROJ3*

```
VECTOR REAL PROJ3
      SIZE        100, N, M
      PROJECTION  3, 1, 2
END
```

la spécification de projection exprime que les dimensions à traiter en parallèle sont, par ordre de priorité décroissante, les dimensions 3, 1, et 2. Sur une machine vectorielle pipeline, une instruction d'incrémentation de *PROJ3*

```
PROJ3 [INDEX, 1:N:2, 1:M:2] = 1
```

s'écrit alors

```
DO J = 1, N, 2
      DO I = 1, 100
            PROJ3 [INDEX [I], J, 1:M:2] = 1
      ENDDO
ENDDO
```

De même, sur une machine massivement parallèle dont le réseau d'interconnexion est une grille de dimension 2, les dimensions 3 et 1 du vecteur *PROJ3* sont allouées sur les dimensions parallèles de la machine. La dimension 2 est allouée dans la mémoire des processeurs. Aucun parallélisme ne pourra donc être obtenu dans les traitements suivant cette seconde dimension.

3–3.4 Les spécifications de dépendances

D'autres spécifications incluses dans les vecteurs LSD2 expriment les dépendances et non-dépendances entre les filtres d'accès aux éléments de vecteurs. Nous distinguons plusieurs types de dépendances entre deux filtres d'un vecteur. Nous allons les présenter dans cette section.

La spécification de ces dépendances détermine la possibilité d'optimisation au niveau de la génération de code. Deux aspects sont concernés : le passage ou non par un temporaire lors d'une affectation vectorielle et la validité de la fusion des boucles de strip-mining de deux instructions successives. (Se référer à la dernière section du premier chapitre de ce document.)

Nous définissons une dépendance entre deux filtres d'un même vecteur comme le partage d'éléments du vecteur entre les filtres. Une dépendance est donc caractérisée par deux indices i et j dans les deux filtres tels que l'élément i du premier filtre et l'élément j du second filtre référencent le même élément du vecteur. A chaque dépendance est associé un sens de la dépendance. Le sens de la dépendances est le signe de la différence j–i.

Nous présentons ici la position de LSD2 quant aux dépendances entre deux vecteurs. Les sous-sections suivantes sont une première présentation des dépendances caractérisées en LSD2 entre les filtres d'un vecteur. Cette présentation sera complétée dans la suite de ce chapitre par un exposé plus complet de ces dépendances et de la génération de code en découlant.

3–3.4.1 Dépendances entre les vecteurs

Préalablement, insistons sur le fait qu'en dehors de deux filtres d'un même vecteur, il n'y a, a priori, aucune dépendance : une dépendance exprime le fait que deux objets accèdent à des éléments de vecteurs communs en mémoire; un tel partage ne peut être réalisé en LSD2 qu'au travers des filtres d'un même vecteur.

Cependant, certaines constructions peuvent amener un tel partage entre deux vecteurs. Le passage de vecteurs en paramètre peut amener un vecteur paramètre à être semblable à un autre vecteur paramètre par exemple. Les instructions FORTRAN **EQUIVALENCE** et **COMMON** ou les pointeurs du langage C peuvent aussi être la source de dépendances entre deux vecteurs différents. Ces cas sont considérés comme extrêmes, nous supposons donc, a priori, qu'il n'y a pas de dépendance entre deux objets référencés par deux noms différents qui ne sont pas deux filtres d'un même vecteur. Toutefois, une telle dépendance pouvant surgir dans un programme LSD2, il est possible de l'exprimer par une déclaration explicite de dépendance :

```
DEPENDENCE VECTNAME VECTNOM
```

qui indique un éventuel partage d'éléments entre les deux vecteurs *VECTNAME* et *VECTNOM*. Il existe une spécification unique de dépendance à ce niveau. Elle correspond à une spécification de dépendance totale (cf. infra).

3–3.4.2 Dépendances totales

En l'absence de spécification de dépendance et non-dépendance, les filtres d'un vecteur sont considérés dépendants deux à deux. Une telle dépendance est qualifiée de *totale*. Deux filtres totalement dépendants sont caractérisés par un partage d'éléments du vecteur sans que rien de particulier n'identifie ce partage.

La prise en compte, au niveau de la génération de code, de la dépendance totale entre deux filtres est double. Une dépendance totale entre deux filtres parties droite et gauche d'une affectation oblige le passage par un temporaire. Une dépendance totale entre deux filtres de deux instructions successives dont l'un au moins est en partie gauche empêche la fusion des boucles.

Une spécification de non dépendance entre deux filtres caractérise deux filtres dont l'intersection des éléments du vecteur est vide. Aucun des éléments du premier filtre n'apparaît dans le second, et inversement. Une telle caractéristique est exprimée, en LSD2, par une spécification **NODEPENDENCE**. Par exemple

```
VECTOR REAL PAIM
      SIZE N
      FILTER      IMPAIR      [1 : N : 2]
      FILTER      PAIR        [2 : N : 2]
      NODEPENDENCE IMPAIR, PAIR
END
```

En effet les deux filtres *PAIR* et *IMPAIR* ne référencent aucun élément commun dans le vecteur.

Cette caractéristique d'intersection vide autorise le codage en une boucle unique et sans passage par un temporaire d'une affectation dont l'un des filtres est partie droite et l'autre appartient à la partie gauche. De

même, elle autorise la fusion de deux instructions successives référençant ces deux filtres, même lorsque ces références sont en partie gauche d'affectations.

3–3.4.3 Dépendances partielles

Nous avons caractérisé des dépendances dites partielles pour lesquelles la qualité du code généré est amélioré par rapport à une dépendance totale. Les conséquences de ces dépendances partielles sur la génération de code se situent entre celles des deux extrêmes de dépendance totale ou de non-dépendance. Nous présentons ces dépendances, leurs expressions en LSD2 et donnons une idée de la génération de code depuis des instructions dont les opérandes sont liés par de telles dépendances. Cette génération de code est détaillée dans la suite du document.

Semi-dépendance

Une semi-dépendance est une dépendance entre deux filtres d'un vecteur caractérisée par un sens constant de toutes les dépendances entre éléments. C'est-à-dire que pour tout couple d'éléments des deux filtres (respectivement d'indice *i* dans le premier filtre, et d'indice *j* dans le second filtre) tels que les deux éléments référencent le même élément du vecteur, la différence entre *i* et *j* est de signe constant. Considérons, par exemple, la déclaration LSD2 du type vecteur

```
VECTOR TYPE PAIR_IMPAIR_TROIS
      SIZE        1000
      FILTER      IMPAIR      [1 : 1000 : 2]
      FILTER      TROIS       [1 : 1000 : 3]
      DEPENDENCE ALL < IMPAIR
      DEPENDENCE IMPAIR < TROIS
END
```

La figure 3–7 représente les filtres *IMPAIR* et *ALL*. Un point matérialisé d'une courbe donne l'indice dans le vecteur de l'élément *i* du filtre tel que *i* est l'abscisse de ce point. La dépendance partielle entre *ALL* et *IMPAIR* est représentée sur le schéma par le fait que pour tout point de *ALL*, il n'existe pas de point de *IMPAIR* de même ordonnée et d'abscisse supérieure.

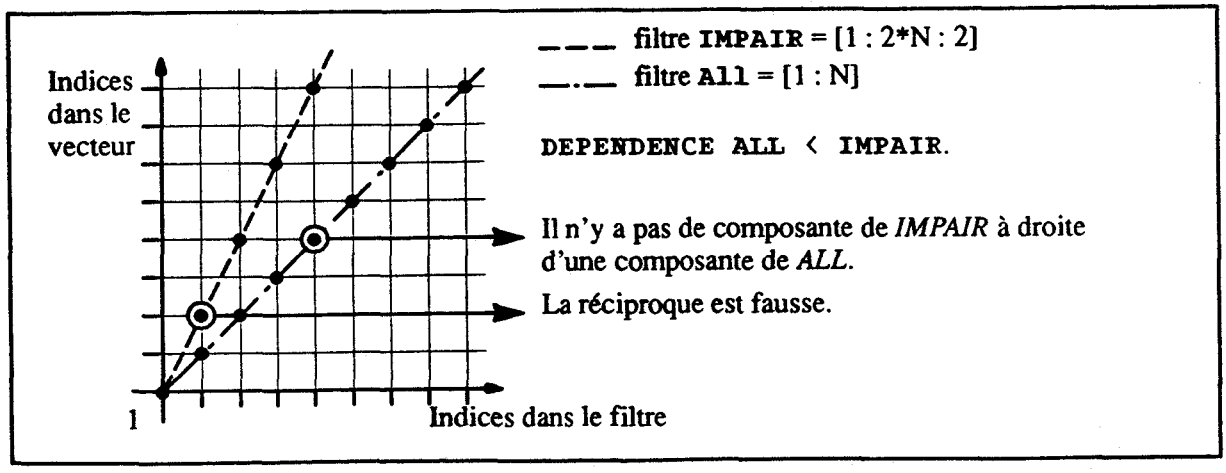


Figure 3–7. Exemple de semi-dépendance.

De même, le filtre *IMPAIR* est semi-dépendant du filtre *TROIS*. En effet, les éléments de vecteur d'indice $6*n+1$, n entier, sont référencés par les deux filtres *IMPAIR* et *TROIS*. Mais, pour un indice donné dans les filtres, l'élément dans le vecteur référencé par le filtre *IMPAIR* est toujours d'indice (dans le vecteur) inférieur à celui de l'élément référencé par l'élément correspondant du filtre *TROIS*.

Nous introduisons la génération de code à partir d'une telle spécification sur un exemple d'instruction. Considérons l'instruction

```
IMPAIR = TROIS
```

Cette instruction peut être générée en une seule boucle, sans utilisation de zone temporaire. En effet lors d'une itération de la boucle, les éléments du vecteur écrits par le filtre *IMPAIR* ne seront plus référencés lors des itérations de boucles suivantes, par définition de la semi-dépendance, l'indice dans *IMPAIR* d'un élément de vecteur commun étant inférieur. De manière duale, les éléments de vecteur lus par le filtre *TROIS* lors d'une itération n'ont pas été écrits par les itérations précédentes. Le cas général de génération de code depuis une telle dépendance est étudié dans la suite de ce chapitre.

Dépendance par blocs

D'autre dépendances partielles ont été caractérisées entre deux filtres d'un vecteur. Il s'agit des dépendances par blocs. Nous présentons une première forme réduite de ces dépendances. Nous introduisons ensuite une paramétrisation supplémentaire de ces dépendances.

Une dépendance par blocs entre deux filtres est caractérisée par un découpage en blocs des indices des filtres tels que deux éléments de filtres de deux blocs différents ne soient pas dépendants. Une spécification précise la taille du bloc.

Considérons l'exemple d'une permutation des lignes d'une matrice. Soit *MATRICE* une matrice de 128 lignes par 65536 colonnes. Comme c'est le cas en FORTRAN, nous supposons que les tableaux multi-dimensionnels sont alloués colonne par colonne en mémoire. Considérons l'instruction de permutation de lignes (figure 3–8) :

```
INTEGER INDEX (128)
VECTOR REAL MATRICE
      SIZE        128, 65536
      FILTER      GATHER (I)        [I, :]
      DEPENDENCE BLOC (128) ALL < GATHER
END

MATRIX = MATRIX. GATHER (INDEX)
```

Nous supposons que le tableau *INDEX* contient une permutation des entiers compris entre 1 et 128. Les deux filtres *ALL* (filtre prédéfini référençant la matrice toute entière) et *GATHER* (la matrice dont les lignes sont permutées) sont liés par une dépendance par blocs de facteur de blocage 128 tel que exprimée par la spécification de dépendance.

Cette spécification exprime qu'un élément du vecteur référencé par le filtre *ALL* dans un bloc de 128 éléments (en fait une colonne de la matrice) ne sera plus référencé par le filtre *GATHER* après le bloc (figure 3–9). (De même, la réciproque est vraie, c'est un cas particulier, la relation de dépendance par blocs n'est en général pas commutative.)

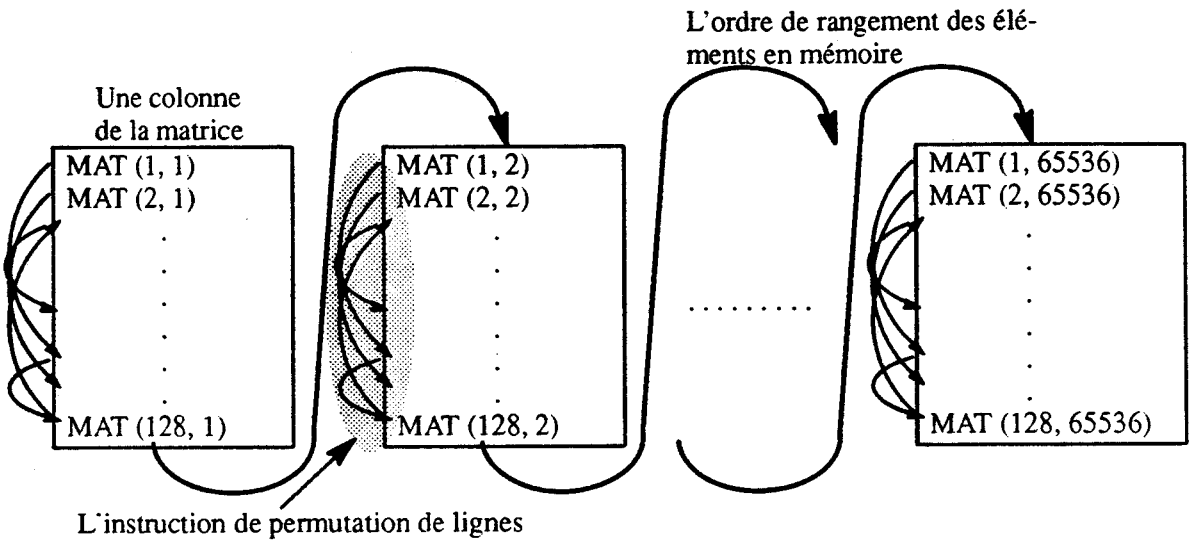


Figure 3–8. Rangement mémoire des éléments de la matrice et instruction de permutation de lignes.

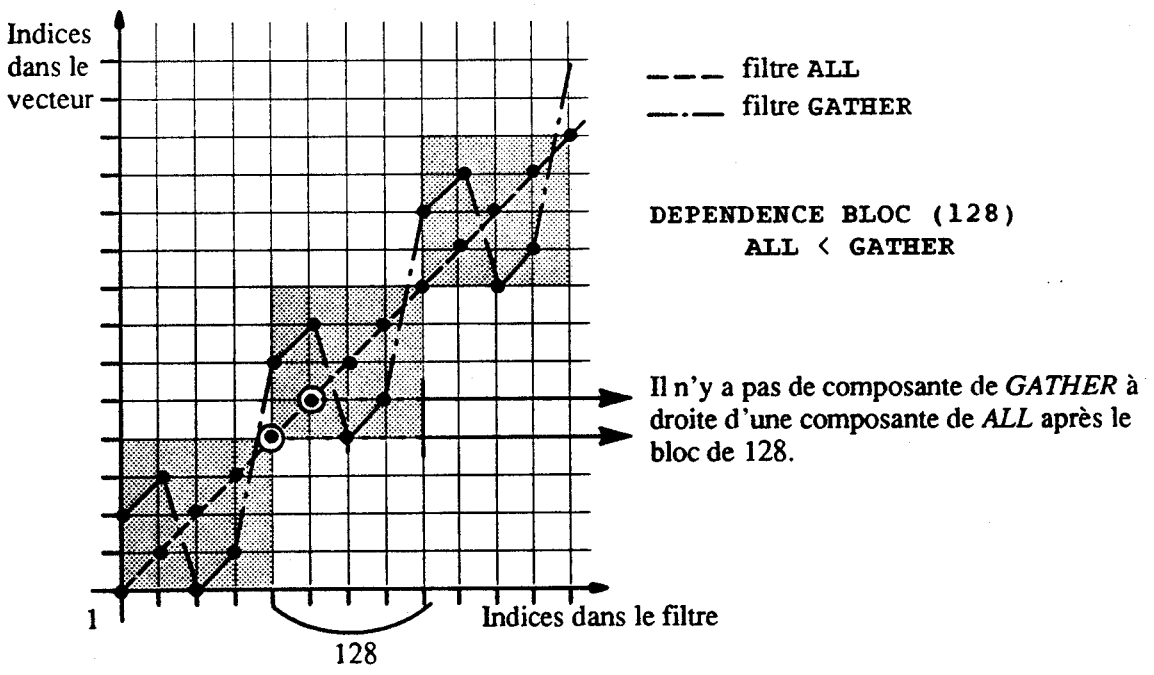


Figure 3–9. Dépendance par blocs pour l'instruction de permutation de lignes.

La génération de code pour cette instruction est basée sur le traitement de colonnes de la matrice. Le rangement dans une colonne, via le filtre *ALL*, peut avoir lieu une fois que tous les éléments de la colonne ont été lu via le filtre *GATHER*. Il est inutile de charger la matrice dans une zone temporaire avant de réécrire cette zone temporaire dans la matrice, la mémorisation d'une colonne suffit. Cette mémorisation peut être réalisée dans quelques registres par exemple.

Nous introduisons maintenant un nouveau paramètre à une telle dépendance par bloc, il s'agit du facteur de groupe.

Considérons la nouvelle définition de la matrice :

```
INTEGER INDEX (128)
VECTOR REAL MATRICE
      SIZE        128, 65536
      FILTER      TRUNC              [:, 1:65535]
      FILTER      GATHER (I)         [I, 2:65536]
      DEPENDENCE BLOC (128, 2) GATHER < TRUNC
END

MATRIX. GATHER (INDEX) = MATRIX. TRUNC
```

La permutation des lignes est associée à un décalage des colonnes. La spécification de dépendance indique qu'en considérant des blocs d'éléments de taille 128, un élément du vecteur référencé par le filtre *GATHER* dans un bloc, ne sera plus référencé par le filtre *TRUNC* à partir du 2me bloc suivant (figure 3–10).

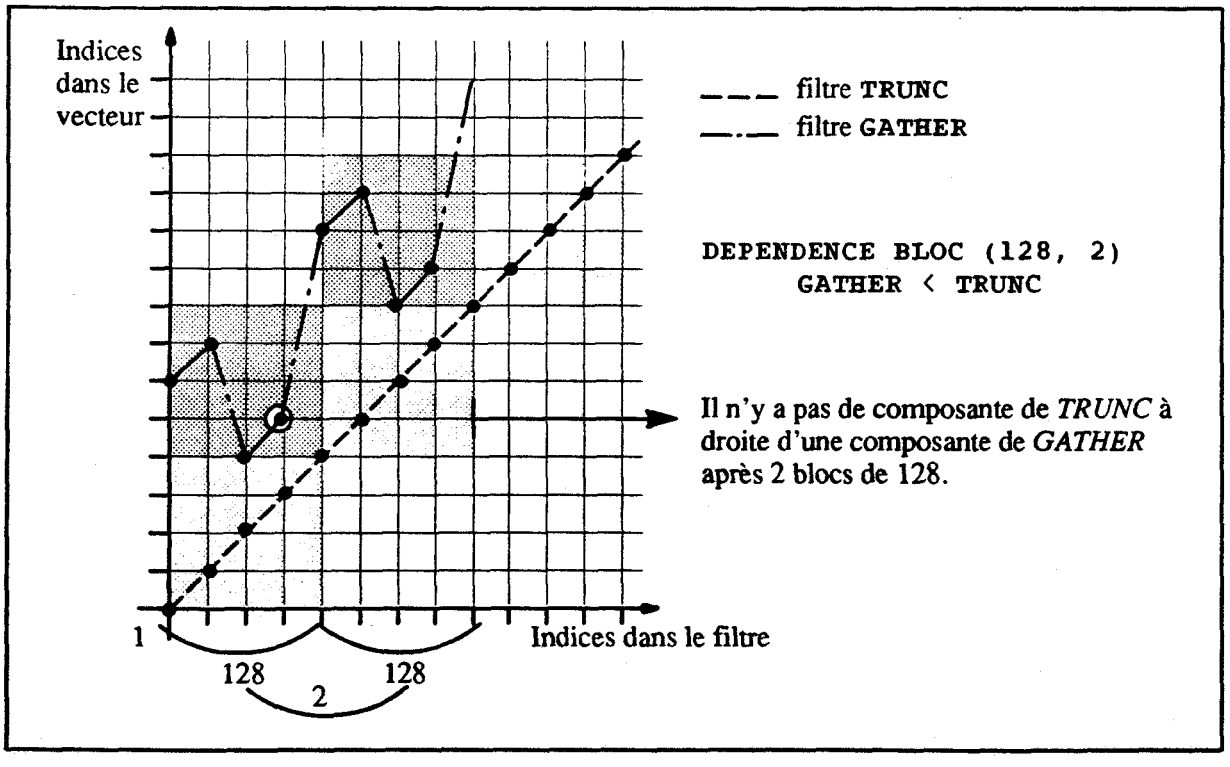


Figure 3–10. Dépendance par blocs avec facteur de groupe.

La génération de code est basée sur le modèle précédent, toutefois il est nécessaire d'avoir chargé deux colonnes (la colonne courante et la suivante) depuis la mémoire avant de pouvoir écrire le nouvelles valeurs des éléments de cette colonne courante.

3–3.5 Manipulation de vecteurs en LSD2

Nous avons présenté les vecteurs LSD2 dans les sections précédentes. Nous précisons ici comment ces vecteurs sont manipulés au sein de LSD2. En particulier, nous présentons l'accès aux éléments de vecteurs via les filtres, les opérations appliquées aux filtres et les vecteurs temporaires qu'elles peuvent produire, ainsi que les instructions manipulant les vecteurs et les filtres.

3–3.5.1 Accès aux éléments de vecteurs

Les accès aux éléments de vecteurs sont réalisés via les filtres en LSD2. Le nom d'un filtre référence les éléments du vecteur correspondant. Cette référence accède à la valeur des éléments ou modifie ces valeurs selon que le filtre est en partie droite ou gauche d'une affectation. Cependant, les filtres construits à l'aide d'une diffusion ne peuvent qu'apparaître en partie gauche d'une affectation. Inversement, un filtre contenant une réduction doit apparaître en partie droite.

La référence se fait par le nom du filtre si cela n'est pas ambigu; elle est préfixée du nom du vecteur dans le cas général. Un filtre de nom *ALL* est prédéfini pour chaque vecteur, il référence tous les éléments du vecteur. En l'absence de suffixe de filtre, le nom d'un vecteur référence ce filtre *ALL*.

3–3.5.2 Opérations sur les vecteurs et les filtres

Les opérations sur les vecteurs et les filtres LSD2 sont essentiellement les opérations du langage hôte dont nous étendons la fonctionnalité aux filtres. Les opérandes vecteurs doivent être de forme (nombre de dimensions) identique. Un opérande scalaire est implicitement diffusé à la forme de l'opérande vecteur.

Les opérations calculatoires sont appliquées aux filtres dimension par dimension. L'application d'une opération à une dimension est réalisée élément par élément, sur une longueur définie. Cette longueur est déterminée par le *contexte* de l'opération. Le contexte d'une opération est défini

1) pour une opération partie droite d'une affectation vectorielle : par les caractéristiques de la partie gauche de l'affectation. Ces caractéristiques sont éventuellement masquées si l'instruction est au sein d'un bloc vectoriel ou décrit (confer infra).

2) pour une opération qui n'est pas partie droite d'une affectation vectorielle : par les caractéristiques du premier opérande vecteur (filtre ou vecteur) de l'expression.

Le contexte d'une opération spécifie le nombre de dimensions des vecteurs traités et les longueurs de chacune de ces dimensions. Les opérandes doivent être de même forme. Pour chaque dimension de cette forme, une affectation vectorielle est réalisée sur la longueur de l'opérande gauche de l'affectation. Toutes les opérations de l'évaluation de la partie droite sont réalisées sur cette longueur. Cette règle est simple, elle autorise la génération d'une boucle unique pour les opérations formant l'instruction (sous condition de dépendance, bien entendu). Cependant, cette régle est inhibée pour une instruction appartenant à un bloc vectoriel (se référer à la section 3–3.6.1 ci-dessous).

A l'inverse, la localisation, sur une machine massivement parallèle, des traitements n'est pas définie en LSD2 : la réalisation d'une opération entre deux filtres produit un vecteur temporaire. Les caractéristiques

de ces vecteurs temporaires ne sont pas entièrement définies. Ces caractéristiques dépendent de l'implantation du langage sur la machine cible. Par exemple, la distribution d'un vecteur résultant de la somme de deux vecteurs qui ne sont pas identiquement distribués n'est pas définie. Cette non-définition laisse la liberté au compilateur de mettre en place des optimisations pour minimiser les communications inter-processeurs.

3–3.5.3 Instructions sur les vecteurs

Les vecteurs et les filtres sont manipulés en LSD2 via deux types d'instructions :

- les affectations vectorielles, et
- le passage en paramètre de vecteurs.

Affectation vectorielle

Une affectation vectorielle est une affectation dont la partie gauche est un vecteur ou un filtre de vecteur. Une telle opération modifie la valeur des éléments référencés par cette partie gauche avec la valeur de la partie droite de l'affectation. La sémantique définie pour cette instruction assure que la partie droite est entièrement évaluée préalablement au rangement dans la partie gauche.

LSD2 suppose donc, a priori, une éventuelle dépendance entre les parties droite et gauche d'une affectation. Une telle dépendance oblige le passage par une zone temporaire dans laquelle est produit le résultat de la partie droite; cette zone temporaire étant ensuite rangée dans la partie gauche.

Cependant, il est des cas où la non dépendance entre les parties droite et gauche peut être simplement acquise. Ces cas ont été exposés dans la section NO TAG ci-dessus. En l'absence de dépendance entre la partie droite et la partie gauche, la traduction de l'instruction d'affectation vectorielle est alors optimisée. Cette génération est produite à partir de la spécification de dépendance ou non-dépendance entre les filtres des vecteurs.

Passage de vecteurs en paramètre

LSD2 autorise le passage de vecteurs en paramètre. A l'inverse, le passage de filtres de vecteurs est considéré comme un passage de valeur temporaire en LSD2. Nous discutons ces deux points dans les paragraphes suivants.

Passage de vecteurs en paramètre Nous traitons ici du passage de vecteurs en paramètre et de la déclaration des vecteurs paramètres formels. Préalablement, indiquons que le passage des vecteurs est, en LSD2, réalisé par adresse. C'est-à-dire que les valeurs d'un vecteur passé en paramètre sont partagées par le vecteur paramètre formel.

Dans la suite, nous donnons les informations qui doivent correspondre entre un vecteur paramètre formel et un vecteur paramètre effectif, nous expliquons les différents moyens d'assurer cette correspondance en LSD2, nous traitons du passage des vecteurs dynamiques, et enfin, nous détaillons une implantation de ces passages de paramètres.

Lors du passage d'un vecteur, les informations suivantes doivent correspondre entre un vecteur paramètre formel et le vecteur paramètre effectif associé :

- le type, au sens type des éléments,
- le nombre de dimensions et la taille de chacune d'elles, et
- les spécifications de distribution, d'alignement et de projection.

Ces informations sont en effet nécessaires pour accéder les éléments d'un vecteur en mémoire. En LSD2, il est à la charge du programmeur d'assurer cette correspondance. Une telle correspondance est assurée par une déclaration appropriée du paramètre formel. Par exemple, cette déclaration utilise le même type vecteur que la déclaration du paramètre effectif,

```
VECTOR TYPE PAIM
      SIZE              1000
      FILTER IMPAIR     [1:1000:2]
      FILTER PAIR       [2:1000:2]
      NODEPENDENCE      PAIR, IMPAIR
      DISTRIBUTION      2
END
```

L'utilisation d'un type paramétré doit assurer que les valeurs des paramètres du type correspondent entre la déclaration du vecteur paramètre effectif et celle du vecteur paramètre formel. Par exemple, avec le type

```
TYPE VECTOR PAIMDECL (TAILLE)
      SIZE              TAILLE
      FILTER IMPAIR     [1:TAILLE:2]
      FILTER PAIR       [2:TAILLE:2]
      NODEPENDENCE      PAIR, IMPAIR
      DISTRIBUTION      2
END
```

Il est nécessaire que la valeur du paramètre *TAILLE* corresponde pour les deux déclarations.

Cependant, LSD2 n'oblige pas la concordance de type à être syntaxique; le programmeur doit juste assurer que les informations énoncées plus haut concordent pour les deux paramètres effectif et formel.

Déclaration ACTUAL des spécifications LSD2 propose un autre outil assurant la correspondance de ces informations entre les paramètres effectifs et formels. La déclaration d'un vecteur paramètre formel peut indiquer, pour la taille et/ou les spécifications de distribution, d'alignement et de projection, que celles-ci doivent être reprises depuis le paramètre effectif. Une telle déclaration utilise le mot clé **ACTUAL**. Par exemple, la déclaration du paramètre formel

```
VECTOR REAL DUMMY
      SIZE              ACTUAL
      DISTRIBUTION      2
END
```

autorise l'appel de vecteur de taille quelconque, de distribution égale à 2, d'alignement et de projection canoniques. La taille du paramètre formel sera reprise de celle du paramètre effectif lors de chaque appel. Bien entendu, l'utilisation de cette facilité, comme dans le cas des vecteurs automatiques, reporte de la compilation à l'exécution une partie des traitements à réaliser pour l'accès au vecteur.

Passage de vecteurs dynamiques en paramètre Le passage de vecteurs dynamiques doit être identifié explicitement en LSD2 comme le sont les vecteurs dynamiques. Pour autoriser la dynamicité d'un vecteur

reçu en paramètre, une fonction doit déclarer ce vecteur comme tel. Par exemple, un vecteur paramètre dont la taille est dynamiquement changée au sein d'une fonction est déclaré

```
VECTOR REAL DUMMYDYN
      SIZE        DYNAMIC
END
```

Une telle déclaration autorise ensuite l'allocation et/ou la réallocation dynamique du vecteur. Une telle déclaration oblige le paramètre effectif correspondant à être déclaré dynamique. Inversement, le passage en paramètre d'un vecteur dynamique n'oblige pas la déclaration du paramètre formel associé à spécifier le vecteur paramètre formel dynamique; toutefois, la dynamicité du paramètre effectif ne pourra être utilisée au sein de la fonction.

Implantation du passage de vecteur en paramètre Nous expliquons maintenant comment ces diverses fonctionnalités de passage de vecteurs en paramètre sont implantées.

Lors du passage d'un vecteur à une fonction, LSD2 doit assurer le passage des informations référencées par le paramètre formel. Ne pouvant déterminer la liste stricte des informations nécessaires, l'ensemble des informations pouvant être référencées sont passées, à savoir : la taille, et les trois spécifications de distribution, alignement et projection. L'utilisation effective de ces informations est liée à leur référence ou non dans la déclaration du paramètre formel (déclaration **ACTUAL**).

Le passage d'une adresse de base de la zone d'allocation assure le partage des éléments entre les paramètres effectif et formel. Pour les vecteurs non dynamiques, un passage en lecture seule des informations suffit. En effet, ces informations ne peuvent être modifiées par la fonction appelée.

Le passage d'un vecteur dynamique assure, au retour de la fonction appelée, la mise à jour de l'information de taille et des spécifications, celles-ci ayant pu être modifiées au sein de la fonction appelée. Cette fonctionnalité peut être implantée par un passage par adresse.

Les copies nécessaires à la réalisation de ces passages de paramètres ont un coût minime par rapport à des opérations de re-distribution ou ré-alignement et autorisent une grande souplesse lors de la déclaration des paramètres formels.

Passage de vecteurs temporaires Nous précisons quelques points à propos du passage de vecteurs temporaires en paramètre.

Notons tout d'abord que les filtres de vecteurs sont consommés préalablement au passage en paramètre. Les valeurs référencées par un filtre passé en paramètre sont copiées dans une zone temporaire. C'est ensuite cette zone temporaire qui est passée. Aucune copie n'a lieu au retour de la fonction.

Comme énoncé plus haut, les caractéristiques des vecteurs temporaires ne sont pas définies en LSD2. Cependant, pour autoriser le passage de vecteurs temporaires en paramètre, il est précisé que, dans ce cas, les vecteurs sont distribués et alignés selon les valeurs par défaut (allocation canonique). La déclaration du paramètre formel correspondant doit donc tenir compte de cette information. Elle se fait sur le modèle suivant

```
SUBROUTINE SBR (VTEMPO)
VECTOR INT VTEMPO
      SIZE  ACTUAL
END
...
```

Un appel à la fonction

```
CALL SBR (V1 + V2)
```

est valide quelles que soient les spécifications des vecteurs *V1* et *V2*. Notons que l'exécution d'un tel passage sur une machine massivement parallèle induit éventuellement une communication pour amener les valeurs du vecteur temporaire sur une allocation canonique.

Il est possible d'éviter un tel inconvénient en rangeant le résultat de l'expression dans une variable vecteur dont les spécifications sont identiques à celles des opérandes de l'expression et de passer ensuite cette variable à la fonction. La déclaration du paramètre formel doit alors tenir compte de ces spécifications (par une déclaration explicite ou **ACTUAL** des spécifications).

3–3.6 Notion de bloc

LSD2 propose plusieurs types de blocs d'instructions. Ces blocs d'instructions étendent les blocs d'instructions du langage hôte de LSD2. LSD2 distingue les blocs d'instructions traditionnels, les blocs vectoriels d'instructions et les blocs décrits d'instructions. LSD2 introduit, au sein de ces blocs, la déclaration de variables vecteurs temporaires dont la projection sur les machines à parallélisme de données peut être particulièrement efficace.

Notons que LSD2 n'impose pas qu'une instruction manipulant des vecteurs soit dans un bloc. L'utilisation de ces blocs autorise des factorisations et donc allège la programmation; ces factorisations aident aussi la génération de code par l'expression de "contextes" (longueur ou description) constants pour une suite d'instructions.

Un bloc d'instructions LSD2 est formé d'une partie déclarative et d'une séquence d'instructions. Ces blocs sont introduits en LSD2 pour les langages hôtes ne possédant pas une telle structure de blocs. Associée à ces blocs d'instructions, LSD2 intègre une notion de visibilité des variables, comme celle existant dans les langages possédant de tels blocs.

3–3.6.1 Blocs vectoriels d'instructions

Les blocs vectoriels LSD2 sont des blocs définissant une longueur de traitement des opérations vectorielles du bloc, et ce, pour chaque dimension des vecteurs du bloc.

Comme nous l'avons déjà décrit, il est important de pouvoir déterminer qu'une suite d'instructions vectorielles sont réalisées sur une longueur unique. Cette longueur est unique est exprimée par le programmeur. Toute expression scalaire est valide comme spécification de longueur; la valeur de l'expression peut ne pas être évaluable à la compilation.

Cette spécification de longueur est exprimée indépendamment pour chacune des dimensions des vecteurs manipulés dans le bloc. Par exemple

```
BLOC 10, 20
      VECTOR REAL TEMPOA, TEMPOB
            SIZE  10, 20
      END
      TEMPOA = MAT.SET1 + MAT.SET2
      TEMPOB = MAT.SET3 + MAT.SET4
      MAT.SET1 = TEMPOB
      MAT.SET3 = TEMPOA
END
```

les quatre– filtres *SETi* référençant des sous-ensembles deux dimensions d'éléments du vecteur *MAT*.

3–3.6.2 Blocs d'instructions décrites

Les blocs décrits sont une extension de la classique instruction **WHERE**.

Comme pour le **WITH** du langage EVA, le **WHERE** LSD2 indique une description par défaut pour toutes les instructions d'affectation du bloc. Cette description n'est pas limitée, comme en général il en est avec le **WHERE**, à un vecteur de valeurs booléennes; elle peut aussi être composée d'une séquence d'index ou d'un triplet *borne inférieure*, *borne supérieure* et *pas*, un opérateur de décalage ou de rotation peut également être présent. Cependant, une telle description par défaut ne peut comporter de description de réduction ou de diffusion, ces deux descriptions ne pouvant apparaître en partie droite *et* partie gauche d'une affectation. Par exemple,

```
WHERE [1 : 1000 : 2]
      V = V1 + V2
      W = F (A)
END
```

L'instruction d'affectation de *V* est uniquement réalisée sur les éléments impairs, d'indice inférieur à 1000 des vecteurs. L'exécution de la seconde instruction est réalisée comme suit. La fonction *F* est appelée avec *A* en paramètre. L'instruction d'affectation de la valeur retournée par cette fonction au vecteur *W* est elle réalisée sous la description par défaut du bloc **WHERE**. La description des paramètres des fonctions appelées au sein d'un bloc **WHERE** peut être réalisée explicitement par le programmeur.

Comme pour les blocs vectoriels d'instructions, il est possible de spécifier une description pour chacune des dimensions des vecteurs manipulés dans le bloc :

```
VECTOR INT A
      SIZE 1000
END
VECTOR REAL V, V1, V2
      SIZE 1000, 1000
END
WHERE [1 : 1000 : 2, A .GE. 0]
      V = V1 + V2
END
```

L'affectation est réalisée sur les éléments des lignes impaires de numéro *i*, tel que *A(i)* est positif.

Notons aussi la différence avec les extensions parallèles du **IF** proposées pour les machines massivement parallèles. Ces extensions ne se veulent pas des descriptions par défaut, mais des structures de contrôle parallèles. Leur exécution est liée à la manipulation d'un masque d'activité des processeurs. En particulier, la construction **WHERE** LSD2 ne "propage" pas la description au travers les appels de fonctions.

3–3.6.3 La classe d'allocation STRIP

LSD2 définit donc des blocs d'instructions. Au sein de ces blocs, il est possible de déclarer des variables vecteurs temporaires **STRIP**. **STRIP** est une classe d'allocation introduite par LSD2. Présentons un exemple

```
VECTOR REAL PAIM
      SIZE              N
      FILTER IMPAIR     [1:N:2]
      FILTER PAIR       [2:N:2]
      NODEPENDENCE PAIR, IMPAIR
END

WHERE [PAIR .LT. IMPAIR]
      STRIP REAL TMP
      TMP          = PAIR
      PAIR         = IMPAIR
      IMAPAIR      = TMP
END
```

L'expression de la non dépendance entre les deux filtres *PAIR* et *IMPAIR* autorise la production d'une boucle unique de strip-mining pour l'ensemble des instructions du bloc **WHERE**. La variable **STRIP** *TMP* ne sera donc pas allouée en mémoire. Elle sera simplement associée à un registre, son allocation est réalisée *strip par strip* dans un registre vectoriel lors de chaque itération de la boucle. (Le terme registre vectoriel dénote aussi bien un registre vectoriel d'une machine vectorielle pipeline qu'un registre de chaque processeur d'une machine massivement parallèle.)

L'avantage d'une telle déclaration sur celle d'un vecteur temporaire local est clair. Aucune place mémoire n'est nécessaire à l'allocation de la variable **STRIP**. Aucun transfert mémoire ne sera non plus nécessaire lors de son utilisation.

3–4 Conclusion

Nous avons présenté le langage LSD2 à parallélisme de données et l'environnement de compilation pour machines à parallélisme de données PARTNER. PARTNER est constitué de différents niveaux d'expression du parallélisme de données. Deux langages machines virtuelles, LIVP et LIMP, ont été conçus pour les machines pipelines vectorielles et les machines massivement parallèles. La production de code pour ces langages est réalisée depuis un langage unique : LSD2, via le langage intermédiaire LOLLA. LSD2 fournit des outils d'expression explicite des algorithmes vectoriels indépendamment d'un langage hôte, tel C ou FORTRAN. LSD2 est bâti autour d'une construction de vecteur regroupant différentes spécifications.

La spécification des dépendances entre les accès aux éléments des vecteurs évite la duplication des boucles et l'utilisation de temporaires pour la réalisation d'instructions d'affectations vectorielles. La spécification de distribution et d'alignement diminue les communications inter-processeurs sur une machine massivement parallèle.

De part son approche, LSD2 autorise l'écriture portable d'algorithmes vectoriels sur stations de travail, machines pipelines vectorielles, et machines massivement parallèles. (Les stations de travail sont utilisées pour la mise au point, les machines à parallélisme de données pour la production effective). Les constructions de LSD2 expriment le parallélisme de données et peuvent être projetées sur les différentes architectures.

LSD2 est un ensemble réduit de primitives et de constructions utilisées dans un langage hôte traditionnel. Cette approche autorise la réécriture progressive de programmes C ou FORTRAN existants.

Une telle écriture progressive est aussi aidée par la séparation entre l'expression de l'algorithme d'une part, et la projection effective sur une machine en vue d'améliorer les performances d'autre part.

Cette séparation est due à l'encapsulation dans les objets vecteurs des informations exprimées par le programmeur sur ses données. Cette encapsulation, associée à un typage fort des vecteurs, assure la génération de codes de bonne qualité lors du passage de vecteurs en paramètre.

Des extensions sont d'ores et déjà prévue à LSD2. En particulier, nous étudions la possibilité d'inclure de nouvelles opérations de descriptions pour les accès multi-dimensionnels pour lesquels il serait intéressant de pouvoir lier les accès aux différentes dimensions. Il serait ainsi par exemple possible d'accéder une bande diagonale d'une matrice. De manière évidente, toutes les combinaisons entre les accès des différentes dimensions ne peuvent être permises sous peine de ne pouvoir générer du code de bonne qualité.

Chapitre 3—Section 2

Spécification des dépendances et génération de code vectoriel

Nous étudions ici la génération de code depuis un langage vectoriel explicite. Cette génération de code est basée sur des informations de dépendances entre vecteurs. Les techniques présentées ici s'adaptent parfaitement au langage LSD2; les informations nécessaires à la génération de code étant présentes dans un code LSD2. Cependant, ces techniques sont plus générales et peuvent être appliquées dès que certaines propriétés peuvent être identifiées préalablement à la phase de génération de code. Si cette identification n'est pas immédiate, on peut envisager qu'elle soit réalisée par des outils de découverte automatique des dépendances existants et des extensions de ceux-ci.

Nous donnons tout d'abord quelques définitions et remarques liminaires.

Vecteur et filtre Nous définissons un vecteur, au niveau du langage source, comme l'union d'un ensemble d'éléments (sa zone d'allocation) et d'un ensemble de filtres. Un filtre caractérise un accès à un ensemble de composantes parmi les éléments du vecteur. Tout accès partiel est réalisé via un filtre. Nous supposons qu'il n'existe pas d'opérateur de description. Les descriptions de vecteurs sont obtenues par utilisation d'un filtre de vecteur. En particulier l'accès à tous les éléments du vecteur est réalisé par un filtre (souvent appelé *all*).

Nous disons que deux filtres appartiennent à un même vecteur s'ils référencent tous deux des éléments d'un même vecteur; ils peuvent alors référencer des éléments communs.

Cette introduction des filtres de vecteurs est principalement syntaxique. Deux alternatives sont envisageables : 1—le langage source ne possède effectivement pas d'opérateur de description et implante ces opérations par des accès à des filtres (LSD2). 2—le langage source possède des opérations de descriptions vectorielles; ces opérations peuvent être réduites à l'utilisation de filtres de vecteurs.

Les descriptions Un filtre peut être : 1—vide; il sélectionne alors tous les éléments du vecteur (c'est le filtre *all*). 2—formé d'un vecteur de bits; il sélectionne alors les éléments du vecteur dont le bit correspondant est à vrai. 3—formé d'un triplet *borne inférieure*, *borne supérieure*, *pas*. 4—formé d'un

vecteur d'index. L'opération réalisée est alors une opération de gather/scatter. Le modèle de fonctionnement ne garantit pas le résultat en cas de doublet dans le vecteur d'index. C'est-à-dire que suite à une affectation telle

```
I = (1, 2, 1, 3)
V [I] = (1, 2, 3, 4)
```

la valeur de *V* [1] peut être 1 ou 3. Cette limitation est moins contraignante que celle de langages comme FORTRAN 90 qui interdisent la présence de doublet dans un vecteur d'index utilisé comme description de vecteur en partie gauche d'une affectation.

Caractéristiques de la machine cible La machine cible est supposée travailler sur des registres vectoriels de taille VRS (Vector Register Size). La longueur effective sur laquelle sont traités les vecteurs est contrôlée par le registre VL (Vector Length). (Pour une machine massivement parallèles à registres, VRS est le nombre de processeurs; le comportement de VL est simulé par positionnement du bit de contexte.)

Boucle Dans cette section, le terme *boucle* se réfère aux boucles découpant les vecteurs en *strips*. Un strip est une tranche de vecteur de longueur VRS (ou moins) pouvant être traitée par la machine cible.

SIV—Séquence d'instructions vectorielles Une séquence d'instructions vectorielles (SIV) est une suite d'instructions vectorielles traitant des vecteurs de même longueur. Les instructions retenues au sein d'une SIV ne sont que des affectations vectorielles; toute autre instruction termine une SIV. De même, le flot d'exécution ne peut commencer qu'au début d'une SIV; il n'existe pas de label autorisant l'exécution d'une partie seulement d'une SIV. Des dépendances entre les différentes instructions de la séquence peuvent interdire la production d'une unique boucle de code pour la séquence.

BIV—Bloc d'instructions vectorielles Un bloc d'instructions vectorielles (BIV) est une séquence d'instructions vectorielles qui à la génération de code ne produit qu'une boucle. Les dépendances apparaissant entre les instructions du bloc ne doivent pas gêner cette compilation en une boucle unique.

Θ—Ordre d'exécution des instructions Nous définissons un ordre sur les instructions correspondant à l'ordre dans lequel elles doivent être exécutées. Cet ordre est noté Θ. Soient deux instructions *S1* et *S2* d'une SIV,

$$S1 \ \Theta \ S2$$

indique que *S1* doit être exécutée avant *S2*. Cet ordre n'est défini ici que pour les instructions d'une même exécution d'une séquence d'instructions vectorielles. Il correspond donc aussi à l'ordre d'apparition des instructions dans le code; aucune instruction de rupture de séquence n'existant au sein d'une SIV.

Affectation vectorielle La sémantique d'une affectation vectorielle telle

V = EXPRESSION

avec *V* un vecteur et *EXPRESSION* une expression scalaire ou vectorielle, est définie de telle sorte que l'instruction soit équivalente aux instructions

#TEMP = EXPRESSION
V = #TEMP

L'évaluation complète de l'expression en partie droite avant le rangement dans le vecteur en partie gauche prend en compte d'éventuelles dépendances entre les parties droite et gauche de l'affectation. Pour

respecter cette sémantique, la génération de code d'une telle instruction utilise une zone temporaire et est basée sur deux boucles. Une première boucle range l'évaluation de la partie droite dans la zone temporaire; une seconde boucle affecte la valeur de la zone temporaire dans la partie gauche.

Cette sémantique est celle retenue par la plupart des langages vectoriels : FORTRAN 90, CFT77 des machines Cray, CM FORTRAN de la Connection Machine, EVA (opérateur =), et LSD2, etc.

Dans la suite de ce chapitre, nous présentons les trois grands types de dépendances entre instructions vectorielles que nous différencions (ces dépendances sont qualifiées de *totales*); nous indiquons les conséquences sur la génération de code. Nous détaillons ensuite certains raffinements que l'on peut apporter à ces dépendances en fonction de caractéristiques des filtres; nous identifions alors des dépendances qualifiées de *partielles*. Nous étudions ensuite les conséquences de ces dépendances partielles sur la génération de code. Nous justifions de ces dépendances partielles par la présentation d'une comparaison du code et des performances obtenus par notre technique et par le compilateur FORTRAN sur Cray Y-MP.

3–5 Les dépendances entre les instructions d'une séquence d'instructions vectorielles

Nous présentons ici les différentes classifications de dépendances totales introduites entre les filtres d'un vecteur d'une part, et entre les instructions d'une séquence d'instructions vectorielles d'autre part. Nous en indiquerons les conséquences sur la décomposition des séquences d'instructions vectorielles en bloc d'instructions vectorielles.

Nous ne supposons aucune dépendance entre deux filtres n'appartenant pas au même vecteur. Entre deux utilisations de filtres d'un même vecteur, nous distinguons deux types de dépendances. Ces deux dépendances sont caractérisées par l'identité ou non des deux filtres. Si les deux utilisations référencent le même filtre, nous qualifions la dépendance de *dépendance de nom*. Sinon nous la qualifions de *dépendance hétéronyme*. Si deux filtres *filtre1* et *filtre2* sont dépendants, nous notons

filtre1 δ *filtre2*.

Deux instructions sont dites dépendantes si elles contiennent deux filtres dépendants. Les différentes dépendances sont identifiées par les dépendances entre les filtres et la position (à droite ou à gauche d'une affectation) des filtres de vecteurs. La classification proposée correspond à une classification des conséquences des dépendances sur le code généré. Les dépendances présentées ici sont des dépendances totales; des dépendances partielles peuvent alléger ces conséquences; nous traiterons de ces dépendances partielles plus loin dans ce chapitre.

Soient deux instructions *S1* et *S2* d'une même SIV telles que *S1* Θ *S2*, nous notons

S1 δ *S2*

pour signifier que les deux instructions sont dépendantes. Nous distinguons :

- les dépendances de nom, notées *S1* δ_v *S2*
- les dépendances hétéronymes, notées *S1* δ_η *S2*, et

- les load-store dépendances, notées *S1* $\delta\lambda$ *S1*.

Nous détaillons ces dépendances dans les sous-sections suivantes.

3–5.1 Les dépendances de nom $\delta\nu$

Une dépendance de nom est caractérisée par la présence d'un filtre en partie gauche d'une affection dans une autre instruction du segment d'instructions vectorielles. Deux cas sont distingués selon que l'affectation a lieu dans une instruction antérieure ou postérieure à la seconde référence au filtre :

```
S1:   filtre = .......
S2:   .... filtre ....
```

ou

```
S1:   .... filtre ....
S2:   filtre = .......
```

Nous notons *S=* l'instruction d'affectation de *filtre* (*S1* dans le premier cas, *S2* dans le second). Les deux instructions *S1* et *S2* peuvent appartenir à la boucle; les éléments du vecteur affectés par l'instruction *S=* lors d'une itération de boucle ne sont pas utilisés dans d'autres itérations. (Sauf dans le cas de doublets dans un filtre composé d'un vecteur d'index. Dans un tel cas la non garantie de résultat nous permet aussi de localiser les deux instructions dans la même boucle, voir ci-dessous).

Par contre du fait de l'affectation de *filtre* dans *S=* et de son utilisation dans l'autre instruction, les deux instructions ne peuvent pas être permutées.

L'hypothèse de non prévisibilité du résultat en cas de gather/scatter

Nous discutons ici de la présence de doublets dans une liste d'index d'un filtre de vecteur. Faisons par exemple l'hypothèse que les éléments d'un vecteur sont affectés dans l'ordre des indices du vecteur d'index lors d'une opération de scatter. Les instructions

```
I = (1, 2, 1, 3)
V [I] = (1, 2, 3, 4)
```

rangent alors 3 dans *V* [1]. Nous allons montrer que sous cette hypothèse, deux instructions liées par une dépendance de nom ne peuvent appartenir à une même boucle. Soit l'exemple suivant

```
      I = (1, 2, 1, 2)
S1:   V [I] = (1, 2, 3, 4)
S2:   A = V [I]
```

Les deux instructions *S1* et *S2* sont liées par une dépendance de nom (*V [I]*). Si deux boucles sont générées, une pour chacune des instructions *S1* et *S2*, le résultat obtenu pour *A* est (3, 4, 3, 4). Supposons maintenant que les deux instructions soient générées dans une même boucle et qu'une itération de boucle traite 2 éléments (VRS = 2). Le résultat obtenu pour *A* est (1, 2, 3, 4).

Analysons le résultat obtenu en cas de doublets dans une liste d'index. Dans une itération de boucle, sont transférées des valeurs correspondant à des indices "doubles" qui ne sont pas le dernier indice traité. Cette

valeur transférée n'est donc pas la valeur définitive de l'élément. Toutefois, ne garantissant pas le résultat lors d'apparition de doublets dans une liste d'index, cette valeur transférée aurait pu être la valeur définitive. Nous acceptons donc le transfert de telles valeurs "provisoires".

En cas de refus de cette proposition, les conclusions de nos travaux restent valables si l'on interdit la présence de doublets dans un filtre liste d'index, partie gauche d'une affectation.

3–5.2 Les dépendances hétéronymes $\delta\eta$

Une dépendance hétéronyme est caractérisée par la présence de deux filtres d'un même vecteur dans deux instructions différentes d'une SIV. Au moins l'un des filtres est partie gauche d'une affectation vectorielle. Nous distinguons donc les deux cas

```
S1:   filtre1 = .......
S2:   .... filtre2 ....
```

et

```
S1:   .... filtre2 ....
S2:   filtre1 = .......
```

Les deux instructions *S1* et *S2* ne peuvent pas appartenir à la même boucle. En effet, dans le premier cas, un élément du vecteur utilisé par le filtre *filtre2* pourrait être modifié par l'instruction *S1* via le filtre *filtre1* lors d'une itération suivante; la valeur utilisée par *filtre2* n'est alors pas la bonne. Dans le second cas, un élément du vecteur peut être modifié lors d'une itération par l'instruction *S2* via le filtre *filtre1* et être ensuite utilisé par le filtre *filtre2*; ce qui ne correspond pas à la sémantique de la séquence d'instructions.

3–5.3 Les load-store dépendances $\delta\lambda$

Une load-store dépendance est une dépendance entre la partie droite et gauche d'une même affectation vectorielle. Deux filtres différents d'un vecteur apparaissent dans une affectation vectorielle. Un des filtres est en partie gauche de l'affectation.

```
S:    filtre1 = .... filtre2 ....
```

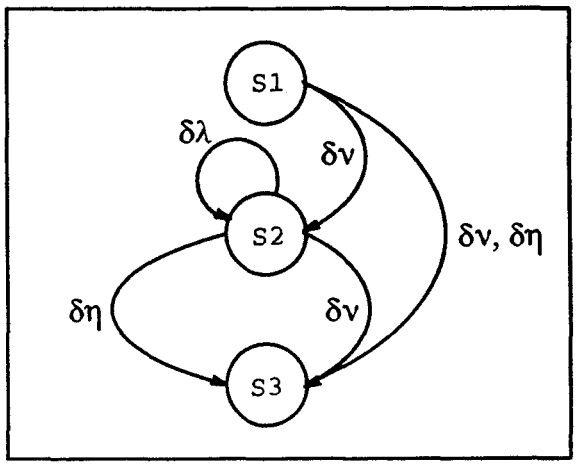
Pour respecter la sémantique de l'affectation vectorielle, il nous faut générer deux boucles. Une première boucle range le résultat de l'évaluation de la partie droite dans une zone temporaire, la seconde boucle range les valeurs de cette zone temporaire dans *filtre1*. (Ces deux boucles ne peuvent être fusionnées du fait même de la dépendance.)

3–5.4 Exemple

Considérons l'instruction de permutation de deux sous ensembles d'éléments d'un vecteur *V*. Les filtres *set1* et *set2* référencent respectivement les éléments de chacun des sous ensembles de *V*. Cette opération d'échange est codée en une séquence d'instructions vectorielles unique :

```
S1:   TMP  = SET1
S2:   SET1 = SET2
S3:   SET2 = TMP
```

On a *S1* Θ *S2* et *S2* Θ *S3*. On obtient le graphe de dépendances de la figure 3–11.



δν Dép. de nom
δη Dép. hétéronyme
δλ Load-store dep.

Figure 3–11. Graphe de dépendances entre instructions.

La séquence d'instructions vectorielles sera découpée en blocs d'instructions vectorielles comme suit (*#TMP* est une variable temporaire utilisée uniquement par le compilateur) :

```
/* premier bloc d'instructions vectorielles (BIV) */
      TMP  = SET1
      #TMP = SET2

/* second BIV */
      SET1 = #TMP

/* troisième BIV */
      SET2 = TMP
```

Nous fournissons l'algorithme de production des BIVs à partir d'une SIV et des dépendances entre les filtres dans la sous-section NO TAG.

3–5.5 Composition des dépendances vectorielles et des dépendances traditionnelles

Traditionnellement, on distingue trois types de dépendances entre deux instructions suivant les intersections des ensembles IN et OUT [KMC72]. L'ensemble IN d'une instruction est l'ensemble des références en mémoire lues par cette instruction. L'ensemble OUT d'une instruction est l'ensemble des références en mémoire écrites par cette instruction. Considérant deux instructions *S1* et *S2*, telles que *S1* Θ *S2*, on observe une dépendance de flot (*t*rue dependence) entre les instructions *S1* et *S2*, notée $S1\ \delta^t\ S2$, si

$$\text{OUT}(S1) \cap \text{IN}(S2) \neq \emptyset,$$

une *a*nti-dépendance, notée $S1\ \delta^a\ S2$, si

IN (*S1*) ∩ OUT (*S2*) ≠ Ø,

et une dépendance de sortie (*o*utput dependence), notée *S1* δ^o *S2*, si

OUT (*S1*) ∩ OUT (*S2*) ≠ Ø.

Le tableau de la figure 3–12 indique les compositions de ces dépendances traditionnelles et de la classification introduite ici. Les combinaisons impossibles sont marquées d'un trait —.

		Dépendances habituelles.		
		Dépendance de flot δ^t	Anti-dépendance δ^a	Dépendance de sortie δ^o
Dépendances vectorielles.	Load-store λ	—	$\delta\lambda$	—
	Hétéronymes η	$\delta\eta^t$	$\delta\eta^a$	$\delta\eta^o$
	De nom ν	$\delta\nu^t$	$\delta\nu^a$	$\delta\nu^o$

Figure 3–12. Compositions des dépendances habituelles et des dépendances vectorielles.

3–5.6 Algorithme de production des blocs d'instructions vectorielles

Nous présentons ici l'algorithme produisant les blocs d'instructions vectorielles (BIVs) à partir d'une séquence d'instructions vectorielles (SIV) et des dépendances apparaissant dans cette séquence. Soit *S1* la première instruction de la séquence. On considère toutes les instructions *S*i de la séquence telles qu'il existe une chaîne de dépendances entre *S1* et *S*i. La génération d'une instruction non liée avec *S1* par une chaîne de dépendances peut intervenir indépendamment de la génération des instructions liées à *S1*.

Nous ne considérons pas les instructions load-store dépendantes (qui seront traitées ensuite). Les instructions d'une SIV sont générées dans un BIV. Une instruction *Scourante* n'est pas générée dans le BIV courant, et oblige la terminaison du BIV courant et la création d'un nouveau BIV, si et seulement s'il existe une instruction *S* produite dans le BIV courant telle que *S* $\delta\eta$ *Scourante*. En effet, les dépendances de nom $\delta\nu$ n'obligent pas la rupture des BIVs.

Notre algorithme n'est pas optimal. En effet la rupture de BIVs intervient toujours entre deux instructions (si l'on exclut les instructions load-store dépendantes). Nous présentons un cas où il pourrait être intéressant de faire cette rupture au sein d'un instruction :

```
S1:   ... filtreA ...
S2:   ... filtreB ...
S3:   ... filtreC1 ...
S4:   filtreC2 = ... filtreA ... filtreB ...
```

Les filtres *filtreA* et *filtreB* appartiennent à deux vecteurs différents. Les filtres *filtreC1* et *filtreC2* appartiennent au même troisième vecteur. Selon notre algorithme, nous produisons deux BIVs, un premier pour les instructions *S1* à *S3* et un second pour l'instruction *S4*. Les filtres *filtreA* et *filtreB* devront donc être rechargés en registre dans ce second BIV alors qu'ils l'ont déjà été dans le premier. Les performances pourraient être améliorées par l'introduction d'un vecteur temporaire mémorisant l'évaluation de la partie droite de *S4* lors de la première BIV; seul le temporaire aurait à être chargé :

```
/* premier BIV */
S1:   ... filtreA ...
S2:   ... filtreB ...
S3:   ... filtreC1 ...
S4a:  #TMP = ... filtreA ... filtreB ...
/* second BIV */
S4b:  filtreC2 = #TMP
```

De telles optimisations ne sont pas mises en place par notre algorithme. Nous donnons ici cet algorithme et détaillons ensuite la gestion des instructions load-store dépendantes.

```
/* On considère une séquence d'instructions vectorielles SIV et les dépendances entres les instructions de la SIV. L'algorithme produit le découpage de cette SIV en blocs d'instructions vectorielles BIVs. */
/* La séquence d'instructions est référencée par la variable SIV, l'instruction courante par la variable Scourante, la première instruction du BIV courant SfirstBIV. */
créer_un_nouveau_BIV
pour toute instruction Scourante ∈ SIV faire
      si Scourante δλ Scourante alors
            traiter_l_instruction_load_store_dépendante (Scourante)
      sinon
            si  il existe S telle que SfirstBIV Θ S Θ Scourante et S δη Scourante alors
                  finir_le_BIV_courant
                  créer_un_nouveau_BIV
            fsi
            générer Scourant dans le BIV courant
      fsi
fait
finir_le_BIV_courant
```

L'algorithme *traiter_l_instruction_load_store_dépendante* de gestion des load-store dépendances est discuté ci-après.

Gestion des load-store dépendances

Nous présentons ici les transformations induites par une instruction load-store dépendante. Les 4 instructions *S1*, *S2*, *S3*, et *S4* représentent les 4 cas de figures de dépendance avec l'instruction load-store dépendante *Sc*. Nous supposons d'abord *Si* Θ *Sc*.

```
S1:   filtre1 = ...
S2:   ... = filtre1
S3:   filtre2 = ...
S4:   ... = filtre2

Sc:   filtre1 = filtre2
```

Nous avons les dépendances de nom

S1 $\delta\nu^{o}$ *Sc*, *S2* $\delta\nu^{a}$ *Sc*, *S3* $\delta\nu^{t}$ *Sc*

et les dépendances hétéronymes

S1 $\delta\eta^{t}$ *Sc* *S3* $\delta\eta^{o}$ *Sc*, *S4* $\delta\eta^{a}$ *Sc*.

L'instruction *Sc* est remplacée par *Sd* et *Sg* :

```
Sd:   #TMP = filtre2
Sg:   filtre1 = #TMP
```

avec une dépendance *Sd* $\delta\eta^{a}$ *Sg* due aux filtres et une dépendance *Sd* $\delta\nu^{t}$ *Sg* liée à l'utilisation de *#TMP*. Les dépendances avec l'instruction *Sc* sont transformées en dépendances avec *Sd* et *Sg* suivant le tableau de la figure 3–13. Les caractères hétéronyme/de nom d'une part, et de flot/anti-/de sortie d'autre part, des dépendances ne sont pas affectés par la transformation.

Dépendance entre *S* et *Sc*	Dépendance induite entre *S* et *Sd*	Dépendance induite entre *S* et *Sg*
$\delta\nu^{t}$	$\delta\nu^{t}$	—
$\delta\nu^{o}$	—	$\delta\nu^{o}$
$\delta\nu^{a}$	—	$\delta\nu^{a}$
$\delta\eta^{t}$	$\delta\eta^{t}$	—
$\delta\eta^{o}$	—	$\delta\eta^{o}$
$\delta\eta^{a}$	—	$\delta\eta^{a}$

Figure 3–13. Conversion des dépendances *S* δ *Sc* en dépendances *S* δ *Sd* et *S* δ *Sg*.

Pour des instructions *Si* telles que *Sc* Θ *Si*, nous obtenons des transformations identiques des dépendances (figure 3–14).

Dépendance entre *Sc* et *S*	Dépendance induite entre *Sd* et *S*	Dépendance induite entre *Sg* et *S*
$\delta\nu^{t}$	—	$\delta\nu^{t}$
$\delta\nu^{o}$	—	$\delta\nu^{o}$
$\delta\nu^{a}$	$\delta\nu^{a}$	—
$\delta\eta^{t}$	$\delta\eta^{t}$	—
$\delta\eta^{o}$	$\delta\eta^{o}$	—
$\delta\eta^{a}$	—	$\delta\eta^{a}$

Figure 3–14. Conversion des dépendances *Sc* δ *S* en dépendances *Sd* δ *S* et *Sg* δ *S*.

On obtient donc l'algorithme de traitement des load-store dépendances suivant :

```
procédure traiter_l_instruction_load_store_dépendante (Scourante)
/* On scinde l'instruction en 2 et met à jour les dépendances */
Sdroite = insérer_une_instruction (#TMP=Scourante.partie_droite)
Sgauche = insérer_une_instruction (Scourante.partie_gauche= #TMP)
créer_une_dépendance (Sdroite δη^a Sgauche)
créer_une_dépendance (Sdroite δv^t Sgauche)
pour toute instruction S telle que SfirstBIV Θ S Θ Scourante faire
      si S δ Scourante alors
            créer_une_dépendance (S δ Sgauche ou S δ Sdroite) selon le tableau 3–13
fait
pour toute instruction S telle que Scourante Θ S Θ dernière_instruction_de (SIV) faire
      si Scourante δ S alors
            créer_une_dépendance (Sgauche δ S ou Sdroite δ S) selon le tableau 3–14
fait
fin procédure
```

3–6 Les load-store dépendances partielles

Dans la section précédente, nous avons détaillé la production de blocs d'instructions vectorielles à partir d'une séquence d'instructions vectorielles et des dépendances entre les instructions formant la séquence. Ces dépendances ont été qualifiées de totales; une dépendance entre deux filtres obligeait la scission en deux blocs d'instructions vectorielles. Nous présentons maintenant des dépendances qualifiées de partielles entre deux filtres d'un même vecteur. Les conséquences de telles dépendances sur la génération de code sont moindres que celles entraînées par une dépendance totale.

Cette section présente ces dépendances partielles et étudie la génération d'instructions partiellement load-store dépendantes. A l'inverse d'une load-store dépendance, la génération d'une instruction partiellement load-store dépendante n'oblige pas la production de deux boucles et le passage par un temporaire.

Après avoir rappelé le code produit par une instruction load-store dépendante totale et par une instruction sans dépendance, les sous-sections suivantes introduisent deux types de load-store dépendances : les semi load-store dépendances et les bloc load-store dépendances. Nous donnons à chaque fois les conditions caractérisant ces dépendances; présentons un exemple de code dans lequel une telle dépendance apparaît et donnons le code généré pour une instruction load-store dépendante. Nous discuterons ensuite d'une évaluation des performances obtenues à partir des telles dépendances.

3–6.1 Génération d'une instruction d'affectation vectorielle

Nous rappelons ici le code à produire pour une instruction d'affectation vectorielle. Ce code est donné dans un pseudo-assembleur tel qu'il est défini en annexe. Notons la représentation des instructions selon le format

<nom d'instruction> <opérande cible> <opérandes source(s)>

Dans le cas général, c'est-à-dire en supposant une dépendance entre les parties droite et gauche de l'affectation, la prise en compte de la sémantique d'une affectation vectorielle load-store dépendante totale est réalisée par la production de deux boucles. La première boucle range l'évaluation de la partie droite dans une zone temporaire. La seconde boucle affecte les valeurs de la zone temporaire dans la partie gauche de l'affectation. Le code type à produire pour une instruction telle que *Filtre1 = Filtre2 + 3* load-store dépendante totale (noté δλ) est le suivant :

```
-- Boucle d'évaluation de la partie droite et de rangement dans un temporaire (TEMP)
      -- VL contient le nombre d'éléments à traiter lors de la première itération
      -- S1 contient le nombre d'itérations
      -- A1 est l'index du premier élément à traiter dans une itération
LOOP1 :
      VLOAD       V1              Filtre2 [A1]
      VADD        V1              #3
      VSTORE      TEMP [A1]       V1
      ADD         A1              VL
      MOVE        VL              VRS
      DECR        S1
      JNZ         S1              LOOP1
-- Instruction assurant la prise en compte des dépendances entre les deux boucles
CMR
-- Boucle de rangement du temporaire dans la partie gauche
      -- Ré-initialisation de VL, S1 et A1
LOOP2 :
      VLOAD       V1              TEMP [A1]
      VSTORE      Filtre1 [A1]    V1
      ADD         A1              VL
      MOVE        VL              VRS
      DECR        S1
      JNZ         S1              LOOP2
```

En l'absence de dépendance entre la partie droite et la partie gauche, une affectation vectorielle produit un code basé sur le modèle suivant : (code pour *Filtre1 = Filtre2 + 3*)

```
-- Boucle d'évaluation de la partie droite et de rangement dans la partie gauche
      -- VL contient le nombre d'éléments à traiter lors de la première itération
      -- S1 contient le nombre d'itérations
      -- A1 est l'index du premier élément à traiter dans une itération
LOOP  :
      VLOAD       V1              Filtre2 [A1]
      VADD        V1              #3
      VSTORE      Filtre1 [A1]    V1
      ADD         A1              VL
      MOVE        VL              VRS
      DECR        S1
      JNZ         S1              LOOP
```

Nous évitons le passage par une zone temporaire et le découpage en deux boucles. Le code généré pour les dépendances partielles que nous allons présenter sera intermédiaire entre les deux cas extrêmes présentés ici.

3–6.2 Semi load-store dépendance

Nous présentons les caractéristiques d'une dépendance partielle qualifiée de *semi-dépendance* et notée à l'aide du signe $\nmid$. Nous précisons les conditions sous lesquelles deux filtres sont semi-dépendants; nous étudions ensuite le code devant être généré pour une affectation semi load-store dépendante (dont les parties droite et gauche sont semi-dépendantes).

Définition de la semi-dépendance

Nous disons qu'un filtre *Filtre1* est semi-dépendant d'un filtre *Filtre2* du même vecteur, et nous le notons

$$Filtre1 \nmid Filtre2$$

si $\forall i \in [1 : VLG], \{ |Filtre1(i)| \} \cap \{ |Filtre2(j)| / j \in [i + 1 : VLG] \} = \emptyset$,

$|Filtre(i)|$ désignant l'indice de l'élément correspondant à *Filtre* (i) dans le vecteur. VLG est la taille (nombre d'éléments) des filtres *Filtre1* et *Filtre2*.

C'est-à-dire que pour un indice *i* donné dans les filtres, l'élément du vecteur référencé par *Filtre1*(i) ne sera plus référencé par les éléments de *Filtre2* d'indice supérieur à *i* (figure 3–15).

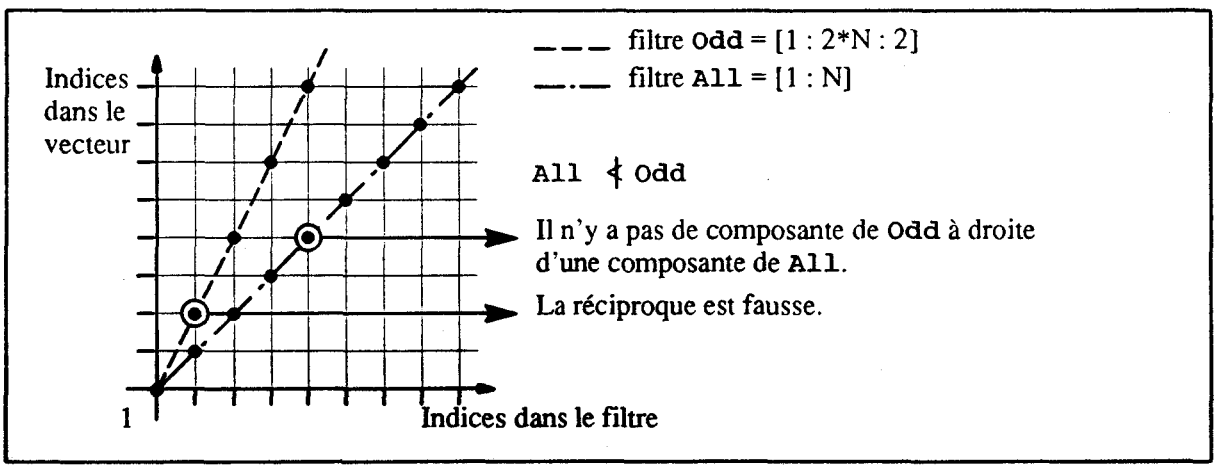


Figure 3–15. Exemple de semi-dépendance.

Génération d'une instruction semi load-store dépendante

Une instruction d'affectation vectorielle est dite semi load-store dépendante si le filtre partie gauche de l'affectation et la partie droite de l'affectation sont semi-dépendants. La génération d'une instruction d'affectation vectorielle semi load-store dépendante évite la génération de deux boucles et l'utilisation d'une zone temporaire. Nous précisons les différents codes à générer pour ce faire.

Une instruction telle que *Filtre1* = *Filtre2* + *3* avec *Filtre1* $\nmid$ *Filtre2* produira un code construit sur le modèle suivant (une boucle unique sans utilisation de zone temporaire) :

```
-- Boucle d'évaluation et de rangement
      -- VL contient le nombre d'éléments à traiter lors de la première itération
      -- S1 contient le nombre d'itérations
      -- A1 est l'index du premier élément à traiter dans une itération
LOOP  :
      VLOAD       V1              Filtre2 [A1]
      VADD        V1              #3
      VSTORE      Filtre1 [A1]    V1
      ADD         A1              VL
      MOVE        VL              VRS
      DECR        S1
      JNZ         S1              LOOP
```

En effet d'après la définition de la semi-dépendance, un élément du vecteur écrit par le filtre *Filtre1* lors d'une itération ne sera plus lu via *Filtre2* dans les itérations suivantes; les valeurs des éléments chargés via *Filtre2* sont donc toujours celles des éléments avant la boucle. Nous respectons ainsi la sémantique de l'opération.

Considérons maintenant le cas d'une instruction telle *Filtre2 = Filtre1 + 3*. On suppose toujours *Filtre1* $\nmid$ *Filtre2*. La prise en compte de la dépendance entre la partie droite et la partie gauche est réalisée par un traitement des itérations dans l'ordre décroissant (instruction SUBB), et par une réalisation dans l'ordre décroissant des opérations de lecture et écriture (VLOADs - - -1) :

```
-- Boucle d'évaluation et de rangement
      -- VL contient le nombre d'éléments à traiter lors de la première itération
      -- S1 contient le nombre d'itérations
      -- A1 est l'index du dernier élément à traiter dans une itération
LOOP  :
      VLOADs      V1              Filtre1 [A1]          -1
      VADD        V1              #3
      VSTOREs     Filtre2 [A1]    V1                    -1
      SUBB        A1              VL
      MOVE        VL              VRS
      DECR        S1
      JNZ         S1              LOOP
```

En effet, par définition, pour deux filtres *Filtre1* et *Filtre2* tels que *Filtre1* $\nmid$ *Filtre2* et pour tout indice *i* dans les filtres, l'élément du vecteur référencé par *Filtre1*(i) n'est pas utilisé par *Filtre2* dans les indices supérieurs à *i*; la valeur de l'élément chargé dans une boucle via *Filtre1* est donc bien celle de l'élément avant la boucle; le code est correct.

Cette technique de transformation de boucles consistant à traiter les éléments dans l'ordre inverse fut introduite par le compilateur FORTRAN vectorisant de Texas Instrument [Wede75]. Elle est communément nommée *loop reversal* [Wolf91].

Nous avons donc introduit les semi-dépendances qui sont des dépendances entre deux filtres dont la prise en compte dans la génération de code est moins pénalisante que celle des dépendances totales.

3–6.3 Bloc load-store dépendance

Nous présentons maintenant un autre type de dépendances partielles : la dépendance par blocs. Nous introduisons la définition de cette dépendance, donnons quelques exemples, et quelques propriétés de ces dépendances. Nous discutons ensuite de la génération de code d'une instruction présentant des dépendances par blocs (instruction bloc load-store dépendante).

La dépendance par blocs

Définition Un filtre *Filtre1* est dit bloc dépendant avec un facteur de blocage β (entier strictement positif) et un facteur de groupe γ (entier strictement positif) d'un filtre *Filtre2* du même vecteur, et nous le notons

$$Filtre1 \nmid Filtre2\ [\beta, \gamma]$$

si $\forall\, i \in [1 : \#Block - 1]$,

$$\{\, |Filtre1\ (j)| \,/\, j \in [(i-1) * \beta + 1 : i * \beta[\ \} \cap \{\, |Filtre2\ (j)| \,/\, j \in [(i+\gamma-1) * \beta + 1 : VLG]\ \} = \emptyset$$

#Block est le nombre de blocs contigus de taille β éléments composant les filtres *Filtre1* et *Filtre2* :

$\#Block = \lceil\, VLG / \beta\, \rceil$.

La figure 3–16 illustre un exemple de dépendance par blocs. Nous considérons les éléments du vecteur référencé par le filtre *Filtre1* par blocs de β éléments. La dépendance par blocs assure que les éléments d'un tel bloc ne seront plus référencés par le filtre *Filtre2* après le $\gamma^{ième}$ bloc suivant.

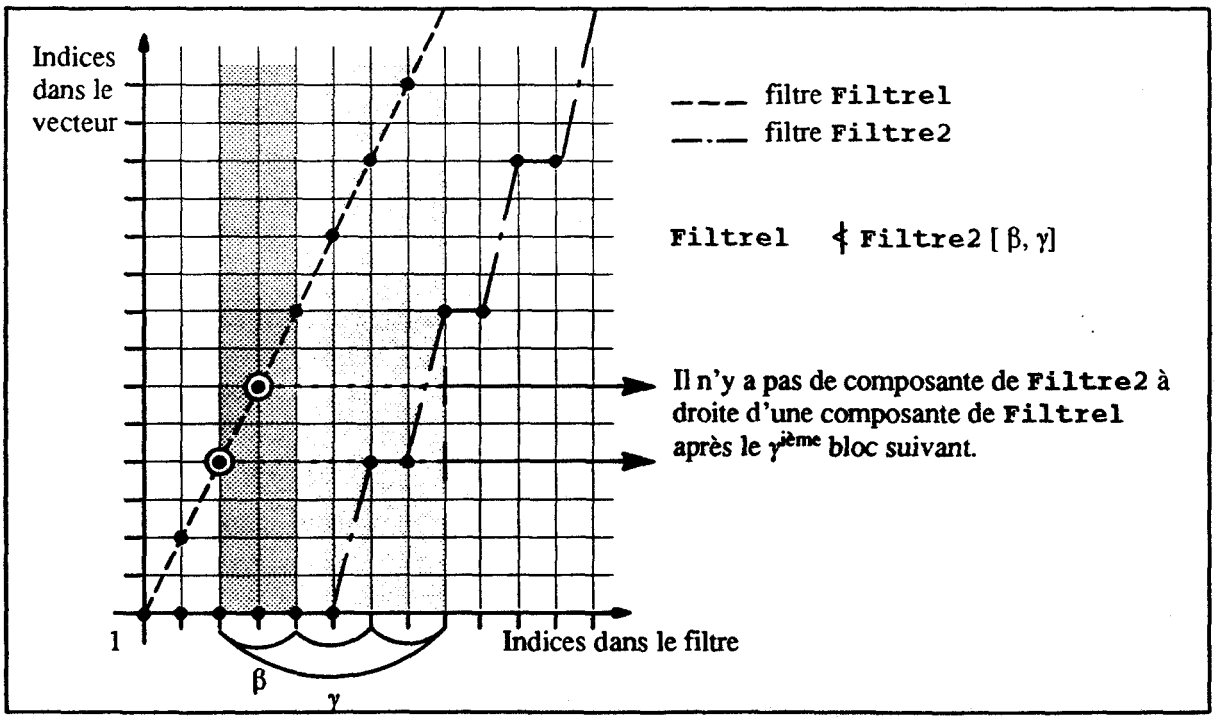


Figure 3–16. Exemple de dépendance par blocs.

Exemples Des exemples de filtres dépendants par blocs ont été donnés dans la première partie de ce chapitre (page 185).

Propriété et remarque Nous précisons maintenant une propriété des dépendances par blocs. La dépendances par blocs entre deux filtres *Filtre1* et *Filtre2* n'est pas unique. En effet, soient deux filtres *Filtre1* et *Filtre2*. Si

Filtre1 $\nleftarrow$ *Filtre2* [β, γ]

alors

Filtre1 $\nleftarrow$ *Filtre2* [$\beta * \gamma$, 2]

d'où la non unicité de la dépendance par blocs

En effet, un élément référencé par *Filtre1* dans le $bg^{ième}$ bloc de taille $\beta*\gamma$ est dans le $b^{ième}$ bloc de taille β avec

$b = \gamma*(bg-1) + i$, $1 \leq i \leq \gamma$.

Au pire ($i = \gamma$),

$b = \gamma*(bg-1) + \gamma$, soit $b = \gamma*bg$.

Cet élément ne sera plus référencé après le $\gamma^{ième}$ bloc de taille β suivant le bloc b, soit le bloc $b+\gamma$ de taille β soit encore le bloc $\gamma*bg+\gamma$ de taille β, c'est-à-dire le bloc $bg+1$ de taille $\beta*\gamma$. D'où la dépendance de facteur de groupe 2.

L'implication réciproque de cette propriété n'est pas vraie.

Il est donc clair que la spécification d'une dépendance par blocs entre deux filtres n'est pas unique. Notons que la génération de code liée à une dépendance par blocs utilise une "zone de registres" de taille $\beta*\gamma$ (voir ci-dessous). La spécification ou la recherche de dépendances par blocs doit donc être exprimée de manière à minimiser ce produit $\beta*\gamma$.

Génération d'instructions bloc load-store dépendantes

Nous détaillons maintenant la génération de code depuis une instruction d'affectation vectorielle bloc load-store dépendante. Une instruction d'affectation vectorielle est dite bloc load-store dépendante si une dépendance par blocs intervient entre sa partie droite et sa partie gauche. Une telle instruction est générée en une boucle unique et ne nécessite pas le passage par une zone temporaire. Nous présentons un exemple de génération et détaillons le cas général à la suite.

Exemple de génération Nous reprenons un exemple introduit précédemment. Soit *Matrix* une matrice de 128 lignes par 65536 colonnes, nous considérons l'instruction de permutation des lignes :

```
REAL Matrix (128, 65536)
INTEGER Index (128)
Matrix (*, *) = Matrix (I, *)
```

Les deux filtres *Matrix* (partie gauche de l'affectation) et *Row_perm* (partie droite de l'affectation) sont liés par une dépendance par blocs de facteur de blocage 128 :

Matrix $\nleftarrow$ *Row_perm* [128, 1].

Supposons les registres vectoriels de taille VRS = 64. Nous avons besoin de $k = 2$ registres. Nous manipulons une boucle unique de 65536 itérations. Une itération traite un bloc de $\beta = 128$ éléments.

```
        -- S1 contient le nombre de boucles (65536)
        -- A1 et V1 traitent les 64 premiers éléments d'une itération
        -- A2 et V2 traitent les 64 derniers éléments d'une itération
        MOVE        VL              #64
        MOVE        A2              A1
        ADD         A2              #64
LOOP    :
        -- 64 premières lectures
        VLOAD       V1              ROW_PERM [A1]
        -- 64 dernières lectures
        VLOAD       V2              ROW_PERM [A2]
        -- 64 premières écritures
        VSTORE      MATRIX [A1]     V1
        ADD         A1              #128
        -- 64 dernières écritures
        VSTORE      MATRIX [A2]     V2
        ADD         A2              #128
        DECR  S1
        JNZ         S1              LOOP
```

Introduction au cas général Nous présentons en premier lieu le traitement sur une instruction telle *Filtre1* = *Filtre2* + *3* avec

$$Filtre1 \not\leftarrow Filtre2\ [\beta, \gamma]$$

Un cas presque général est tout d'abord discuté; la prise en compte du cas général et les nombreux aménagements pour des valeurs extrêmes de β et γ seront précisés ensuite. Nous supposons donc que β est multiple de VRS :

$\beta = k * \text{VRS}$.

Le code produit utilise $\gamma * k$ registres et est basé sur une boucle unique. A chaque itération de boucle, nous manipulons β éléments, c'est-à-dire k registres.

Le principe est d'anticiper le chargement des valeurs des éléments de *Filtre2* avant l'écriture des éléments de *Filtre1* pouvant modifier ces valeurs : on accède ainsi à travers *Filtre2* aux valeurs des éléments du vecteur avant l'instruction, comme il est nécessaire de le faire.

Après une période de démarrage remplissant les $(\gamma - 1) * k$ premiers registres avec une évaluation de la partie droite, chaque itération produit un résultat dans les k derniers registres, et range, dans la partie gauche, les k premiers registres écrits. Ceci est obtenu par une gestion en file des $\gamma * k$ registres. Fonctionnellement, nous décalons les valeurs des registres de k positions à la fin de chaque boucle. (En fait un déroulement de boucle évite ce décalage des registres; nous le présentons après.) :

```
-- Initialisation des n = (γ-1) * k premiers registres
        VLOAD       V1              Filtre2 [A1]
        ADD         A1              VL
        VADD        V1              #3
        .           .               .
        VLOAD       Vn              Filtre2 [A1]
        ADD         A1              VL
        VADD        Vn              #3
LOOP    :
        -- Chargement des k derniers registres
        VLOAD       Vn+1            Filtre2 [A1]
        ADD         A1              VL
        VADD        Vn+1            #3
        .           .               .
        VLOAD       Vn+k            Filtre2 [A1]
        ADD         A1              VL
        VADD        Vn+k            #3
        -- Rangement des k premiers registres
        VSTORE      Filtre1 [A2]    V1
        ADD         A2              VL
        .           .               .
        VSTORE      Filtre1 [A2]    Vk
        ADD         A2              VL
        -- Décalage des registres
        VMOVE       V1              Vk+1
        .           .               .
        VMOVE       Vn              Vk+n
        -- Fin de la boucle
        DECR        S1
        JNZ         S1              LOOP
-- Rangement des n premiers registres
        VSTORE      Filtre1 [A2]    V1
        ADD         A2              VL
        .           .               .
        VSTORE      Filtre1 [A2]    Vn
```

Déroulage de la boucle Le décalage des valeurs dans les registres est inutile. Il nous suffit de dérouler la boucle γ fois et de réordonnancer les instructions au sein de la boucle. On obtient le code :

```
-- Initialisation des n = (γ–1) * k premiers registres
        VLOAD       V1              Filtre2 [A1]
        ADD         A1              VL
        VADD        V1              #3
        .           .               .
        VLOAD       Vn              Filtre2 [A1]
        ADD         A1              VL
        VADD        Vn              #3
LOOP    :
        -- boucle 1 : traite k registres, soit un bloc de β éléments
        VLOAD       Vn+1            Filtre2 [A1]
        ADD         A1              VL
        VADD        Vn+1            #3
        .           .               .
        VLOAD       Vn+k+1          Filtre2 [A1]
        ADD         A1              VL
        VADD        Vn+k+1          #3
        VSTORE      Filtre1 [A2]    V1
        ADD         A2              VL
        .           .               .
        VSTORE      Filtre2 [A2]    Vk+1
        ADD         A2              VL
        -- boucle 2
        VLOAD       Vn+k+2          Filtre2 [A1]
        ADD         A1              VL
        VADD        Vn+k+2          #3
        .           .               .
        VLOAD       Vn+2k+2         Filtre2 [A1]
        ADD         A1              VL
        VADD        Vn+2k+2         #3
        VSTORE      Filtre1 [A2]    Vk+2
        ADD         A2              VL
        .           .               .
        VSTORE      Filtre2 [A2]    V2k+2
        ADD         A2              VL
        -- boucle ...
        .           .               .
        -- boucle γ
        .           .               .
        -- Fin de la boucle
        DECR        S1
        JNZ         S1              LOOP
-- Rangement des n premiers registres
        VSTORE      Filtre1 [A2]    V1
        ADD         A2              VL
        .           .               .
        VSTORE      Filtre1 [A2]    Vn
```

Cas où β n'est pas multiple de VRS Nous considérons maintenant le cas où β n'est pas un multiple de VRS. Nous avons besoin de γ blocs de k registres, k tel que

$k = \lceil \beta / VRS \rceil$.

Les traitements ont lieu par blocs de k registres. Dans le cas où β est multiple de k, tous les traitements peuvent être réalisés sur une longueur fixe,

$VL = \beta / k$.

Sinon, les $k-1$ premières instructions d'un bloc traitant β éléments sont réalisées sur VL = VRS éléments; ensuite VL est chargé avec une valeur

VL' = β modulo VL,

le traitement des derniers éléments du bloc est réalisé sur cette valeur VL'. VL est rechargé avec son ancienne valeur VLG.

Cas où β < VRS Dans la cas où β < VRS, nous groupons plusieurs blocs de taille β dans un registre vectoriel. Soit l le nombre de blocs de taille β alloués dans un registre, nous avons

$l = \lfloor VRS / \beta \rfloor$.

Nous chargeons le registre VL avec $l * \beta$. Nous avons besoin de k registres de longueur VL tel que

$k = \lceil \gamma / l \rceil$.

Instructions *Filtre2 = Filtre1 + ...* Les opérations telles que *Filtre2 = Filtre1 + ...* sont générées sur le même modèle. Toutefois, les blocs de registres sont traités dans l'ordre inverse, du dernier au premier selon la technique de "loop reversal" et les traitements sur ces blocs sont eux aussi effectués du dernier élément au premier.

3–6.4 Détection des dépendances partielles

Nous donnons dans cette sous-section les possibilités de caractérisation des dépendances partielles et rattachons celles-ci des travaux classiques de test de dépendance.

Une dépendances entre deux filtres de vecteur peut être caractérisée par toutes les dépendances entre les éléments des filtres. Une dépendance entre deux éléments de deux filtres $F1$ et $F2$ d'un même vecteur V est caractérisée par un couple d'entiers $i1$, $i2$ tel que

$|F1\ (i1)| = |F2\ (i2)|$

C'est-à-dire que les deux éléments de filtres $F1\ (i1)$ et $F2\ (i2)$ référencent le même élément du vecteur V.

Deux informations sont associées à une telle dépendance entre éléments : le *sens* de la dépendance et la *distance* de la dépendance. Considérons une dépendance entre deux filtres $F1$ et $F2$ caractérisée par le couple d'entiers $i1$, $i2$, le sens de cette dépendance est le signe de la différence $i2 - i1$. La distance de cette dépendance est la valeur de cette différence $i2 - i1$.

Nous pouvons définir certaines des dépendances partielles à l'aide de ces deux notions de sens et de distance des dépendances entre les éléments de filtre.

Deux filtres $F1$ et $F2$ sont semi-dépendants si et seulement si toutes les dépendances entre éléments des filtres $F1$ et $F2$ sont de même sens :

$$\text{si } \forall\ i1, i2 \,/\, |F1\ (i1)| = |F2\ (i2)| \text{ alors } i2 - i1 < 0$$
$$\Leftrightarrow \quad F1 \nleftarrow F2$$

Deux filtres *F1* et *F2* sont dépendants avec un facteur de blocage β=1 et un facteur de groupe γ si et seulement si toutes les dépendances entre éléments sont de distance bornée par γ :

$$\text{si } \forall\, i1, i2 \,/\, |F1\,(i1)| = |F2\,(i2)| \text{ alors } i2 - i1 < \gamma$$
$$\Leftrightarrow \quad F1 \nleq F2\,[1, \gamma]$$

Remarquons que cette détection des dépendances partielles n'est pas complète. Le cas général de dépendance par bloc n'est pas caractérisé simplement à l'aide des notions de sens et distance de dépendance entre éléments de filtres.

3–6.5 Résultats expérimentaux

Nous présentons ici les performances obtenues par la spécification des dépendances partielles afin d'évaluer la validité de leur définition. Nous avons implanté cette génération de code pour un Cray Y-MP/432 et effectué quelques séries de tests.

Pour chacune des trois dépendances partielles : semi-dépendance, dépendance de blocs avec facteur de groupe γ=1 ou $\gamma \neq 1$non, nous comparons deux codes :

- le code produit par le compilateur FORTRAN Cray `cf77` en utilisant la notation explicite du FORTRAN CFT77, ce code est produit par le compilateur en activant toutes les optimisations possibles,
- le code CAL (Cray Assembly Language) produit à la main comme selon les techniques présentées dans les sections précédentes.

Les deux codes sont produit pour un processeur unique. Les résultats ont été obtenus sous une charge moyenne du Cray. Les valeurs de temps présentées ici sont des moyennes d'un grand nombre de valeurs, chaque valeur étant obtenue depuis cinq exécutions de chaque code. Les proportions entre les deux codes étant quasiment constantes, elles peuvent être considérées comme significatives. Les mesures ont été obtenues par la fonction `second ()` et des comparaisons réalisées à l'aide de **hpm** (Hardware Performance Monitor). `second ()` retourne le temps CPU écoulé depuis le début d'un programme. Le *Hardware Performance Monitor* est un système d'analyse post-mortem de l'exécution d'un programme. Il fournit des statistiques sur l'utilisation des unités fonctionnelles, de la mémoire, les temps d'attente, etc.

La comparaison se fait à deux niveaux. Nous pouvons comparer la vitesse d'exécution des codes et la taille mémoire utilisée. Pour chacun des trois types de dépendances partielles comparées, nous présentons les deux codes testés, les résultats de l'exécution et les discutons.

Semi-dépendance

Un exemple de semi-dépendance est obtenu par un décalage des colonnes d'une matrice. Nous l'avons codé de la manière suivante en CFT77 :

```
A (1:15, 2:10000) = A (1:15, 1:9999)
```

Cette instruction est dans une fonction. Le tableau *A* est passé en paramètre et déclaré de taille 15 par 10000. Les ratios mesurés pour ce code sont résumés dans la figure 3–17. *CFT* réfère au temps d'exécution

du code produit par le compilateur FORTRAN. *LSD* réfère à celui produit à la main selon les techniques présentées. *LSD-d2* est le temps d'exécution de ce même programme déroulé deux fois.

CFT/LSD	1,95
CFT/LSD-d2	1,96

Figure 3–17.

Le code produit selon notre technique est exécuté deux fois plus rapidement que le code produit par le compilateur FORTRAN. De plus, nous n'utilisons pas de mémoire supplémentaire, alors que le code produit par FORTRAN nécessite une zone temporaire de 15*9999 éléments.

CFT génère deux boucles successives traitant 15*9999 éléments. La première boucle range la partie droite de l'affectation dans une zone temporaire; la seconde boucle transfère cette zone temporaire dans le tableau.

Notre code consiste en une boucle unique. Les éléments sont traités en ordre inverse. Chaque itération traite VRS=64 éléments. Cette boucle comportant peu d'instructions et n'utilisant qu'un unique registre vectoriel, nous pouvons la dérouler. Nous obtenons alors un ratio de 1.96. Ce résultat constant est dû à la saturation des opérations de chargement/déchargement mémoire déjà présente dans la première boucle et que le déroulage ne peut éviter. En déroulant quatre fois la boucle, le résultat reste identique.

Dépendance de blocs (cas γ=1)

Nous avons codé une permutation de ligne d'une matrice. En CFT77, on écrit

```
A = A (INDEX, :)
```

avec *A* un tableau de taille *N* sur 10000 et *INDEX* un tableau de *N* valeurs entières. Il n'y a pas de dépendance entre *A* et *INDEX*. La dépendance par bloc est caractérisée par les valeurs β=*N*, et γ=1. Nous avons fait quatre tests pour des valeurs différentes (mais toujours connues à la compilation) de *N*. *INDEX* est initialisé de manière à minimiser les conflits d'accès à la mémoire lors de l'accès à

```
A (INDEX, :)
```

Les résultats ne reflètent ainsi que les effets de notre technique de génération de code.

N	15	32	64	128
CFT/LSD	1,93	1,45	1,30	1,45
CFT/LSD-d2	2,73	1,63	1,57	1,60

Figure 3–18.

Le compilateur CFT génère deux boucles vectorielles. Ces deux boucles sont incluses dans une boucle scalaire traitant une colonne à chaque itération. La première boucle vectorielle charge les éléments dans une zone temporaire (en mémoire). Une fois les accès mémoire de cette première boucle terminés, la seconde boucle est déclenchée (les deux boucles sont séparées par une instruction CAL **CMR**). Cette seconde boucle transfère la zone mémoire temporaire dans le tableau.

Le code produit par l'application des techniques de dépendances par blocs utilise des registres vectoriels pour mémoriser un bloc. Une boucle unique charge les éléments du tableau dans les registres et les range ensuite en mémoire. Avant le déclenchement de ce rangement, une instruction **CMR** assure la prise en compte de la possible dépendance entre les deux opérations.

Note. Pour des petites valeurs de *N* (inférieures au nombre de registres vectoriels disponibles), une optimisation est mise en place par CFT. Le code généré est largement différent de celui expliqué ici. L'opération est réalisée par tranches de VRS=64 colonnes. Le traitement d'une tranche de colonne charge la ligne *INDEX (1)* dans un premier registre vectoriel, la ligne *INDEX (2)* dans un second registre, etc. Les registres sont ensuite rangés dans le tableau. Les performances obtenues sont alors bien meilleures. Ce fait n'intervient en rien dans la comparaison CFT/LSD, un compilateur utilisant les techniques des dépendances partielles pouvant fort bien reprendre cette optimisation de CFT.

Dépendance de blocs (cas $\gamma \neq 1$)

Le code CFT suivant comporte une dépendance par bloc avec $\gamma \neq 1$:

```
A (:, 2:10000) = A (INDEX, 1:9999)
```

Nous avons une combinaison des deux codes précédents. Les colonnes de *A* sont décalées alors que, dans le même temps, les éléments de chaque colonne sont permutés. Le tableau *A* comporte 15*10000 réels. La dépendance par bloc est caractérisée par $\beta=15$ et $\gamma=2$.

CFT/LSD	1,36
CFT/LSD-d2	1,38

Figure 3–19.

CFT utilise un tableau temporaire de 15*9999 réels. Le code produit selon notre technique n'utilise pas de zone temporaire en mémoire, mais nécessite un registre supplémentaire.

CFT produit deux boucles. La première boucle range les valeurs du tableau dans la zone temporaire. La seconde boucle transfère cette zone dans le tableau initial.

Notre code est composé d'une boucle unique. Une itération traite un bloc, soit 15 éléments. Ces 15 éléments sont chargés depuis la partie droite. Une instruction CAL **CMR** assure la terminaison de cette lecture. Le rangement dans le tableau des éléments chargés à l'itération précédente est alors réalisé. Nous utilisons la technique présentée de déroulage et re-ordonnancement de la boucle qui évite tout décalage des registres vectoriels (se référer page 213).

Les performances présentées attestent sans conteste l'intérêt des load-store dépendances partielles. En aucun cas, la génération de code suivant la technique présentée ici ne produit de performances inférieures à celles du compilateur FORTRAN CFT. De plus, l'espace mémoire utilisé, aussi bien pour les données que pour le code, est moindre dans tous les cas. Le gain obtenu sur Cray s'explique par le fait que nous réduisons les accès à la mémoire et que nous pouvons dérouler les boucles d'instructions. Ce déroulage est possible car le code produit est plus court que celui produit par FORTRAN. On évite ainsi des problèmes de rechargement du *buffer* d'instructions. Le déroulage des boucles autorise un parallélisme plus important, bien que ce fait n'ait pas été sensible ici. En effet, les instructions testées sont uniquement constituées de chargement/déchargement mémoire. Une affectation dont la partie droite demande un fort temps de calcul bénéficierait d'un parallélisme plus important lors du déroulage de boucle.

3–7 Génération d'instructions partiellement dépendantes

Nous avons présenté les dépendances partielles et la génération d'instructions load-store partiellement dépendantes dans les sous-sections précédentes. Nous discutons maintenant de la génération d'une séquence d'instructions vectorielles pouvant être liées par des dépendances partielles, généralisant ainsi la génération des SIVs comportant des dépendances totales et la génération des instructions load-store partiellement dépendantes.

Nous présentons tout d'abord les différentes dépendances que nous pouvons rencontrer lors de la combinaison des dépendances vectorielles et des dépendances partielles. Ces combinaisons sont présentées à la figure 3–20. Les dépendances de nom sont forcément totales.

		Les trois types de dépendances entre deux filtres		
		Dépendance totale	Semi dépendance	Bloc dépendance
Dépendances vectorielles.	Load-store λ	$\delta\lambda$	$\delta\lambda_s$	$\delta\lambda$ [β, γ]
	Hétéronymes η	$\delta\eta$	$\delta\eta_s$	$\delta\eta$ [β, γ]
	Dép. de nom ν	$\delta\nu$	—	—

Figure 3–20. Combinaison des dépendances vectorielles et des dépendances partielles.

Les deux dépendances non encore traitées sont donc les semi-dépendances hétéronymes et les dépendances hétéronymes par blocs. Ces dépendances sont aussi caractérisées par

- le type de la dépendance : dépendance de flot, anti-dépendance ou dépendance de sortie.

- le sens de la dépendance entre les deux filtres. Soient deux instructions *S1*, référençant le filtre *Filtre1*, et *S2*, référençant le filtre *Filtre2*, telles que
  $S1 \ominus S2$.
  La dépendance entre les deux filtres est dite positive, et notée $S1\ \delta^{+}\ S2$, si
  Filtre1 δ *Filtre2* ,
  elle est dite négative, et notée $S1\ \delta^{-}\ S2$, si
  Filtre2 δ *Filtre1*.

Semi-dépendance hétéronyme

Nous discutons ici de la génération de code liée à une semi-dépendance hétéronyme. Nous considérons en premier lieu le cas d'une dépendance négative. Nous traitons ensuite celui d'une dépendance positive. Les deux discussions sont basées sur les instructions *S1* et *S2* de la même SIV

```
S1:   ... Filtre1 ...
...
S2:   ... Filtre2 ...
```

telles que $S1\ \delta\eta_s\ S2$.

Dépendance négative Nous sommes dans le cas où *Filtre2* $\not\in$ *Filtre1*. La prise en compte de cette dépendance n'oblige en rien à modifier l'algorithme de génération de SIV défini à la section NO TAG. L'utilisation d'un élément du vecteur par le filtre *Filtre2* est réalisée après son éventuelle utilisation par *Filtre1* et cet élément ne sera ensuite plus utilisé par aucun des filtres. L'ordre de traitement des éléments est donc conservé.

Dépendance positive Nous sommes dans le cas où *Filtre1* $\not\in$ *Filtre2*. La prise en compte de cette dépendance est réalisée par le traitement en arrière des tranches de vecteurs du BIV. Notons cependant que cette transformation n'est possible que si le BIV ne comporte pas déjà une dépendance $\delta\eta_s^{-}$ dont l'exécution en arrière serait incorrecte.

La génération d'une instruction liée par une semi-dépendance hétéronyme marque donc le BIV du signe de la dépendance (un BIV n'est marqué d'aucun signe s'il ne contient pas de telles dépendances). Lors de la génération d'un BIV, la rencontre d'une semi-dépendance négative ou positive rompt le BIV si le signe du BIV n'est pas identique à celui de la dépendance. Lors de la terminaison de la génération du BIV, son signe détermine le sens nécessaire du traitement des éléments (signe négatif traitement en avant; signe positif traitement en arrière).

Dépendance hétéronyme par blocs

Nous présentons ici les conséquences des dépendances hétéronymes par blocs sur la génération de code. De même que pour les semi-dépendances par blocs, nous associons un signe à un BIV lors de la génération de deux instructions liées par une dépendance par blocs. Le sens de traitement des éléments de vecteur dans le BIV est également déterminé lors de la terminaison de sa génération et en fonction du signe qui lui est associé. De même, la rencontre d'une instruction hétéronyme dépendante avec une instruction antérieure du BIV dont le signe ne correspond pas avec celui du BIV oblige la terminaison de ce BIV.

De même que pour une load-store dépendance par blocs, la génération des deux instructions liées par une dépendance hétéronyme par blocs utilise une zone de registres pour anticiper les lectures via un filtre afin d'autoriser les écritures via le second filtre.

La génération d'une SIV ne possédant qu'une dépendance $\delta\eta$ [β, γ] est donc bâtie sur le modèle de la génération d'une instruction load-store dépendante par blocs. Nous regardons maintenant le cas de multiples dépendances $\delta\eta$ [β, γ] au sein d'une même SIV.

Considérons une SIV incluant plusieurs dépendances hétéronymes par blocs de mêmes caractéristiques : même facteur de groupe γ et même sens de dépendance; les facteurs de blocage β sont éventuellement différents. La génération en un unique BIV est calquée sur celle d'une SIV comportant une unique dépendance. Le BIV est déroulé γ fois. Pour les instructions liées par une dépendance $\delta\eta$ [β, γ], ce déroulage est effectué de manière identique à celui du code produit par une load-store dépendance par blocs; il évite le décalage des registres. Pour les autres instructions, ce déroulage consiste en γ recopies du code non déroulé; à l'inverse d'un déroulage "réel", on utilise γ fois les mêmes registres.

Dans le cas le plus général, deux dépendances $\delta\eta$ [β, γ] de facteur de bloc différents γ_1 et γ_2 peuvent être générées dans une même boucle. Il est alors nécessaire de dérouler la boucle $\gamma = ppmc(\gamma_1, \gamma_2)$ fois. Une instruction dépendante par bloc de facteur de groupe γ_i est déroulée γ_i fois puis ce code produit est dupliqué γ/γ_i fois. On obtient un déroulage total de facteur γ.

3–8 Conclusion et discussion

Habituellement, la dépendance entre deux accès aux éléments d'un même vecteur est une information binaire : il y a dépendance ou il n'y a pas dépendance. Une dépendance entraîne alors une rupture entre le traitement des deux accès lors de l'exécution. De plus, si la dépendance survient entre les parties droite et gauche d'une affectation, le résultat de l'évaluation de la partie droite doit être sauvegardé dans une zone temporaire avant de pouvoir être affecté à la partie gauche.

Nous avons caractérisé des dépendances partielles dont les conséquences sont moindres. Nous évitons la rupture de l'exécution entre les deux accès. Le passage par une zone temporaire en mémoire est remplacé par l'utilisation de quelques registres vectoriels. Les implantations réalisées sur Cray Y-MP ont montré des gains de performances non négligeables.

Nous discutons maintenant d'autres points relatifs à ces dépendances et introduisons quelques précisions.

1) Dans la présentation précédente, nous utilisons des registres vectoriels stockant les lectures anticipées d'éléments de vecteurs. Pour un nombre d'éléments important à charger, il peut arriver que le nombre de registres disponibles ne soit pas suffisant. Sur une machine disposant d'une mémoire cache (tel le Cray 2), il est alors intéressant d'utiliser cette mémoire cache au lieu des registres pour ranger les blocs chargés par anticipation.

2) Les dépendances partielles caractérisées ici sont intégrées dans les spécifications des vecteurs LSD2. Le programmeur LSD2 exprime la non dépendance, le type de dépendances partielles ou la dépendance totale entre deux filtres de vecteurs au sein de la déclaration du vecteur. Les autres informations nécessaires à la délimitation des séquences d'instructions vectorielles (spécification de longueur) par exemple sont elles aussi exprimées par le programmeur en LSD2. L'implantation des techniques présentées ici est donc immédiate depuis LSD2.

3) L'application des techniques de génération de code présentées ici n'est pas limitée aux machines pipelines vectorielles. Leur prise en compte sur les machines massivement parallèles disposant de registres

(par exemple MasPar MP-1) est immédiate. Cependant, pour une instruction load-store dépendante, les gains de performances sont directement liés aux ratios entre les instructions de chargement et déchargement mémoire et les autres opérations nécessaires à la réalisation de l'instruction. En effet, le gain est uniquement réalisé par le non-passage par une zone temporaire mémoire et donc quelques non-accès mémoire. Sur les machines actuelles, le déclenchement des instructions ne peut avoir lieu en parallèle; on ne peut donc pas, comme sur le Cray, améliorer l'exécution parallèle du code obtenu. (Notons cependant que les accès mémoire peuvent être effectués en parallèle avec d'autres opérations sur la MasPar.)

4) Notre étude est donc basée sur le langage LSD2. Cependant, les résultats peuvent être utilisés pour tous les langages utilisant une syntaxe explicite de manipulation de vecteurs. Les informations utiles à l'utilisation des techniques présentées ici doivent être exprimées par le programmeur ou retrouvées automatiquement par une phase préliminaire du compilateur. Des travaux sur la possibilité de découverte automatique des dépendances partielles sont nécessaires dans cette seconde hypothèse.

Conclusion

Les travaux présentés dans ce document sont relatifs à la proposition d'outils pour l'expression des algorithmes à parallélisme de données et à l'implantation de ces outils sur les machines à parallélisme de données.

Une première partie a démontré l'intérêt de travaux dans cette voie par une présentation des architectures à parallélisme de données, et un tour d'horizon des propositions de langages dans ce domaine.

Nous avons montré que le regroupement de plusieurs types d'architectures (pipelines vectorielles et massivement parallèles) sous le vocable unique de machines à parallélisme de données est justifié par nombre de points communs. Ces points communs nous ont autorisés à considérer cette classe de machines en tant que telle et à proposer des outils uniques pour la programmation et la génération de code sur ces machines.

De nombreux langages explicites ont été proposés pour la programmation de ces machines. Le terme explicite recouvre la manipulation effective des structures de données traitées par les algorithmes et l'expression d'informations sur ces données pouvant être prises en compte lors de la génération d'un code performant. Bien des points étant encore sans réponse (opération de description, manipulation de "sous-vecteurs", longueur de traitement des opérandes vecteurs, etc.), cette approche pouvait être poursuivie.

Nous avons ensuite présenté deux phases de nos travaux. Chacune est axée autour d'un langage explicite à parallélisme de données. Le langage EVA implante une structure de données adaptée à la programmation à parallélisme de données, le vecteur EVA. L'importance des opérations de descriptions, dans les langages vectoriels habituels et les problèmes d'algorithmiques vectorielles, est liée à la nécessité d'identifier des sous-vecteurs au sein des vecteurs. Le vecteur EVA inclut, par construction, les opérations de description. Un vecteur EVA est un "sous-vecteur", en ce sens qu'il autorise l'accès à certains éléments d'un autre vecteur. EVA manipule ainsi des "sous-vecteurs". D'autres aspects contribuent à la définition de EVA comme un langage explicite. Nous avons défini des moyens simples d'expliciter ou de déterminer les longueurs sur lesquelles traiter les vecteurs. Nous avons proposé différents opérateurs d'affectation vectorielle et séparateurs entre instructions explicitant les dépendances entre les accès aux éléments de vecteurs. Nous avons introduit une structure de "bloc de description", extension du `WHERE` traditionnel.

L'ensemble de ces constructions autorisent une génération de code simple depuis EVA. Celle-ci est réalisée via DEVIL, un langage intermédiaire vectoriel manipulant directement les vecteurs EVA. DEVIL inclut aussi des notions de blocs vectoriels et parallèles et de vecteurs temporaires, implantations de bas niveau des constructions EVA. Ces notions sont à la base d'une génération de code simple et performante. Nous avons présenté des éléments de performances obtenues sur Cray Y-MP.

Quelques points sensibles de EVA/DEVIL, notamment en ce qui concerne le passage de paramètre (aucune information ne peut être connue sur la forme riche des vecteurs EVA lors d'un passage de vecteur en paramètre) et la prise en compte de toutes les architectures à parallélisme de données (rien, en EVA, n'est spécifique aux machines massivement parallèles), nous ont naturellement amenés à proposer une nouvelle génération de langage autour de LSD2. LSD2 vise clairement les deux types d'architectures à parallélisme de données. Les vecteurs LSD2 sont associés à un typage fort assurant la qualité de la génération de code lors de passage en paramètre. Le vecteur LSD2 est défini comme la réunion des "sous-vecteurs" se partageant une zone d'allocation, c'est-à-dire référencant des éléments communs. Au sein du vecteur sont spécifiées des informations sur des liens entre des groupes d'éléments et l'utilisation des diverses dimensions du vecteurs reflétant les manipulations réalisées sur ce vecteur. A partir de ces informations, l'allocation produite pour le vecteur sur une machine massivement parallèle diminuera sensiblement les coûts des communications entre les processeurs. D'autres spécifications reflètent les dépendances entre les différents accès aux éléments du vecteur. Différentes classes de dépendances partielles ont été introduites. Leurs conséquences sur les performances du code généré ont été justifiées par la présentation de tests sur Cray Y-MP.

Différentes voies de poursuite de nos travaux sont possibles. Elles s'articulent principalement autour de LSD2, le prolongement de EVA ayant déjà été réalisé au sein de LSD2. Clairement, l'implantation effective des langages machines virtuelles LIMP (pour machines massivement parallèles) et LIVP (pour machines pipelines vectorielles), autorisant ainsi une réelle portabilité de LSD2 sur les machines à parallélisme de données, est une priorité.

Les travaux en cours autour de LIMP concernent la prise en compte des spécifications de distribution et alignement exprimées en LSD2 lors de l'allocation des vecteurs et lors de la manipulation de ceux-ci. Dans une première phase, nous nous limitons à l'expression de spécifications statiques, c'est-à-dire qui peuvent être déterminées lors de la compilation. La prise en compte de spécifications dynamiques et l'implantation effective de cette fonctionnalité autorisera seule une évaluation précise des éventuels sur-coûts.

L'expérience acquise lors du développement de DEVIL nous autorise à nous concentrer sur des techniques plus avancées de génération de code prenant en compte l'aspect multi-processeur des machines pipelines vectorielles actuelles. Il sera éventuellement possible de faire remonter en LSD2 des constructions aidant une telle génération de code. Cependant, il apparaît dès maintenant qu'un point essentiel est la définition d'une notion de machine vectorielle multi-processeur virtuelle. En effet, selon notre approche, l'expression, dans les différents niveaux de langages proposés sous LSD2, de quelque information que ce soit, doit se faire indépendamment des caractéristiques de la machine cible.

D'ores et déjà, nous entrevoyons la prise en compte des spécifications actuelles de distribution et de dépendances partielles par blocs sur ces machines vectorielles multi-processeurs. En effet, la distribution spécifie des regroupements d'éléments de vecteurs liés par des interactions fortes. Le traitement d'un groupe donné gagnerait certainement à être réalisé sur un même processeur vectoriel. De même les dépendances partielles par blocs identifient des groupes d'éléments liés par des dépendances, le traitement d'un bloc devrait être confié à un processeur unique.

La disposition effective d'un langage intermédiaire à parallélisme de données (LOLLA) sur des machines massivement parallèles et pipelines vectorielles est une plate-forme intéressante pour la compilation de langages tel FORTRAN 90. Etant entendu que les informations explicitées en LSD2 ne peuvent pas l'être en FORTRAN 90, la mise en place d'une phase préliminaire, proche des actuels travaux de vectorisation automatique, est nécessaire. En particulier, la reconnaissance automatique des dépendances partielles introduites par LSD2 amènerait des gains de performances appréciables.

Ces travaux de reconnaissance automatique seraient d'autant plus intéressants à développer que la prise en compte des dépendances partielles serait généralisée. L'extension des algorithmes de génération de code aux cas non traités ici tels la présence de multiples dépendances partielles entre parties droite et gauche d'une affectation ou entre deux instructions, est la source de travaux déjà entrepris.

Une autre voie pouvant être abordée, et dans laquelle de nombreux travaux existent déjà, concerne l'implantation d'un langage à parallélisme de données tel LSD2 sur une machine à parallélisme de tâches. Cette approche de la programmation des machines à parallélisme de tâches nous semble en effet une condition indispensable à l'acceptation de ces machines par la communauté scientifique. La génération de code devrait pouvoir prendre en compte les informations de distribution et d'alignement des éléments de vecteurs présentes en LSD2, pour répartir les données et les traitements à réaliser sur les processeurs. De plus, comme dans le cas des machines vectorielles multi-processeurs, l'utilisation des informations de dépendances par blocs pourrait amener des résultats intéressants.

Références bibliographiques

Les références sont présentées dans l'ordre alphabétique des renvois. Cet ordre est différent de l'ordre alphabétique sur les auteurs. Un renvoi est constitué de 3 ou 4 lettres correspondant aux initiales des auteurs, deux chiffres indiquant l'année de publication, et d'une éventuelle lettre minuscule différenciant les renvois des entrées dont les caractères précédents sont identiques. Les initiales des auteurs représentent : Les 4 premières lettres du nom pour un auteur unique, les deux premières lettres des noms pour deux auteurs, les initiales des noms pour 3 ou 4 auteurs, les initiales des noms des trois premiers auteurs suivies d'un signe plus (+) pour plus de 4 auteurs.

[ALS91] Eugène ALBERT, Joan D. LUKAS, et Guy L. STEELE Jr. Data parallel computers and the FORALL statement. *Journal of Parallel and Distributed Computing* **13**(2):185–192, Octobre 1991.

[ANSI91] ANSI. *FORTRAN 90.* X3J3 Draft S8.118, submitted as text for ANSI X3.198–1991, Mai 1991.

[Arya85] Siamak ARYA. An optimal instruction-scheduling model for a class of vector processors. *IEEE Transactions on Computers* **C-34**(11):981–995, Novembre 1985.

[ASU86] Alfred AHO, Ravi SETHI, et Jeffrey ULLMAN. *Compilateurs : principes, techniques et outils.* InterÉditions, 1989 dans la trad. française de l'édition originale (Addison-Wesley, 1986).

[Babb89] Robert G. BABB II. SARA: a Cray assembly language speedup tool. *Proc. of the NATO Advanced Research Workshop on Supercomputing*, Juin 19–23, 1989, Trondheim, Norvège.

[BaKn75] V. R. BASILI et J. C. KNIGHT. A language design for vector machines. *ACM Sigplan Notices* **10**(3):39–43, Mars 1975.

[Bann88a] Uptal BANERJEE. An introduction to a formal theory of dependance analysis. *The Journal of Supercomputing* **2**(3):133–149, Novembre 1988.

[Bann88b] Uptal BANERJEE. *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers (Boston), 1988.

[Batc80] Kenneth E. BATCHER. Design of a massively parallel processor. *IEEE Transactions on Computers* **C-29**(9):836–840, Septembre 1980.

[BBES91] Ingo BARTH, Thomas BRÄUNL, Stefan ENGELHARDT, et Frank SEMBACH. PARALLAXIS version 2 user manual. Rapport **2/91**, Computer Science Dept., Université de Stuttgart, Allemagne, Février 1991.

[BBK+68] George H. BARNES, Richard M. BROWN, Maso KATO, David J. KUCK, Daniel L. SLOTNICK, et Richard A. STOKES. The Illiac IV computer. *IEEE Transactions on Computers* **C-17**(8):746–757, Août 1968.

[BDHR90] Donald W. BLEVINS, Edward W. DAVIS, Robert A. HEATON, et Johan H. REIF. BLITZEN: a highly integrated massively parallel machine. *Journal of Parallel and Distributed Computing* **8**(2):150–160, 1990.

[Blan90] Tom BLANK. The MasPar MP-1 architecture. *Proc. of the IEEE Compcon Spring 1990*, San Francisco, CA, Février 3–5, 1990, IEEE Society Press, Février 1990, pages 20–24.

[BoLe91] Luc BOUGÉ et Jean-Luc LEVAIRE. Control structures for data-parallel SIMD languages: semantics and implementation. Rapport de Recherches 91–06, Laboratoire d'Informatique du Parallélisme, ENS Lyon, Lyon, 1991.

[BoMe81] Alain BOSSAVIT et Bertrand MEYER. The design of vector programs. *Proc. of the Int'l Conf. on Algorithmic Languages*, North–Holland, Amsterdam, 1981, pages 99–114.

[Boug91] Luc BOUGÉ. On the semantics of languages for massively parallel SIMD architectures. *Proc. of the Conf. on Parallel Architecture and Languages Europe (PARLE'91)*, Eindhoven, The Netherlands, June 10–13, 1991.

[Bräu89] Thomas BRÄUNL. Structured SIMD programming in PARALLAXIS. *Structured Programming* **10**(3):121–132, Juillet 1989.

[Cala89] David A. CALAHAN. Some results in memory conflicts analysis. *Proc. Supercomputing'89*, Reno (Nevada), Novembre 1989, pages 775–778.

[CDC83] Control Data Corporation. *FORTRAN 200 Version 1, Reference Manual*. Publication 60480200, Control Data Corp. (St. Paul, MN), Novembre 1983. révision G, Avril 1986.

[CDC86] Control Dat Corporation. *CDC Cyber 200 VECTORC Reference*. Pub. 60000018, 1986.

[CDC87] Control Data Corporation. *CDC Cyber 200 Model 205 Computer System — Hardware Reference Manual*. Publication 60256020 revision F, Control Data Corp. (St. Paul, MN), Octobre 1987.

[Chri90] Peter CHRISTY. Software to support massively parallel computing on the MasPar MP-1. *Proc. of the IEEE Compcon Spring 1990*, San Francisco, CA, Février 3–5, 1990, IEEE Society Press, Février 1990, pages 29–33.

[Chri91] Peter CHRISTY. Virtual processors considered harmful. *Proc. of the Sixth Distributed Memory Computing Conference*, Portland, OR, Avril 28–Mai 2, 1991.

[CPHS83] M. CLINT, R. PERROTT, C. HOLT, and A. STEWART. The influence of hardware and software considerations on the design of synchronous parallel algorithms. *Software—Practice and Experience* **13**:961–974, 1983.

[Cray83] Cray Research Inc. (by Peter J. SYDOW). *Optimization Guide*. Publication SN–0220 revision B, Cray Research Inc. (Mendota Heights, MN), Décembre 1983.

[Cray87] Cray Research Inc. *Cray X-MP Computer Systems Functionnal Description Manual*. Publication HR-3005, Cray Research Inc. (Mendota Heights, MN), Février 1987.

[Cray88a] Cray Research Inc. *Cray Y-MP Computer Systems Functionnal Description Manual*. Publication HR-4001, Cray Research Inc. (Mendota Heights, MN), Janvier 1988.

[Cray88b] Cray Research Inc. *Symbolic Machine Instruction Reference Manual*. Publication SR-0085 revision B, Cray Research Inc. (Mendota Heights, MN), Mars 1988.

[Cray88c] Cray Research Inc. *CAL Assembler Version 2 Reference Manual*. Publication SR-2003 revision C, Cray Research Inc. (Mendota Heights, MN), Juin 1988.

[Davi69] Robert L. DAVIS. The Illiac IV processing element. *IEEE Transactions on Computers* **C-18**(9):800–816, Septembre 1969.

[Deke90] Jean-Luc DEKEYSER. *Algorithmes, Langages et Architectures Vectoriels — Le Projet WEST*. Mémoire d'habilitation à diriger des recherches, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, Décembre 1990.

[DKV88] David C. DOUGLAS, Brewester A. KAHLE, et Alex'VASILEVSKY. The architecture of the CM-2 data processor. Technical Report **HA88–1** ou **TMC–91**, Thinking Machine Corp. (Cambridge, MA), 1988.

[DMP90a] Jean-Luc DEKEYSER, Philippe MARQUET, and Philippe PREUX. EVA—An explicit language. An alternative language to FORTRAN 90. *ACM Sigplan Notices* **25**(8):53–71, Août 1990.

[DMP90b] Jean-Luc DEKEYSER, Philippe MARQUET, and Philippe PREUX. DEVIL: An intermediate vector language— Definition and implementation. *Proc. Int'l Workshop on Compilers for Parallel Computers*, Paris, Décembre 1990, pages 273–284.

[DMP91a] Jean-Luc DEKEYSER, Philippe MARQUET, and Philippe PREUX. A multi-level environment for data parallel code generation. *European Workshops on Parallel Computing*, Barcelone, Espagne, Mars 1992.

[DMP91b] Jean-Luc DEKEYSER, Philippe MARQUET, and Philippe PREUX. Load-store dependence and data-parallel code generation. (Soumis pour publication.) Rapport **ERA-87**, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, Juin 1991.

[DMP91c] Jean-Luc DEKEYSER, Philippe MARQUET, and Philippe PREUX. A step by step approach to transform FORTRAN code in a vector form using the embedded language LSD2. Rapport **ERA-101**, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, Juillet 1991.

[Dong87] J. J. DONGARRA (ed.). *Experimental Parallel Computing Architectures*. Special Topics in Supercomputing 1, North-Holland, 1987.

[EHJD90] R. EIGENMANN, J. HOEFLINGER, G. JAXON, et D. PADUA. CEDAR FORTRAN and its restructuring compiler. *Rapport CSRD 1041*, Center for Supercomputing Research and Development, Univerity of Illinois at Urbana-Champaign, Setpembre 1990.

[FHK+91] Geoffrey FOX, Seema HIRANANDANI, Ken KENNEDY, Charles KOELBEL, Uli KREMER, Chau-Wen TSENG, and Min-You WU. FORTRAN D Language Specification. Research Report Rice COMP TR90-141, Dept. of Computer Science, Rice University, Dec. 1990, revised Apr. 91.

[Fich85] Françoise FICHEUX-VAPNÉ (sous la direction de). *FORTRAN 77—Guide pour l'Ecriture de Programmes Portables*. Collection de la Direction des Etudes et Recherches d'Electricité de France **60**, Eyrolles (Paris), 1985.

[Giss86] Richard GISSELQUIST. An Experimental C compiler for the Cray 2 computer. *ACM Sigplan Notices* **21**(9):32–41, Septembre 1986.

[GGJ+90] K. GALLIVAN, D. GANNON, W. JALBY, A. MALONY, et H. WIJSHOFF. Experimentally characterizing the behavior of multiprocessor memory system : a case study. *IEEE Trans. on Soft. Eng.* **16**(2):216–223, Février 1990.

[GPHL90] Mark D. GUZZI, David. A. PADUA, Jay HOEFLINGER, and Ducan H. LAWRIE. CEDAR FORTRAN and other vector and parallel FORTRAN dialects. *The Journal of Supercomputing* 3:37–62, 1990.

[Guzz87] Mark D. GUZZI. CEDAR FORTRAN programmer's handbook. *Rapport CSRD 801*, Center for Supercomputing Research and Development, Univerity of Illinois at Urbana-Champaign, Juin 1987.

[Hart77] Edward T. HARTNETT. STAR FORTRAN—An overview of essential characteristics. *ACM Sigplan Notices* **12**(4):57–66, Avril 1977.

[Hext75] J.B. HEXT. Array reference operations. *ACM Sigplan Notices* **10**(3):113–118, Mars 1975.

[Hill85] W. Daniel HILLIS. *The Connection Machine*. The MIT Press (Cambridge, MA), 1985. Traduction française *La Machine à Connexions*. Masson (Paris), 1988.

[HKK+91] Seema HIRANANDANI, Ken KENNEDY, Charles KOELBEL, Ulrich KREMER, and Chau-Wen TSENG. An Overview of the FORTRAN D Programming System. Research Report Rice COMP TR91-154, Dept. of Computer Science, Rice University, March 1991.

[HKMP91] Philippe HOOGVORST, Ronan KERYELL, Philippe MATHERAT, et Nicolas PARIS. POMP or how to design a massively parallel machine with small developments. *Proc. of the Conf. on Parallel Architectures and Languages Europe (PARLE'91)*, Eindhoven, The Netherlands, June 10–13, 1991. Also in Rapport de Recherches, LIENS 91-5, Ecole Normale Supérieure, Paris, Avril 1991.

[HKT91a] Seema HIRANANDANI, Ken KENNEDY, and Chau-Wen TSENG. Compiler support for machine-independent parallel programming in FORTRAN D. Research Report Rice COMP TR91-149, Dept. of Computer Science, Rice University, Jan. 1991.

[HKT91b] Seema HIRANANDANI, Ken KENNEDY, and Chau-Wen TSENG. Compiler optimizations for FORTRAN D on MIMD distributed-memory machines. *Proc Supercomputing'91*, Albuquerque (NM), Novembre 1991, pages 86–100.

[HoJe81] Roger W. HOCKNEY et C. R. JESSHOPE. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger Ltd (Bristol), 1981.

[HoJe88] Roger W. HOCKNEY et C. R. JESSHOPE. *Parallel Computers 2: Architecture, Programming and Algorithms*. Adam Hilger Ltd (Bristol, Philadelphia), 1988.

[Hord90] R. Michael HORD. *Parallel Supercomputing in SIMD Architectures.* CRC Press (Boston), 1990.

[HwBr84] Kai HWANG et Fayé A. BRIGGS. *Computer Architecture and Parallel Processing.* McGraw-Hill (New-York), 1984.

[IBM88] IBM. *IBM Entreprise Systems Architecture/370 and System/370. Vector Operations.* IBM publication SA22-7125-3, Fourth edition, Août 1988.

[Jalb87] William JALBY. Cedar architecture. In [LiSa87], *Supercomputing — State-of-the-Art.* A. LICHNEWSKY et C. SAGUEZ (eds.). Elsevier Science Publishers B. V. (North-Holland), 1987, pages 41–51.

[Jégo87a] Yvon JEGOU. Access patterns: A usefull concept in vector programming. *Proc. Int'l Conf. on Supercomputing*, Athens, Greece, Juin 8–12, 1987. In *Lectures Notes in Computer Sciences* **297**, pages 377–391.

[Jégo87b] Yvon JEGOU. Le langage vectoriel HELLENA. Rapport de recherche **703**, INRIA-Rennes, Juillet 1987.

[KaHi89] Brewster A. KAHLE et W. Daniel HILLIS. The Connection Machine model CM-1 architecture. *IEEE Transactions on Systems, Mans, and Cybernetics* **19**(4):707–713, Juillet/Août 1989.

[KMC72] David J. KUCK, Yoichi MURAOKA, et Shyh-Ching CHEN. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speedup. *IEEE Transactions on Computers* **C-21**(12):1293–1310, Décembre 1972.

[KPSW82] D. KUCK, D. PADUA, A. SAMEH, and M. WOLFE. Languages and high-performances computations. *Proc of the IFIP TC 2 Working Conf. on The Relationship between Numerical Computation and Programming Languages*, Boulder, Colorado, Août 1981, John. K. REID (ed.), North-Holland Pub., 1982, pages 205–221.

[Kuck68] David J. KUCK. The Illiac IV software and application programming. *IEEE Transactions on Computers* **C-17**(8):758–770, Août 1968.

[Kuck78] David J. KUCK. *The Structure of Computers and Computations.* John Wiley & Sons (New-York), 1978.

[Li86] Kuo-Cheng LI. Notes on the VECTOR C language. *ACM Sigplan Notices* **21**(1):49–57, Janvier 1986.

[LiCh91] Jingke LI et Marina CHEN. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems* **2**(3):361–376, Juillet 1991.

[LiSa87] A. LICHNEWSKY et C. SAGUEZ (eds.). *Supercomputing — State-of-the-Art.* Elsevier Science Publishers B. V. (North-Holland), 1987.

[LiSc84] Kuo-Cheng LI and Herb SCHWETMAN. Implementing a scalar C compiler on the Cyber 205. *Software—Practice and Experience* **14**(9):867–888, Septembre 1984.

[LiSc85] Kuo-Cheng LI and Herb SCHWETMAN. VECTOR C: A vector processing language. *Journal of Parallel and Distributed Computing* **2**:132–169, Mai 1985.

[LLBR75] D. H. LAWRIE, T. LAYMAN, D. BAER, et J. M. RANDAL. Glypnir — A programming language for Illiac IV. *Communication of the ACM* **18**(3):157–164, Mars 1975.

[Marq90] Philippe MARQUET. The EVA language grammar. Rapport **ERA-82**, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, Mai 1990.

[Marq91] Philippe MARQUET. Manuel de référence du langage EVA. Rapport **ERA-91**, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, Février 1991.

[Masp91c] MasPar Computer Corp. *MasPar FORTRAN—Reference Manual, Software Version 1.0*, Doc. 9303–0000, Rev. A1, Mars 1991.

[Masp91a] MasPar Computer Corp. *MasPar Parallel Application Language (MPL)—Reference Manual, Software Version 2.0*, Doc. 9302–0000, Rev. A4, Mars 1991.

[Masp91b] MasPar Computer Corp. *MasPar Parallel Application Language (MPL)—User Guide, Software Version 2.0*, Doc. 9302–0100, Rev. A4, Mars 1991.

[MeRe90] Michael METCALF et John REID. *FORTRAN 8x Explained (revised edition)*. Oxford Science Publication, reprinted with corrections 1990.

[Metc88] Michael METCALF. *Effective FORTRAN 77*. Oxford University Press (Oxford), 1985, reprinted with corrections 1988.

[Nage90] Wolfgang E. NAGEL. Exploiting autotasking on a Cray Y-MP: an improved software interface to multitasking. *Parallel Computiong* **13**(2):225–233, Février 1990.

[Nage90] Wolfgang E. NAGEL. Exploiting autotasking on a Cray Y-MP: an improved software interface to multitasking. *Parallel Computiong* **13**(2):225–233, Février 1990.

[NEC89] NEC Corporation. *NEC Supercomputer SX-3 Series General Description*. Publication GAZ02E-1, Août 1989.

[Nick90] John R. NICKOLLS. The design of the MasPar MP-1 : a cost effective massively parallel computer. *Proc. of the IEEE Compcon Spring 1990*, San Francisco, CA, Février 3–5, 1990, IEEE Society Press, Février 1990, pages 25–28.

[NPA88] Alexandru NICOLAU, Keshav PINGALI, et Alexander AIKEN. Fine-grained compilation for pipelined machines. *The Journal of Supercomputing* **2**(3):279–295, Novembre 1988.

[OeLa85] W. OED et O. LANGE. On the effective bandwidth in interleaved memories in vector processingg systems. *IEEE Transactions on Computers* **C–34**(10):949–957, Octobre 1985.

[ONK86] Toshihiko ODAKA, Shiegeo NAGASHIMA, et Shun KAWABE. Hitachi supercomputer S-810 array processor system. In *Supercomputers, Class VI Systems, Hardware and Software*. S. FERNBACH (ed.). Elsevier Science Publishers B.V. (North-Holland), 1986, pages 113–136.

[Pari91] Nicolas PARIS. Définition de POMPC (version 1.11). Rapport de Recherches, LIENS, École Normale Supérieure, Paris, 1991.

[Paul84] George PAUL. VECTRAN and the proposed vector/array extensions to ANSI FORTRAN for scientific and engineering computation. *Proc. of the IBM Conf. on Parallel Computers and Scientific Computations*, Rome, Italy, 1982. Reprinted in H. KWANG (ed.). *Tutorial on Supercomputers: Design and Application*. IEEE Press, 1984.

[PaWi75] George PAUL and M. Wayne WILSON. The VECTRAN language: An experimental language for vector/matrix array processing. Technical report **G320-3334**, IBM Palo Alto Scientific Center, Palo Alto (CA), Août 1975.

[PaWi78] George PAUL and M. Wayne WILSON. An introduction to VECTRAN and its use in scientific applications programming. *Proc. of the 1978 Workshop on Vector and Parallel Processors*, Los Alamos (NM), Septembre 20–22, 1978.

[PaWo86] David A. PADUA et Michael J. WOLFE. Advanced compiler optimizations for supercomputers. *CACM* **29**(12):1184–1201, Décembre 1986.

[PCM83] R. H. PERROTT, D. CROOKES, et P. MILLIGAN. The programming language Actus. *Software—Practice and Experience* **13**(4):305–322, Avril 1983.

[PCM84] R. H. PERROTT, D. CROOKES, et P. MILLIGAN. Implementing a parallel language on the Cray 1. *Proc of the 2nd Int'l Conf. on Vector and Parallel Processors in Computationnal Science*, Oxford 28–31 Août, 1984. In *Computer Physics Communications* **37**:119–124, 1985.

[PCMP83] R. H. PERROTT, D. CROOKES, P. MILLIGAN, et W. R. M. PURDY. A compiler for the synchronous parallel programming language Actus. *Proc of the Int'l Conf. on Paralell Computing 83*. In M. FEILMEIER, J. JOUBERT et U. SCHENDEL (eds.). *Parallel Computing 83*. North-Holland, 1984, pages 475–480.

[PCMP85] R. H. PERROTT, D. CROOKES, P. MILLIGAN, et W. R. M. PURDY. A compiler for an array and vector processing language. *IEEE Transactions on Software Engineering* **SE-11**(5):471–478, Mai 1985.

[PeLu91] R. H. PERROT and T. F. LUNNEY. A syntax-directed integrated programming environment for developing SIMD supecomputer software. *Software—Practice and Experience* **21**(3):2639–286, Mars 1991.

[Perr79a] R. H. PERROTT. A language for array and vector processors. *ACM Trans. on Programming Languages and Systems* **1**(2):177–195, Octobre 1979.

[Perr79b] R. H. PERROTT. ACTUS — A language for array and vector processors. User manual. Rapport CS005, Department of Computer Science, The Queen's University, Belfast, N. Ireland, Juillet 1979, révisé Septembre 1981.

[Perr87] Ronald H. PERROTT. *Parallel Programming*. Addison-Wesley (Int'l computer science serie), 1987.

[PeSt81a] R. H. PERROT and D. K. STEVENSON. Considerations for the design of array processing languages. *Software—Practice and Experience* **11**:683–688, 1981.

[PeSt81b] R. H. PERROT and D. K. STEVENSON. Users' experiences with the Illiac IV system and its programming languages. *ACM Sigplan Notices* **16**(7):75–88, July 1981.

[Peti88] Serge PETITON. *Du Développement de Logiciels Numériques en Environnements Parallèles*. Thèse de doctorat de l'université Paris 6, Décembre 1988, Publiée dans Rapport MASI 91.27, Mai 1991.

[PeZa87] R. H. PERROTT et Adib ZAERA-ALIABADI. A supercomputer program development system. *Software—Practice and Experience* **17**(10):663–683, October 1987.

[PLD87] R. H. PERROTT, R. W. LYTTLE, et P. S. DHILLON. The design and implementation of a Pascal-based language for array processor architectures. *Journal of Parallel and Distributed Computing* **4**:266–287, 1987.

[Preu90] Philippe PREUX. Présentation de la machine virtuelle MAD et de son langage DEVIL. Rapport **ERA-81**, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, Mai 1990.

[Preu91] Philippe PREUX. *MAD : Une Machine Virtuelle Vectorielle—Conséquences sur l'Architecture des Machines Vectorielles*. Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, Janvier 1991.

[RaHa90] Ram RAGHAVAN et John P. HAYES. On randomly interleaved memories. *Proc. Supercomputing'90*, New-York, Novembre 1990, pages 49–58.

[RaSa89] J. RAMANUJAM et P. SADAYAPPAN. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. *Proc. Supercomputing'89*, Reno (Nevada), Novembre 1989, pages 637–646.

[Reid89] John K. REID. FORTRAN 8x, the new FORTRAN standard. Report **AERE R 12857**, Computer Science and Systems Division, Harwell Laboratory, Oxfordshire, UK, Août 1989 (2nd édition).

[Rein85] Steve REINHARDT. A data-flow approach to multitasking on Cray X-MP computers. *ACM Operating Systems Review* **13**(5):107–114, 1985.

[Rein88] Steve REINHARDT. Two parallel processing aspects of the Cray Y-MP computer systems. *Proc. of the Int'l Conf. on Parallel Processing 1988 (ICPP'88)*, pages 311–314.

[Rose87] John ROSE. C*: a C++-like language for data-parallel computation. *Proc. USENIX C++ Conf.*, Décembre 1987.

[RoSt87] John ROSE et Guy L. STEELE Jr. C*: an extented C language for data parallel programming. Thinking Machine Technical Report PL87–5, Avril 1987.

[Schl91] Judith SCHLESINGER. DBC reference manual, Techgnical report, Supercomputing Research Center, Bowie (MD), 1991.

[Siem88a] Siemens. *Vector Processor System VP Series Hardware Principles of Operations*. Publication U2006-J-Z69-3-7600, Mars 1988.

[Siem88b] Siemens. *Vector Processor System VP-EX Series Hardware Principles of Operations*. Publication U4012-J-Z69-3-7600, Juillet 1988.

[Schö87a] Willi SCHÖNAUER. *Scientific Computing On Vector Computers*. Special Topics in Supercomputing 2, North-Holland, 1987.

[Schö87b] Willi SCHÖNAUER. The ETA 10. In [Schö87a], *Scientific Computing On Vector Computers*. Willi SCHÖNAUER. Special Topics in Supercomputing 2, North-Holland, 1987.

[Stev75] K. G. STEVEN Jr. CFD — A Fortran-like language for the Illiac IV. *ACM Sigplan Notices* **10**(3):72–76, Mars 1975.

[Ston87] Harnold S. STONE. *High-Performance Computer Architecture*. Addison-Wesley Publishing Company, 1987.

[TaMe89] P. TANG et R. H. MENDEZ. Memory conflicts and machines performances. *Proc. Supercomputing'89*, Reno (Nevada), Novembre 1989, pages 826–831.

[TMC89] Thinking Machines Corporation. *Paris Reference Manual*. Version 5.0, release 5.2, Thinking Machines Corporation (Cambridge, MA), Octobre 1989.

[TMC90a] Thinking Machines Corporation. *Getting Started in CM FORTRAN*. Version 5.2–0.6, Thinking Machines Corporation (Cambridge, MA), Fevrier 1990.

[TMC90b] Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary*. Version 6.0, Thinking Machines Corporation (Cambridge, MA), Novembre 1990.

[TMC91a] Thinking Machines Corporation. *Connection Machine CM-200 Series Technical Summary*. Thinking Machines Corporation (Cambridge, MA), Juin 1991.

[TMC91b] Thinking Machines Corporation. *Getting Started in C**. Thinking Machines Corporation (Cambridge, MA), Février 1991.

[TrRo88] Lewis W. TUCKER et George G. ROBERTSON. Architecture and applications of the Connection Machine. *IEEE Computer* **21**(8):26–38, Août 1988.

[TsKu86] Takao TSUDA et Yoshitoshi KUNIEDA. Mechanical vectorization of multiply nested FOR loop by vector indirect addressing. *Proc. IFIP 10th World Computer Congress*, Dublin, Ireland, Sept. 1–5, 1986, pages 785–790.

[Tsud88] Takao TSUDA. Asynchrounous parallel execution of a multiply nested FOR loop by vector indirect addressing. *Proc IFIP Working Conf. on Aspects of Computation on Asynchronous Parallel Processors*. Standford, CA, Août 22–26, 1988, pages 101–110.

[TsKu90] Takao TSUDA et Yoshitoshi KUNIEDA. V-PASCAL: an automatic vectorizing compiler for PASCAL with no language extensions. *The Journal of Supercomputing* **4**(3):251–275, Septembre 1990.

[Wave91] WaveTracer. The multiC programming language. Technical report PUB–00001–001–1.00, WaveTracer, Inc. 1991.

[Wede75] Dorothy WEDEL. FORTRAN for the Texas Instrument ASC System. *ACM Sigplan Notices* **10**(3):119–132, Mars 1975.

[Weth80] Charles WETHERELL. Design considerations for array processing languages. *Software—Practice and Experience* **10**(4):265–271, Avril 1980.

[Wolf89] Michael J. WOLFE. *Optimizing Supercompilers for Supercomputers*. Pitman (London) et MIT Press (Cambridge, MA), 1989.

[Wolf91] Michael J. WOLFE. Data dependence and program restructuring. *The Journal of Supercomputing* **4**(4):321–344, Jan. 1991.

[ZiCh90] Hans ZIMA with Barbara CHAPMAN. *Supercompilers for Parallell and Vector Computers*. ACM Press (New-York), Addison-Wesley, 1990.

[Zwak75] R. G. ZWAKENBERG. Vector extension to LRLTRAN. *ACM Sigplan Notices* **10**(3):77–86, Mars 1975.

Annexe A

Pseudo-assembleur vectoriel

Nous définissons un pseudo-assembleur vectoriel. Cet assembleur est une forme lisible proche du CAL (Cray Assembly Language). Il est utilisé pour la présentation de code CAL au sein de ce document. Il n'est en aucun cas destiné à être implanté.

Cet assembleur est incomplet; les instructions non utilisées dans ce document n'étant pas données ici.

Cet assembleur utilise des registres vectoriels `Vi`, des registres scalaires `Si` et des registres d'adresses `Ai`. Il existe un registre `VL`, contrôlant la longueur effective des registres vectoriel, et un registre `VM`, contrôlant les opérations de *merge* et *mask*.

Les instructions calculatoires sont réalisées registre à registre.

Nous utilisons la constante symbolique `VRS` (Vector Register Size) pour dénoter le nombre d'éléments contenus dans un registre vectoriel.

Une ligne de commentaire est introduite par deux tirets
    -- la fin de la ligne est un commentaire

Les étiquettes de saut sont de la forme
    <nom d'étiquette> :

Instructions

Les instructions sont codées sur le modèle suivant :
    <nom d'instruction> <opérande cible> <opérande(s) source(s)>

Dans les opérandes des instructions, *Vi* dénote un registre vectoriel, *Ai* un registre d'adresses, *Si* un registre scalaire, *ASi* un registre d'adresse ou un registre scalaire, et *VASi* un registre vectoriel, un registre d'adresse ou un registre scalaire.

Dans les opérandes des instructions de chargement et rangement mémoire, *VAR* [*Ai*] désigne une adresse mémoire. *VAR* est le nom d'une variable, *Ai* est un déplacement par rapport à l'adresse de base de cette variable.

Les opérandes entre accolades { et } sont optionnels.

Attente de la fin des accès mémoire (lecture et écriture) en cours

```
CMR
```

Chargement et rangement scalaire :

```
LOAD      ASi  Aj
STORE     Ai   ASj
```

Chargements vectoriels direct, par pas, et gather :

```
VLOAD     Vi   Aj
VLOAD     Vi   VAR [Aj]
VLOADs    Vi   Aj          Step
VLOADs    Vi   VAR [Aj]    Step
VLOADi    Vi   Aj          Vindirect
VLOADi    Vi   VAR [Aj]    Vindirect
```

Rangements mémoire vectoriels direct, par pas, et scatter :

```
VSTORE    Ai        Vj
VSTORE    VAR [Ai]  Vj
VSTOREs   Ai        Vj     Step
VSTOREs   VAR [Ai]  Vj     Step
VSTOREi   Ai        Vj     Vindirect
VSTOREs   VAR [Ai]  Vj     Vindirect
```

Sauts inconditionnel, si non nul et si nul :

```
JUMP      étiquette
JNZ       étiquette   Si
JZ        étiquette   Si
```

Appel de routine
(Ces instructions cachent la gestion de la pile, la sauvegarde du contexte. De plus, le résultat peut être produit dans un registre)

```
CALL      routine          {VASi}
CALL      ASi  routine     {VASi}
VCALL     Vi   routine     {VASi}
```

Transferts scalaire et vectoriel

```
MOVE      ASi  ASj
VMOVE     Vi   VASj
```

Opérations d'inversion scalaires et vectorielles
(Les instructions à deux opérandes ont une cible et une source; les instructions à une opérande réalisent l'opération <cible> = <op> <cible>)

```
MINUS     ASi  {ASj}
VMINUS    Vi   {VASj}
INV       ASi  {ASj}
VINV      Vi   {VASj}
```

Opérations arithmétiques scalaires et vectorielles
(Les instructions à trois opérandes ont une cible et deux sources; les instructions à deux opérandes réalisent l'opération <cible> = <cible> op <source>)

ADD	ASi	ASj	{ASk}
VADD	Vi	VASj	{VASk}
SUB	ASi	ASj	{ASk}
VSUB	Vi	VASj	{VASk}
MUL	ASi	ASj	{ASk}
VMUL	Vi	VASj	{VASk}

Incrémentation et décrémentation

INCR	ASi
DECR	ASi

Annexe B

Les langages à parallélisme de données

Dans le premier chapitre de ce document (section 1–4), nous avons donné un large éventail des langages utilisés pour la programmation à parallélisme de données. De nombreuses caractéristiques communes ont émergé de ce tour d'horizon. Cette annexe résume les particularités de chacun des langages. Pour ce faire, nous avons dressé une liste de caractéristiques et donnons la position de chacun des langages vis-à-vis de ces caractéristiques.

Ces caractéristiques sont : le nom du langage, une date, le langage de base duquel il est dérivé, les machines ciblées par le langage (machines pipelines vectorielles, machines tableaux, machines massivement parallèles, une machine particulière), la forme des objets vecteurs manipulés, leur virtualité (taille, nombre de dimensions, placement) et leur dynamicité, les caractéristiques de l'allocation de ces vecteurs, les opérations de descriptions existantes et/ou les opérations de comunications disponibles, les strutures de contrôle parallèles proposées, les caractéristiques des fonctions et des passages de vecteur en paramètre.

Nom	**VECTRAN**
Date	1975
Basé sur	FORTRAN (sur-ensemble de FORTRAN)
Machines ciblées	machines pipelines vectorielles (sans précision)
Objets vecteurs	"array" (classique tableau FORTRAN) et "virtual array" (association d'un tableau et d'un descripteur triplet)
allocation	association dymnamique `IDENTIFY` allocation statique (FORTRAN)
Opération de description	par triplet (y compris lors d'`IDENTIFY`), par liste d'index
Structure de contôle parallèle	affectations contrôlées `WHEN` et `AT`
Fonctions	extension de FORTRAN, retour de vecteur de taille statique
Paramètre vecteur	passage de vecteurs, passage de sous-vecteurs (triplets) par utilisation d'association.

Nom	**FORTRAN 200**
Date	1983
Basé sur	FORTRAN (sur-ensemble de FORTRAN)

Machines ciblées	machines CDC Cyber 200, 203, et 205
Objets vecteurs	"vector reference" (couple base, longueur) et "descriptor" (association d'un couple base, longueur)
allocation	association dynamique d'une base et une longueur (**ASSIGN**) allocation dynamique (**ASSIGN**)
Opération de description	par triplet (y compris lors d'**IDENTIFY**), par liste d'index
Structure de contôle parallèle	affectation contrôlée **WHERE**
Fonctions	extension de FORTRAN, retour de vecteur de taille variable
Paramètre vecteur	passage de vecteurs, passage de sous-vecteurs (descripteurs).

Nom	**ACTUS**
Date	1975
Basé sur	PASCAL
Machines ciblées	Cray 1, puis ICL DAP (ACTUS II), sans que ce soit spécifique
Objets vecteurs	"tableau parallèle", nombre de dimensions parallèles figé et limité
allocation	statique (PASCAL)
Opération de description	par triplet, par liste d'index (projection)
Structure de contôle parallèle	extensions parallèles des **while**, **for**, etc. de PASCAL
Fonctions	extension de PASCAL, retour de tableau de taille tableau de taille fixe
Paramètre vecteur	passage de tableaux de taille variable, nombre de dimensions parallèles fixe.

Nom	**FORTRAN 90**
Date	1990
Basé sur	l'ancienne norme FORTRAN 77
Machines ciblées	toutes
Objets vecteurs	tableau, "pointeur" (association d'un tableau et d'un triplet)
allocation	statique, automatique et dynamique
Opération de description	par triplet, par liste d'index
Structure de contôle parallèle	construction de blocs **WHERE**
Fonctions	retour de tableau de taille tableau de taille fixe
Paramètre vecteur	passage de tableaux de taille variable, passage de sous-tableaux décrits uniquement par des triplets.

Nom	**CM FORTRAN**
Date	1989
Basé sur	FORTRAN 90
Machines ciblées	Connection Machine CM
Objets vecteurs	tableau
allocation	statique, allocation sur le frontal ou les processeurs (**LAYOUT**) et alignement des tableaux (**ALIGN**)
Opération de description	par triplet, par liste d'index
Structure de contôle parallèle	instruction d'affectation **WHERE**
Fonctions	pas de retour de tableau
Paramètre vecteur	passage de tableaux de taille variable, passage de sous-tableaux décrits uniquement par des triplets, alignement et allocation frontal/PE figé.

Nom	**FORTRAN D**
Date	1990
Basé sur	FORTRAN
Machines ciblées	toutes (y compris MIMD)
Objets vecteurs	tableau (accès scalaire)
allocation	statique (FORTRAN), alignement et distribution des tableaux (`ALIGN`, `DISTRIBUTE`) dynamiques
Opération de description	scalaire (y compris `FORALL`)
Structure de contôle parallèle	structure de contrôle `FORALL`
Fonctions	pas de retour de tableau
Paramètre vecteur	passage de tableaux de taille, alignement, et distribution variable (passage par une forme normale).

Nom	**C***
Date	(1985 pour la première version), 1990
Basé sur	C
Machines ciblées	Connection Machines
Objets vecteurs	"shape" (classe de vecteurs)
allocation	dynamique (taille et nombre de dimensions des "shapes", taille puissance de 2)
Opération de description	communication globales et de grille
Structure de contôle parallèle	positionnement de la "shape"courante (`with`), positionnement de contexte (`where`)
Fonctions	retour de vecteur de shape variable, définition de fonction de shapes "génériques"
Paramètre vecteur	passage de vecteurs de shapes "génériques".

Nom	**POMPC**
Date	1990
Basé sur	C
Machines ciblées	toutes les machines massivement parallèles
Objets vecteurs	"collection" (classe de vecteurs)
allocation	dynamique (taille et nombre de dimensions des "collections")
Opération de description	communication globales
Structure de contôle parallèle	extension de toutes les structures de contrôle de C, ("collection courante" déduite des opérateurs)
Fonctions	retour de vecteur de shape variable, définition de fonction de collections "génériques"
Paramètre vecteur	passage de vecteurs de collections "génériques".

Nom	**MPL**
Date	1990
Basé sur	C
Machines ciblées	MasPar MP
Objets vecteurs	"plural" (classe de vecteurs de la taille de la machine physique)
allocation	taille et nombre de dimensions des variables "plural" figées par l'architecture

Opération de description	communication globales et de grille
Structure de contôle parallèle	extension de toutes les structures de contrôle de C
Fonctions	retour de valeur "plural"
Paramètre vecteur	passage de valeurs "plural.