

jeu 2010 3583



50376
1995
39



50376
1995
39



THÈSE

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

par

Dominique LAZURE

Programmation géométrique à parallélisme de données : modèle, langage et compilation

Thèse soutenue le 13 Janvier 1995, devant la commission d'examen :

Président :	Serge PETITON	Professeur	LIFL - USTL
Rapporteurs :	Luc BOUGÉ	Professeur	LIP - ENS Lyon
	Paul FEAUTRIER	Professeur	PRISM - UVSQ
Examineurs :	Jean-Luc DEKEYSER	Professeur	LIFL - USTL
	Philippe MARQUET	Maître de conférences	LIFL - USTL
	Jean-Louis PAZAT	Maître de conférences	IRISA - INSA

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

U.F.R. d'I.E.E.A. Bât. M3 - 59655 VILLENEUVE D'ASCQ CEDEX

Tél. (33) 20 43 47 24 Télécopie (33) 20 43 65 66

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. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS, A. DUBRULLE

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CHAMLEY Hervé	Géotechnique
M. CONSTANT Eugène	Electronique
M. ESCAIG Bertrand	Physique du solide
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. LABLACHE COMBIER Alain	Chimie
M. LOMBARD Jacques	Sociologie
M. MACKE Bruno	Physique moléculaire et rayonnements atmosphériques

M. MIGEON Michel
M. MONTREUIL Jean
M. PARREAU Michel
M. TRIDOT Gabriel

EUDIL
Biochimie
Analyse
Chimie appliquée

PROFESSEURS - 1^{ère} CLASSE

M. BACCHUS Pierre	Astronomie
M. BLAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie-Physique
M. BOSCOQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BRASSELET Jean Paul	Géométrie et topologie
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie Physique
M. BRUYELLE Pierre	Géographie
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. CROSNIER Yves	Electronique
Mme DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DEBRABANT Pierre	Géologie appliquée
M. DECLERCQ Roger	Sciences de gestion
M. DEGAUQUE Pierre	Electronique
M. DESCHEPPER Joseph	Sciences de gestion
Mme DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DHAINAUT André	Biologie animale
Mme DHAINAUT Nicole	Biologie animale
M. DJAFARI Rouhani	Physique
M. DORMARD Serge	Sciences Economiques
M. DOUKHAN Jean Claude	Physique du solide
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUPOUY Jean Paul	Biologie
M. DYMENT Arthur	Mécanique
M. FOCT Jacques Jacques	Métallurgie
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. FRONTIER Serge	Ecologie numérique
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie-Physique
M. GRANELLE Jean Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILBAULT Pierre	Physiologie animale
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. LACOSTE Louis	Biologie Végétale
M. LANGRAND Claude	Probabilités et statistiques

M. LATTEUX Michel
M. LAVEINE Jean Pierre
Mme LECLERCQ Ginette
M. LEHMANN Daniel
Mme LENOBLE Jacqueline
M. LEROY Jean Marie
M. LHENAFF René
M. LHOMME Jean
M. LOUAGE François
M. LOUCHEUX Claude
M. LUCQUIN Michel
M. MAILLET Pierre
M. MAROUF Nadir
M. MICHEAU Pierre
M. PAQUET Jacques
M. PASZKOWSKI Stéfan
M. PETIT Francis
M. PORCHET Maurice
M. POUZET Pierre
M. POVY Lucien
M. PROUVOST Jean
M. RACZY Ladislas
M. RAMAN Jean Pierre
M. SALMER Georges
M. SCHAMPS Joël
Mme SCHWARZBACH Yvette
M. SEGUIER Guy
M. SIMON Michel
M. SLIWA Henri
M. SOMME Jean
Melle SPIK Geneviève
M. STANKIEWICZ François
M. THIEBAULT François
M. THOMAS Jean Claude
M. THUMERELLE Pierre
M. TILLIEU Jacques
M. TOULOTTE Jean Marc
M. TREANTON Jean René
M. TURRELL Georges
M. VANEECLOO Nicolas
M. VAST Pierre
M. VERBERT André
M. VERNET Philippe
M. VIDAL Pierre
M. WALLART François
M. WEINSTEIN Olivier
M. ZEYTOUNIAN Radyadour

Informatique
Paléontologie
Catalyse
Géométrie
Physique atomique et moléculaire
Spectrochimie
Géographie
Chimie organique biologique
Electronique
Chimie-Physique
Chimie physique
Sciences Economiques
Sociologie
Mécanique des fluides
Géologie générale
Mathématiques
Chimie organique
Biologie animale
Modélisation - calcul scientifique
Automatique
Minéralogie
Electronique
Sciences de gestion
Electronique
Spectroscopie moléculaire
Géométrie
Electrotechnique
Sociologie
Chimie organique
Géographie
Biochimie
Sciences Economiques
Sciences de la Terre
Géométrie - Topologie
Démographie - Géographie humaine
Physique théorique
Automatique
Sociologie du travail
Spectrochimie infrarouge et raman
Sciences Economiques
Chimie inorganique
Biochimie
Génétique
Automatique
Spectrochimie infrarouge et raman
Analyse économique de la recherche et développement
Mécanique

PROFESSEURS - 2ème CLASSE

M. ABRAHAM Francis	Composants électroniques
M. ALLAMANDO Etienne	Biologie des organismes
M. ANDRIES Jean Claude	Analyse
M. ANTOINE Philippe	Génétique
M. BALL Steven	Biologie animale
M. BART André	Génie des procédés et réactions chimiques
M. BASSERY Louis	Géographie
Mme BATTIAU Yvonne	Systèmes électroniques
M. BAUSIERE Robert	Mécanique
M. BEGUIN Paul	Physique atomique et moléculaire
M. BELLET Jean	Physique atomique, moléculaire et du rayonnement
M. BERNAGE Pascal	Sciences Economiques
M. BERTHOUD Arnaud	Sciences Economiques
M. BERTRAND Hugues	Analyse
M. BERZIN Robert	Physique de l'état condensé et cristallographie
M. BISKUPSKI Gérard	Algèbre
M. BKOUCHE Rudolphe	Biologie végétale
M. BODARD Marcel	Biochimie métabolique et cellulaire
M. BOHIN Jean Pierre	Mécanique
M. BOIS Pierre	Génie civil
M. BOISSIER Daniel	Spectrochimie
M. BOIVIN Jean Claude	Physique
M. BOUCHER Daniel	Biologie appliquée aux enzymes
M. BOUQUELET Stéphane	Gestion
M. BOUQUIN Henri	Chimie
M. BROCARD Jacques	Paléontologie
Mme BROUSMICHE Claudine	Mécanique
M. BUISINE Daniel	Biologie animale
M. CAPURON Alfred	Géographie humaine
M. CARRE François	Chimie organique
M. CATTEAU Jean Pierre	Sciences Economiques
M. CAYATTE Jean Louis	Electronique
M. CHAPOTON Alain	Biochimie structurale
M. CHARET Pierre	Composants électroniques optiques
M. CHIVE Maurice	Informatique théorique
M. COMYN Gérard	Composants électroniques et optiques
Mme CONSTANT Monique	Psychophysiologie
M. COQUERY Jean Marie	Sciences Economiques
M. CORIAT Benjamin	Paléontologie
Mme CORSIN Paule	Physique nucléaire et corpusculaire
M. CORTOIS Jean	Chimie organique
M. COUTURIER Daniel	Tectonique géodynamique
M. CRAMPON Norbert	Biologie
M. CURGY Jean Jacques	Physique théorique
M. DANGOISSE Didier	Analyse
M. DE PARIS Jean Claude	Composants électroniques et optiques
M. DECOSTER Didier	Electrochimie et Cinétique
M. DEJAEGER Roger	Informatique
M. DELAHAYE Jean Paul	Physiologie animale
M. DELORME Pierre	Sciences Economiques
M. DELORME Robert	Sociologie
M. DEMUNTER Paul	Physique atomique, moléculaire et du rayonnement
Mme DEMUYNCK Claire	Informatique
M. DENEL Jacques	Physique du solide - cristallographie
M. DEPREZ Gilbert	

M. DERIEUX Jean Claude	Microbiologie
M. DERYCKE Alain	Informatique
M. DESCAMPS Marc	Physique de l'état condensé et cristallographie
M. DEVRAINNE Pierre	Chimie minérale
M. DEWAILLY Jean Michel	Géographie humaine
M. DHAMELINCOURT Paul	Chimie physique
M. DI PERSIO Jean	Physique de l'état condensé et cristallographie
M. DUBAR Claude	Sociologie démographique
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBOIS Jean Jacques	Géographie
M. DUBUS Jean Paul	Spectrométrie des solides
M. DUPONT Christophe	Vie de la firme
M. DUTHOIT Bruno	Génie civil
Mme DUVAL Anne	Algèbre
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FARVACQUE Jean Louis	Physique de l'état condensé et cristallographie
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FELIX Yves	Mathématiques
M. FERRIERE Jacky	Tectonique - Géodynamique
M. FISCHER Jean Claude	Chimie organique, minérale et analytique
M. FONTAINE Hubert	Dynamique des cristaux
M. FORSE Michel	Sociologie
M. GADREY Jean	Sciences économiques
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GOBLOT Rémi	Algèbre
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. GUELTON Michel	Chimie physique
M. GUICHAOUA André	Sociologie
M. HAIMAN Georges	Modélisation,calcul scientifique, statistiques
M. HOUDART René	Physique atomique
M. HUEBSCHMANN Johannes	Mathématiques
M. HUTTNER Marc	Algèbre
M. ISAERT Noël	Physique de l'état condensé et cristallographie
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme
M. JOURNAL Gérard	Spectroscopie hertzienne
M. KOENIG Gérard	Sciences de gestion
M. KOSTRUBIEC Benjamin	Géographie
M. KREMBEL Jean	Biochimie
Mme KRIFA Hadjila	Sciences Economiques
M. LANGEVIN Michel	Algèbre
M. LASSALLE Bernard	Embryologie et biologie de la différenciation
M. LE MEHAUTE Alain	Modélisation,calcul scientifique,statistiques
M. LEBFEVRE Yannic	Physique atomique,moléculaire et du rayonnement
M. LECLERCQ Lucien	Chimie physique
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Marc	Composants électroniques et optiques
M. LEFEVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
M. LEGRAND Michel	Astronomie - Météorologie
M. LEGRAND Pierre	Chimie
Mme LEGRAND Solange	Algèbre
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne

M. LE MAROIS Henri	Vie de la firme
M. LEMOINE Yves	Biologie et physiologie végétales
M. LESCURE François	Algèbre
M. LESENNE Jacques	Systèmes électroniques
M. LOCQUENEUX Robert	Physique théorique
Mme LOPES Maria	Mathématiques
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU François	Sciences économiques
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MANSY Jean Louis	Géologie
M. MAURISSON Patrick	Sciences Economiques
M. MERIAUX Michel	EUDIL
M. MERLIN Jean Claude	Chimie
M. MESMACQUE Gérard	Génie mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MOCHE Raymond	Modélisation, calcul scientifique, statistiques
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORE Marcel	Physique de l'état condensé et cristallographie
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NIAY Pierre	Physique atomique, moléculaire et du rayonnement
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PALAVIT Gérard	Génie chimique
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. PERTUZON Emile	Physiologie animale
M. PETIT Daniel	Biologie des populations et écosystèmes
M. PLIHON Dominique	Sciences Economiques
M. PONSOLLE Louis	Chimie physique
M. POSTAIRE Jack	Informatique industrielle
M. RAMBOUR Serge	Biologie
M. RENARD Jean Pierre	Géographie humaine
M. RENARD Philippe	Sciences de gestion
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean Claude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROLLAND Paul	Composants électroniques et optiques
M. ROLLET Philippe	Sciences Economiques
Mme ROUSSEL Isabelle	Géographie physique
M. ROUSSIGNOL Michel	Modélisation, calcul scientifique, statistiques
M. ROY Jean Claude	Psychophysiologie
M. SALERNO François	Sciences de gestion
M. SANCHOLLE Michel	Biologie et physiologie végétales
Mme SANDIG Anna Margarete	
M. SAWERYSYN Jean Pierre	Chimie physique
M. STAROSWIECKI Marcel	Informatique
M. STEEN Jean Pierre	Informatique
Mme STELLMACHER Irène	Astronomie - Météorologie
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. TANRE Daniel	Géométrie - Topologie
M. THERY Pierre	Systèmes électroniques
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du travail

M. TURREL Georges
M. VANDIJK Hendrik
Mme VAN ISEGHEM Jeanine
M. VANDORPE Bernard
M. VASSEUR Christian
M. VASSEUR Jacques
Mme VIANO Marie Claude
M. WACRENIER Jean Marie
M. WARTEL Michel
M. WATERLOT Michel
M. WEICHERT Dieter
M. WERNER Georges
M. WIGNACOURT Jean Pierre
M. WOZNIAK Michel
Mme ZINN JUSTIN Nicole

Spectrochimie infrarouge et raman

Modélisation, calcul scientifique, statistiques
Chimie minérale
Automatique
Biologie

Electronique
Chimie inorganique
géologie générale
Génie mécanique
Informatique théorique

Spectrochimie
Algèbre

Remerciements

Je remercie vivement Serge Petiton qui a accepté de présider ce jury. Je lui suis aussi reconnaissant de ses éclaircissements sur le domaine des matrices creuses qui ont permis d'étendre la portée de mes travaux.

Je remercie Luc Bougé qui a accepté de rapporter cette thèse. Sa disponibilité et ses conseils ont contribué à améliorer ce document. Son ardeur pour l'organisation des réunions nationales comme *C³parad* ou *Paradigme* m'ont permis d'entrer en contact avec la communauté, je l'en remercie.

Je remercie Paul Feautrier qui a accepté de rapporter cette thèse, et pour l'intérêt qu'il a manifesté pour ces travaux. Ses remarques constructives m'ont permis de retoucher ce document pour y effacer quelques maladresses.

Je remercie Jean-Louis Pazat pour son enthousiasme et pour sa présence comme examinateur dans ce jury.

Je remercie Jean-Luc Dekeyser de m'avoir ouvert la porte de son équipe en début de thèse. Sa rigueur et son esprit critique m'ont appris la remise en cause et le bénéfice qui en découle finalement.

Je remercie Philippe Marquet. Son aide durant les trois dernières années n'a connu aucune défaillance, et m'a beaucoup apporté. Je lui suis aussi reconnaissant d'avoir accepté le difficile rôle de premier lecteur.

Je remercie Akram et Cyril, mes deux compères de bureau. Cette expérience de tri-rédaction a été pour moi une motivation supplémentaire. L'ambiance du bureau, parfois polémique souvent joviale et toujours amicale, m'a permis d'évoluer dans un cadre agréable. Ils en sont responsables.

Je remercie les autres membres de l'équipe WEST, thésards ou ex-thésards, et les membres du LIFL qui ont contribué au bon déroulement de mes travaux.

Je remercie Pascal pour avoir tenté de dissiper à mes yeux les mystères de l'océanographie. J'espère que les futurs échanges avec l'IFREMER seront bénéfiques pour tous.

Je remercie aussi mes autres frères pour l'aide qu'ils n'auraient pas hésité à m'apporter si je leur avais demandé.

Je remercie Catherine d'avoir supporté stoïquement mon humeur durant la rédaction de cette thèse. Ce ne fut pas facile tous les jours...

Enfin, je remercie mes parents. Simplement parce que je leur dois tout.

Table des matières

Table des matières	i
Avant-propos	1
Introduction	5
I Calcul scientifique et parallélisme	9
1 De l'histoire de l'informatique	9
2 Les modèles d'exécution	11
2.1 Le modèle <i>von Neumann</i>	11
2.2 Le modèle pipe-line	14
2.3 Le modèle parallèle synchrone	18
2.4 Le modèle parallèle asynchrone	22
3 Les super-calculateurs	26
4 Le Grand Challenge	28
4.1 Environnement logiciel	28
4.2 Les applications	29
4.3 Le parallélisme de données	31
4.4 Un modèle universel?	32
5 Conclusion	34
II Modèles de programmation à parallélisme de données	37
1 Le parallélisme de données	38
1.1 Programmation et exécution	39
1.2 Sémantique et preuves de programmes	40
1.3 Les problèmes de base	41
2 Une illustration: HPF	51
2.1 Vers un standard? Le forum HPFF	51
2.2 Le langage HPF	52
3 Les modèles à flot de données	59
3.1 Le modèle de programmation systolique	59
3.2 Le modèles basé sur les équations récurrentes	60
3.3 Le modèle 8 1/2	62
4 Les modèles de programmation par structures globales	63
4.1 Les modèles réguliers	63
4.2 Les modèles irréguliers	65
5 Le modèle géométrique	68

TABLE DES MATIÈRES

5.1	L'usage courant de la géométrie	68
5.2	Adéquation de la géométrie aux machines parallèles	74
5.3	Le modèle HELP	74
5.4	Projection	78
6	Conclusion	80
III Le langage C-HELP		81
1	Les objets du modèle HELP	82
1.1	Hyper-espace	82
1.2	Les variables parallèles : DPO	88
1.3	Variables scalaires	92
1.4	Pointeurs et tableaux	92
2	Sémantique des opérations d'affectation C-HELP	93
2.1	Association	94
2.2	Affectations injectives	94
3	Expressions data-parallèles C-HELP	96
3.1	Découpage hiérarchique	96
3.2	Opérations microscopiques	97
3.3	Opérations macroscopiques	105
4	Fonctions et Bibliothèques	120
4.1	Fonctions intrinsèques	121
4.2	Fonctions microscopiques	122
4.3	Passage de DPO en paramètre et retour de DPO	124
4.4	Fonction de transfert extra-hyper-espace	127
4.5	Fonctions à géométrie générique	128
4.6	Les entrées/sorties	131
5	C-HELP et l'hétérogénéité	133
6	Conclusion	140
IV Compilation du langage C-HELP		141
1	L'atelier de compilation	141
1.1	Machine cible	141
1.2	Le langage intermédiaire	143
1.3	Outils de développement	144
2	Représentation des données à l'exécution	145
2.1	DPO	145
2.2	Hyper-espaces	146
3	Conséquences du modèle HELP	147
3.1	Allocation mémoire	147
3.2	Boucle de virtualisation	157
3.3	Compilation des appels macroscopiques	160
4	Étude des performances	161
4.1	Avertissement	161
4.2	Un langage de comparaison : MP-FORTRAN	162
4.3	Allocation mémoire	164
4.4	Calculs d'adresses	165
4.5	Boucle de virtualisation	166

5	Conclusion	170
V	Vers les structures irrégulières	171
1	Le calcul « creux »	171
1.1	Compression des données	172
1.2	La programmation du creux	179
2	HELP et le creux	181
2.1	Modèle de programmation	181
2.2	Creux et hyper-espace	182
2.3	Creux et C-HELP	183
2.4	La compilation du creux	187
2.5	Exemple	190
3	Conclusion	191
VI	Des exemples	193
1	Énumération d'un vecteur	194
2	Calculs matriciels	195
2.1	La multiplication de matrices	195
2.2	Algorithme de Gauss-Jordan	197
2.3	Factorisation LU	200
2.4	Le gradient conjugué	202
3	Squelettisation d'une image	204
4	Un modèle hydro-dynamique	205
4.1	Le modèle physique	205
4.2	La programmation par le modèle géométrique	208
5	Conclusion	211
	Conclusion	213
1	Résumé des travaux	213
2	Perspectives	214
2.1	Des objets plus généraux	214
2.2	Des opérations ensemblistes	218
2.3	Des polyèdres convexes	219
2.4	Vue microscopique et rééquilibrage	220
2.5	Extension de la vue macroscopique	220
2.6	Généralisation	221
A	Programmes de test	223
1	Nombre de références dans une expression	223
1.1	Programme C-HELP	223
1.2	Programme MP-Fortran	224
2	Virtualisation	225
2.1	Programme C-HELP	225
2.2	Programme MP-Fortran	226
	Bibliographie	227
	Liste des figures	237

Avant-propos

AU COURS du déroulement de mes travaux de thèse, on m'a souvent demandé une question fort simple : « à quoi ça sert ? » C'est dans l'idée de répondre le plus simplement possible à cette interrogation que j'ai décidé d'écrire ces quelques lignes en ouverture de ce rapport.

Intimer l'ordre à une voiture de freiner pour s'arrêter au carrefour est le résultat d'un calcul difficile qui doit prendre en compte un grand nombre de paramètres qui peuvent intervenir à tout moment. Le temps de calcul est dans ce cas sévèrement borné : le résultat ne peut arriver alors que la voiture est déjà à la casse.

Prévoir précisément le temps qu'il fera dans dix jours est aussi un calcul qui demande un nombre gigantesque d'opérations élémentaires. Aujourd'hui, ce calcul s'effectue en dix jours, avec une précision moindre qu'en regardant par la fenêtre au moment où le résultat sort de l'ordinateur.

Reconstituer une image médicale pour fournir une vision en trois dimensions permet une grande précision de la part du praticien qui décide et agit en fonction du résultat qu'il observe à travers le système informatique. La précision du traitement peut directement influencer cette décision : plus la finesse des calculs sera proche de la précision maximale, plus le praticien sera en mesure de choisir la bonne solution.

Les besoins d'une informatique fiable et performante sont énormes. À chaque progression des capacités offertes, la communauté scientifique imagine de nouvelles applications qui nécessitent de plus en plus de rapidité et de précision de la part des ordinateurs. Nombreux sont les systèmes informatiques aujourd'hui couramment utilisés qui n'auraient pu être construits, ni même pensés, il y a seulement dix ou vingt ans. Les années qui viennent seront identiques sur ce point : l'informatique de forte puissance est un outil qui va permettre la conception d'autres outils qui entreront dans notre vie quotidienne.

La course à la puissance de calcul vise aujourd'hui le « Téra-flops ». La signification informatique de « flops » est une abréviation qui désigne une unité représentant un nombre d'opérations entre nombres flottants exécutées pendant une seconde. On l'utilise pour jauger la rapidité de calcul des ordinateurs. Le préfixe « Téra- » signifie simplement un million de

million.

Le challenge offert aux informaticiens est de créer un ensemble matériel et logiciel capable de résoudre des problèmes par une évaluation d'un million de millions d'opérations par seconde. Au delà du nombre d'opérations, c'est bien sûr le facteur temps qui nous intéresse dans ce mémoire. Un ordinateur, aussi peu puissant soit-il, est capable d'exécuter un million de millions de traitements en un temps plus ou moins long. Par contre, le faire en une seconde est encore une performance qui n'a jamais été atteinte par une machine. Le besoin de rapidité est pourtant primordial pour la résolution de problèmes concrets, voire vitaux.

La technologie actuelle permet l'obtention de circuits fonctionnant au rythme du giga-hertz, qui leur permet de faire un milliard d'actions par secondes. Pour construire ces puces, l'intégration est telle que les pistes ont une section de l'ordre de la taille de quelques dizaines d'atomes. Pour atteindre le Tera-flops à l'aide d'un unique processeur, ces pistes devraient donc être réduites à quelques centièmes d'atome, ce qui est malheureusement impossible : casser un atome est généralement beaucoup trop dangereux. Le fréquence de fonctionnement d'un circuit sera donc toujours largement inférieure au téra-hertz.

Un ordinateur « téra-flopique » ne peut donc reposer uniquement sur un processeur. Puisqu'un processeur ne peut pas être assez rapide, multiplions-en le nombre et partageons le travail entre eux. Le principe est rudimentaire : c'est le parallélisme.

Le principe de multiplier les unités de traitement pose le problème d'organiser efficacement ce type de machines : un travail donné n'est pas forcément facile à répartir et le fait même de le distribuer est lui-même un travail qui alourdit la tâche à effectuer. Prenons l'exemple d'un étudiant qui entame joyeusement ses travaux de thèse [Laz95] : il mettra de l'ordre de mille jours pour obtenir un résultat (comme celui que vous avez dans les mains) ; demandez à mille thésards de se grouper pour rédiger une seule thèse, il n'est pas sûr que vous obteniez un tel rapport le lendemain. Le problème est identique pour les processeurs d'une machine parallèle : avec N processeurs, on ne va pas N fois plus vite.

Il existe plusieurs facteurs qui empêchent que cet accroissement des performances théoriques ne soit pas suivi en même proportion par l'accroissement des performances effectives. La première raison est que certains travaux sont par nature « séquentiels ». En reprenant l'exemple du déroulement de la thèse, on s'aperçoit que l'avancée des travaux se fait en répondant à une question par une autre question. Le travail d'investigation se résume à choisir les bonnes questions et à suivre ainsi un cheminement inconnu au départ. On ne peut imaginer partager ce travail entre plusieurs acteurs : on ne peut répondre à une question sans avoir répondu à la précédente. Le travail global est ainsi constitué d'une *séquence* de travaux élémentaires.

La seconde raison est celle sur laquelle on peut intervenir afin d'en limiter les conséquences.

Un travail qui peut être partagé entre plusieurs acteurs n'est pas facile à caractériser finement. Pour rédiger un rapport, mille thésards peuvent le partager en chapitres, puis en section, puis en paragraphes ; et distribuer un paragraphe par rédacteur. Alors que la rédaction de deux chapitres semble être possible sans interférences, la rédaction de deux paragraphes qui se suivent doit amener les rédacteurs à coopérer. Ce temps de coopération ralentit fatalement le processus global et entraîne une perte de temps par rapport au rendement maximal théorique. Le rôle de l'encadrement consiste en partie à organiser les acteurs de façon à limiter ces surcoûts introduits par l'idée même de partage du travail.

Voilà le cadre dans lequel un grand nombre d'informaticiens évoluent pour accroître sans cesse la puissance dispensée par les calculateurs des divers laboratoires scientifiques. Le but de ce mémoire est d'apporter une contribution, si modeste soit-elle, à la construction de l'environnement futur de l'informatique scientifique.

Introduction

AU REGARD des applications informatiques fortement consommatrices de temps de calcul et aujourd'hui en exploitation, on peut s'apercevoir que les scientifiques de tout domaine sont constamment plus exigeants sur les performances de la machine qui leur permettra d'élargir le champ de leurs investigations. La communauté informatique se doit de proposer sans cesse des solutions novatrices qui puissent apporter aux utilisateurs un moyen le plus simple possible d'accéder à leurs requêtes de performances.

Le parallélisme est une idée simple dans sa conception mais sa réalisation pose de gros problèmes quant à l'obtention effective de l'accroissement escompté des performances. Alors que l'on sait construire des machines parallèles comportant plusieurs milliers de processeurs élémentaires, et par conséquent une forte puissance théorique, l'offre logicielle n'est pas capable de fournir au programmeur un atelier dans lequel les capacités de la machine puissent être exploitées pleinement.

C'est dans ce cadre que de nombreux chercheurs évoluent dans le but d'aboutir à un ensemble matériel et logiciel cohérent. Pour ce faire, il faut répondre aux questions de base découlant directement de l'introduction du parallélisme dans une architecture de machine :

- comment construire une machine plus puissante?
 - comment intégrer plus de processeurs?
 - comment intégrer des processeurs plus puissants?
 - comment organiser les processeurs à l'intérieur de cette machine?
 - par quel moyen faire communiquer les processeurs entre eux?
 - comment organiser la mémoire de cette machine pour permettre de minimiser les temps d'accès?
- comment exploiter la puissance d'une machine?
 - faut-il écrire un programme pour chaque processeur?
 - comment un programme peut-il être réparti sur les processeurs?

- comment rendre un programme indépendant du nombre de processeurs et de leur organisation ?
- quelles sont les informations qu'un programmeur est capable d'exprimer pour faciliter l'obtention de bonnes performances ?
- comment programmer une application pour permettre le parallélisme effectif ?
- quel modèle de programmation adopter ?
- quel langage ?
- comment compiler le langage pour l'exécuter sur machine parallèle ?

Nous ne prétendons pas répondre à toutes ces questions. Notre intérêt s'est porté sur le coté logiciel du parallélisme : étant donné les machines les plus performantes actuellement construites, quelles solutions proposer au programmeur pour rentabiliser son investissement ? Peut-on construire un environnement et un modèle de programmation qui permette à un scientifique de s'abstraire des problèmes directement issus du parallélisme ?

Notre étude sera décrite progressivement. Dans un premier chapitre, nous retracerons l'histoire du calcul scientifique. Pour cela, nous parlerons des machines inventées, étudiées et parfois construites. Nous verrons apparaître les différents types d'architectures qui organisent les processeurs entre eux. Au delà des machines parallèles, nous porterons notre attention sur les outils logiciels proposés, en particulier sur Fortran qui fut le premier langage de programmation de l'histoire. Conjointement, cette étude historique nous permettra de présenter le modèle à parallélisme de données comme un concept latent à l'évolution de l'informatique depuis la préhistoire de cette science (il y a bien 50 ans).

Dans le second chapitre, nous exposerons les travaux existants dans le domaine du parallélisme de données dont le principe est lui aussi très simple : puisque généralement chaque donnée à traiter représente une petite partie de l'état global d'un système, les traitements à appliquer sont équivalents sur chacune des données ; il suffit donc d'associer un processeur à une donnée et d'effectuer le travail en parallèle sur chaque processeur. Le contrôle et l'organisation du travail est ainsi facilitée : le parallélisme est exprimé au niveau des instructions, mais le déroulement séquentiel du programme est conservé. Nous caractériserons ce modèle de programmation par la présentation des problèmes de base relatifs à l'appréhension de ce type de parallélisme, et en particulier à l'expression du parallélisme dans les langages, à la compilation de ces langages et au placement des données.

Notre proposition sera présentée en fin de second chapitre. Nous y appréhenderons le modèle HELP, point central de cette thèse. Le troisième chapitre renferme une description du langage C-HELP qui met en œuvre le modèle HELP. Ces propositions ont été dessinées dans le but de proposer au programmeur un outil proche de la pensée algorithmique.

Le quatrième chapitre nous permettra de montrer la rationalité de nos propositions conceptuelles par l'étude de la compilation du langage. Nous décrirons le compilateur développé durant cette thèse et les points originaux qui y sont intégrés. Nous illustrerons les avantages de ces techniques de génération de code par des mesures de performances obtenues sur une machine *Massivement Parallèle*. Nous montrerons ainsi que ce choix peut être compatible avec l'obtention d'un modèle efficacement compilable, et efficacement exécutable.

Au cours du cinquième chapitre, nous étudierons l'ouverture de l'environnement vers la gestion des matrices creuses. Ces matrices sont rencontrées dans certains domaines scientifiques, en sortie de manipulations diverses qui fournissent de très nombreuses données dont une faible proportion a réellement de l'importance vis-à-vis du phénomène étudié. En conséquence, les données comportent une grande proportion de zéros, et sont de trop grande taille pour pouvoir être traitées directement comme toute autre matrice. Le calcul creux consiste à s'abstraire des valeurs nulles et transcrire le traitement à effectuer vers les compressions des données de départ ne comportant que les éléments significatifs. En ne traitant pas les zéros, on gagne de la place en mémoire et on évite les temps de calculs inutiles (sur les valeurs non-significatives). Le calcul creux permet donc aussi de réduire le temps global de traitement de la matrice.

Le dernier chapitre sera consacré à des exemples développés en C-HELP pour montrer l'adéquation de ce langage au domaine de l'informatique scientifique. Nous avons choisi dans un premier temps des exemples simples permettant de traiter des cas rudimentaires d'algèbre linéaire, comme une méthode d'inversion de matrice. Nous détaillerons un exemple d'application réelle qui met en œuvre un modèle de simulation hydro-dynamique de calcul de courants côtiers.

Nous résumerons enfin les travaux qui ont amené ce mémoire à terme et nous ouvrirons quelques pistes vers l'élargissement de nos propositions à la gestion d'objets plus généraux, dans le but d'étendre le spectre des applications supportées. En route vers le *Yota-flops*¹...

1. Téra-Téra-flops

Chapitre I

Calcul scientifique et parallélisme

1 De l'histoire de l'informatique

L'HISTOIRE du traitement de l'information ne se décline que sur une échelle de temps réduite aux dernières décennies. L'*informatique* est apparue avec les premières machines programmables, imaginées par Charles Babbage au début du XIX^e comme une rencontre entre un métier à tisser de Jacquard, et le calcul en numération binaire de Leibniz [Lig87]; les premières machines seront construites à partir des années 1940.

À l'intérieur du laps de temps réduit entre cette date et nos jours, il est possible de discerner plusieurs ères de technologies propres, régulièrement révolutionnées par de nouvelles inventions qui ont toutes en commun le fait de proposer une innovation *matérielle*. Les recherches effectuées dans le domaine du matériel ont mobilisé les énergies et, résultat très honorable, eu des répercussions importantes à la fois vis-à-vis des performances de nos ordinateurs actuels, mais aussi sur la fréquence élevée d'apparition de nouvelles machines. En contrepartie, ce renouvellement constant de l'offre rend inévitablement obsolète l'avant-dernière machine « la plus performante en date ».

De ces évolutions matérielles de l'informatique, il est possible de distinguer deux types. Le premier type d'évolution, certainement à l'origine de nombreux bouleversements, est d'ordre *technologique*. Dans la courte histoire, c'est de la technologie que sont apparus les avancées les plus spectaculaires : la lampe, le transistor, les mémoires à accès direct, le circuit intégré, les réseaux haut-débit, la fibre optique... Ces nouveautés améliorent toujours sensiblement les performances des machines en terme de temps d'exécution, mais ne bouleversent pas leur fonctionnement de base. Le second type d'évolution matérielle regroupe les avancées de l'*architecture* des machines. Souvent issus de la communauté des informaticiens, plutôt que des électroniciens, ces idées utilisent les avancées technologiques mais proposent régulièrement de nouvelles organisations de machines, indépendantes de la technologie de mise en œuvre. Ainsi,

on a vu apparaître le processeur à calculs flottants, les machines non-*von Neumann* (cf *infra*), les processeurs RISC, les mémoires caches... Bien que parfois moins spectaculaires en terme de retombées immédiates, ces avancées ont l'avantage de perdurer au-delà des générations technologiques, et deviennent alors reconnues et largement adoptées. C'est de ces nouveaux concepts que viennent les problèmes d'exploitation du matériel par le logiciel, du fait de leur indispensable prise en compte dans le développement des outils proposés au programmeur ou à l'utilisateur.

Les évolutions de la partie *logicielle* des systèmes d'information n'ont jamais connu l'incidence des métamorphoses matérielles de l'informatique. Alors qu'un nouveau concept *technologique* est rapidement généralisé par tous les constructeurs qui abandonnent de ce fait le précédent, certains outils logiciels passent les âges et demeurent même au delà des nouveautés architecturales du matériel. À ce phénomène, plusieurs raisons : les acquis dans le développement d'applications importantes ne permettent pas de remettre en cause le travail effectué et difficilement reproductible ; et l'apparition du concept logiciel demande à l'utilisateur (ou au programmeur) une adaptation de ses connaissances et de ses capacités, à la différence du passage de la machine N à la machine $N + 1$ qui (ne) lui demande (qu') une rallonge budgétaire.

Alors que l'évolution matérielle est difficilement prévisible, brutale et sans retenue, l'évolution logicielle est un phénomène souvent lent, rarement spectaculaire et toujours discuté. La dernière génération de machines, avec ses performances annoncées, semble rendre son acquisition indispensable à l'utilisateur pour « garder le contact » ; tandis que la dernière version de son logiciel sera abordée avec plus de retenue, du fait de l'importante quantité d'apprentissage qu'elle requiert par rapport au bénéfice immédiatement escompté.

Le calcul scientifique est à l'image de l'informatique. Les évolutions technologiques y sont capitales. La course à la puissance de calcul impose à tout constructeur de proposer constamment de nouvelles machines (parfois même avant la livraison des machines de la génération précédente). Nombreux sont ceux qui ont échoué (ou échoueront ?) après la commercialisation de leur première ou seconde machine.

Indépendamment de ces progrès matériels, de gros efforts sont accomplis dans le domaine des outils logiciels dédiés au calcul scientifique. Les travaux fondamentaux sur les langages et sur les méthodes de compilation visent à outrepasser les bouleversements architecturaux, ou du moins à s'en accommoder, par la définition de concepts indépendants des machines cibles. Ainsi, beaucoup de recherches ont comme point de départ le langage Fortran, utilisé par les numériciens, et proposent des aides à l'obtention de performances sur les nouvelles architectures par la mise au point d'outils de parallélisation automatique ou par la définition de nouveaux « concepts », souvent intégrés aux langages existants, permettant l'adaptation à faible coût des algorithmes ou des programmes déjà existants à une nouvelle architecture de

machine.

Dans ce chapitre, nous allons parcourir succinctement l'histoire du développement du matériel dédié au calcul scientifique, en observant les passages importants d'un concept architectural à son successeur. Nous montrerons les choix capitaux opérés par les divers constructeurs pour la conception de leurs machines. Nous présenterons ensuite brièvement l'état actuel des possibilités du calcul scientifique par la présentation des caractéristiques principales des machines massivement parallèles actuellement commercialisées et répertoriées comme faisant partie des plus performantes, en terme de puissance de calcul.

Nous nous intéresserons conjointement au côté logiciel : outre la rémanence de Fortran, nous mettrons en valeur l'existence d'un concept apparu très tôt qui persiste : le modèle à *parallélisme de données*.

Enfin, en se penchant sur le « *Grand Challenge* » [HPCC94], nous essaierons de montrer que le parallélisme de données est probablement le meilleur modèle de programmation candidat à la résolution des problèmes reconnus pour être les plus exigeants quant à la quantité requise de calculs et d'occupation mémoire.

2 Les modèles d'exécution

2.1 Le modèle *von Neumann*

JOHN VON NEUMANN a décrit en 1947 le principe de base d'une architecture d'ordinateur avec l'introduction de l'idée de mémoire banalisée. La mémoire de la machine contient aussi bien les données à traiter que le programme qui effectue les traitements. L'usage courant associe maintenant son nom au modèle d'exécution des machines comportant un unique processeur scalaire. Le déroulement séquentiel du programme est effectué par le traitement une à une des instructions en suivant un cycle de 5 étapes : lecture de l'instruction en mémoire, décodage de cette instruction, lecture des opérandes, traitement calculatoire et rangement du résultat.

Flynn a proposé à partir de 1966 (puis dans [Fly72]) une classification des architectures suivant leur modèle d'exécution. Le modèle *von Neumann* est repris sous l'appellation *Single Instruction stream, Single Data stream*¹. En effet, le modèle implique qu'à un instant donné, la machine traite une instruction sur une donnée ; d'où l'appellation courante de processeur séquentiel. Cette instance de la classification de Flynn est couramment schématisée par la figure I.1.

1. « *flot unique d'instructions, flot unique de données* »

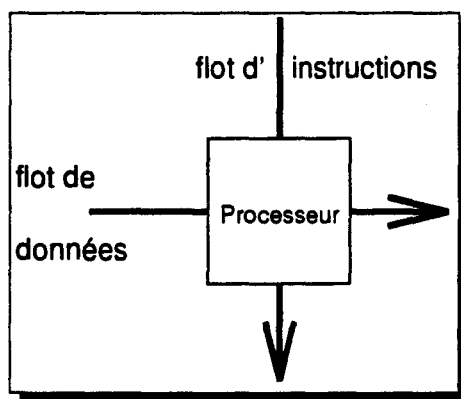


FIG. I.1 - *Le modèle d'exécution SISD*

L'énorme majorité des ordinateurs construits depuis l'origine des temps informatiques possèdent cette architecture simple. Le concept *von Neumann* est donc largement diffusé : un unique processeur de calcul est connecté à une unique mémoire. Ce modèle est accepté par tous et répond facilement aux attentes des utilisateurs [TW91], par l'adéquation du raisonnement algorithmique simple à l'ordonnancement linéaire des étapes de calculs.

Toutefois, les besoins de puissance de calcul pour les applications scientifiques sont restés insatisfaits par ce modèle d'exécution. Le premier de ces calculateurs séquentiels dédiés au calcul flottant, proposé par Gene Amdahl d'IBM en 1955, dispensait une puissance d'environ 5 kilo-flops. Les progrès technologiques ont certes largement amélioré ces performances, mais rares sont les architectures d'aujourd'hui qui conservent ce modèle de fonctionnement.

Parallèlement aux développements technologiques, les langages de programmation apparaissent. Le langage machine devient rapidement trop contraignant pour le programmeur. Sous le terme de « *automatic programming* »², les langages de programmation, plus ou moins structurés, en particulier les premières versions de Fortran inaugurent l'abstraction des caractéristiques du matériel dans la programmation.

La notion de **tableau** apparaît en même temps que ces langages (cf. extraits de « *original FORTRAN Manual* » dans [Bac78]). Un tableau représente un ensemble de données homogènes du programme. Le concept de tableau à une dimension, à l'image de la mémoire physique des machines, sera généralisé par la définition de tableaux à plusieurs dimensions, plus proches des algorithmes à mettre en place, et implantés dans la mémoire linéaire sous la responsabilité du compilateur.

Généralement, un tableau est traité par un parcours séquentiel de ses éléments, accompagné d'un traitement itératif sur l'élément courant. Dès lors, un élément du tableau su-

² *programmation automatique*

bit le même traitement qu'un autre élément. La factorisation des traitements à appliquer à chaque élément a pour conséquence d'introduire dans la programmation des structures de contrôle en boucle, de par le fonctionnement séquentiel de la machine supportant l'exécution du programme. Le fait que ces opérations soient effectuées les unes après les autres, ou *séquentiellement*, ne vient pas toujours du problème à traiter ou de l'algorithme, mais souvent du fonctionnement de la machine suivant le modèle *von Neumann*, repris comme modèle de programmation. Il est peut-être regrettable qu'au moment de la création de ces premiers langages, on ait ainsi intégré indifféremment les conséquences des caractéristiques du matériel et les fonctionnalités relatives à l'expression des algorithmes.

Dès lors, on peut admettre qu'il apparaît qu'un modèle de pensée algorithmique est présent depuis l'apparition des langages de programmation, et qui consiste à effectuer un même traitement sur plusieurs données de même type. Sans le caractériser plus précisément pour l'instant, nous l'appelons dès maintenant le *parallélisme de données*.

Un langage de programmation doit inévitablement amener les concepteurs à produire un compilateur. Au début de l'année 1955, une première version d'un compilateur Fortran est développée. Conjointement on commence déjà à se pencher sur les problèmes d'optimisation du code généré à partir de ce langage (en particulier, les problèmes d'accès aux tableaux bi-dimensionnels) [Bac78].

À cette époque apparaissent certains problèmes fondamentaux qui seront à la base de nombreux travaux pour leur résolution dans un contexte non séquentiel. Ils sont issus du regroupement des données dans un tableau. Par exemple, le problème des dépendances de données existe déjà. En effet, la figure I.2 présente dans sa partie gauche un programme qui a pour objet un décalage des éléments d'un vecteur A.

<p>C programme faux REAL A(100)</p> <p>DO 10 I = 2,100 10 A(I)=A(I-1)</p>	<p>C correction C sur le temps REAL A(100)</p> <p>DO 10 I = 100,2,-1 10 A(I)=A(I-1)</p>	<p>C correction C sur l'espace REAL A(100),B(100)</p> <p>DO 10 I = 2,100 10 B(I)=A(I-1) DO 20 I = 2,100 20 A(I)=B(I)</p>
---	---	--

FIG. I.2 - Le problème des dépendances

L'ordre de parcours du tableau est primordial, le programme montré en partie gauche de la figure ne donne pas le résultat voulu. Dans le contexte d'exécution (et de programma-

tion) séquentielle, on parlera d'erreur du programmeur. Toutefois, un décalage est bien une opération qui consiste à affecter la valeur d'un élément à celle de son voisin. Indépendamment de l'ordre d'exécution, l'expression qui apparaît à la ligne 10 du programme est donc intrinsèquement valide. Par conséquent, c'est l'interférence entre la notion de tableau et la boucle d'itération parcourant ce tableau, qui induit ce problème. Dès lors, le programmeur doit prendre en compte ces comportements inattendus et considérer une des deux notions comme imposée par l'autre. Il pourra ainsi corriger le déroulement de la boucle en modifiant le parcours des éléments (correction sur le temps) ou alors, garder le même déroulement mais changer l'allocation de ses objets pour la rendre compatible avec son parcours (correction sur l'espace).

Rapidement, les problèmes scientifiques complexes nécessiteront des performances qui ne seraient pas atteintes, malgré les progrès technologiques, en suivant le modèle *von Neumann*. En effet, au fur et à mesure des progressions de la puissance des processeurs, les utilisateurs (principalement physiciens) apparaissent plus exigeants quant à la quantité des calculs à effectuer dans un temps toujours trop important par rapport aux besoins.

Au delà de ces progrès, la limite des contraintes physiques de la technologie (fréquence horloge, taux d'intégration...) impose aux processeurs scalaires une limite de performances. Alors que l'on a assisté pendant plusieurs décennies au doublement régulier des performances, ce rythme de croissance ne pourra pas être maintenu indéfiniment, en restant dans le schéma d'un processeur unique, basé sur l'intégration croissante de semi-conducteurs. À défaut d'une révolution technologique assez improbable dans un proche avenir, comme l'utilisation du photon plutôt que de l'électron, une barrière de puissance sera inévitablement atteinte d'ici peu.

2.2 Le modèle pipe-line

L'idée du traitement pipe-line est de découper un traitement en étapes successives dans le but d'associer à chaque étape une unité indépendante. Les analogies avec des organisations collectives de la vie courante sont nombreuses ; le chaînage de tâches effectuées indépendamment sur un certain nombre de postes de travail permet à la fois de réduire les exigences de capacité des acteurs et d'augmenter la productivité.

Une machine (ou un processeur) vectorielle est un exemple d'application de ce système d'organisation. Le principe est d'éclater les traitements de l'unité arithmétique et logique en plusieurs étapes. Ainsi, une unité vectorielle comporte plusieurs unités de calcul (appelées *étages*) qui sont chaînées et les valeurs en entrée subissent successivement plusieurs sous-traitements, dont la séquence est équivalente au traitement global voulu. Le gain de performances est dû au fait que lorsqu'un des étages a terminé le traitement d'un élément, il peut

commencer le traitement de l'élément suivant. Avec ce modèle, le taylorisme fait son entrée dans l'architecture des ordinateurs.

Ce modèle d'architecture fait l'objet d'une discussion qui vise sa classification dans le cadre de Flynn [Ker92]. Certains pensent que la classification de Flynn se basant sur le niveau le plus bas, c'est-à-dire au niveau du matériel, il faut classer ce type d'architecture dans la catégorie MISD³. En effet, comme le montre la figure I.3, en considérant une vue atomique sur le modèle d'exécution, on remarque qu'une donnée est traitée successivement par plusieurs processeurs qui exécutent chacun un traitement différent. On a donc bien un seul flot de données et plusieurs flots d'instructions.

En considérant la machine vectorielle au niveau supérieur, on peut considérer que le modèle SISD est concordant avec le modèle des processeurs vectoriels car il n'existe toujours qu'un flot unique d'instructions qui sont décomposées de façon cachée au programmeur. L'obtention finale de plusieurs flots d'instructions est en effet à la charge du compilateur ou même de l'architecture.

Enfin, un troisième classement est possible : SIMD⁴ [Hwa93]. Cette vue, à la différence des précédentes, considère le pipe-line du point de vue du programmeur. Celui-ci n'explicite qu'une instruction (qui sera décomposée par la suite) qui s'applique sur un flot comportant plusieurs données. À un instant donné, le modèle décrit effectivement un fonctionnement dans lequel plusieurs données sont traitées (une par étage du pipe-line) par un même flot d'instructions (mais par des instructions atomiques différentes).

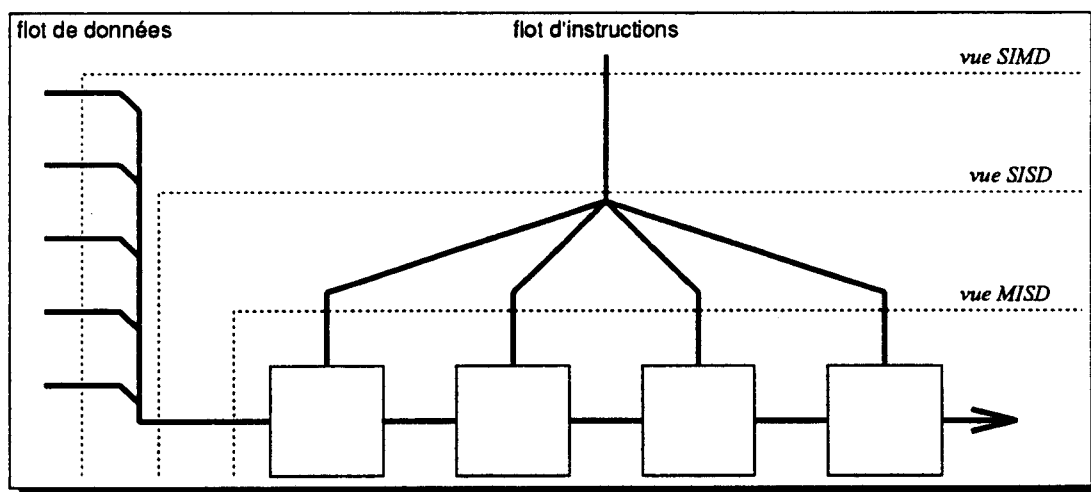


FIG. I.3 - Le modèle d'exécution pipe-line

3. « multiples flots d'instructions, flot unique de données »

4. « simple flot d'instructions, multiples flots de données »

Devant ces points de vues différents, mais tous justifiés, nous éviterons donc d'appliquer la classification de Flynn à ces architectures, que nous appellerons simplement, et dans un esprit de consensus, les architectures vectorielles.

Des exemples simples d'unités vectorielles sont généralisés dans de nombreux processeurs : l'addition et la multiplication des nombres flottants sont par exemple décomposées en quatre étapes chacune suivant le modèle de la figure I.4.

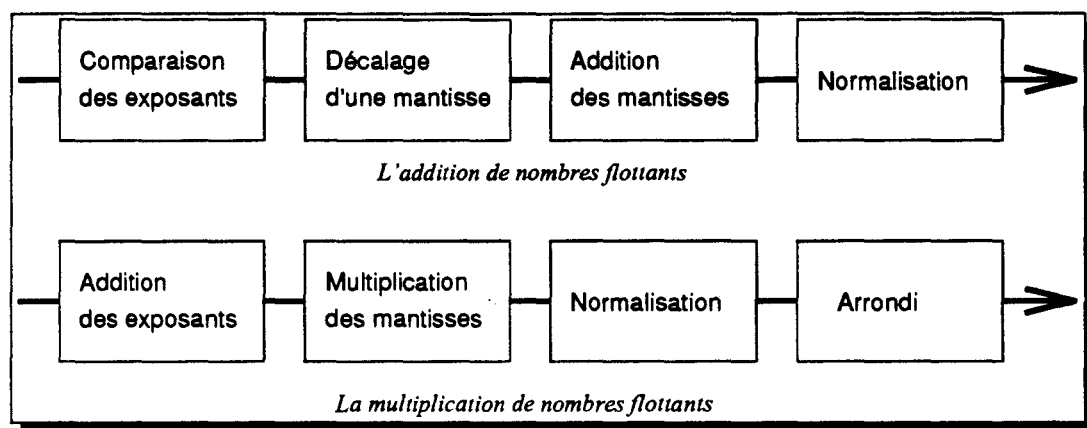


FIG. I.4 - Deux exemples de pipe-lines

Adoptée par les constructeurs de super-calculateurs, en particulier par Cray, Control-Data, et les constructeurs japonais, ce modèle architectural va se généraliser rapidement dans le monde des super-calculateurs. La totalité des super-calculateurs commercialisés jusque la fin des années 80 vont adopter ce modèle d'exécution. Aujourd'hui encore, nombreuses sont les machines en exploitation à suivre ce modèle d'exécution (cf. section 3). C'est la première révolution conceptuelle du matériel dédié au calcul scientifique.

L'exploitation de ces architectures amène le programmeur à faire face à de nouveaux problèmes. Les gains de performances ne sont obtenus qu'à partir du moment où il existe suffisamment de données pour que le temps de chargement et de déchargement des différents étages soit minimisé par rapport au temps total de calcul. Plus les données sont de grandes taille, plus le gain obtenu s'approche du maximal théorique. Lorsque les données sont considérées de taille infinie, le temps moyens de calcul pour un élément est divisé par le nombre d'étages, à condition de décomposer le traitement en portions de temps de traitement équivalentes sur chaque étage.

Du point de vue du programmeur, le modèle des données homogènes regroupées dans une même structure qui fera office de flot d'entrée pour le pipe-line devient impératif. Une architecture vectorielle ne traite efficacement que les vecteurs ! Les tableaux du langage For-

tran, à l'image des vecteurs d'entrée, correspondent à cet impératif. Le langage Fortran est alors universellement adopté par les constructeurs de ces machines. Fortran a ainsi survécu à l'apparition des architectures vectorielles, et devient le langage universel du calcul scientifique.

Cependant, quelques ré-aménagements ont été reconnus indispensables, pour permettre l'exploitation maximale de la machine cible. Par exemple, l'introduction de ce genre de parallélisme implique la prise en compte impérative du problème des dépendances de données au niveau du compilateur, contrairement au modèle séquentiel qui le laissait à la charge du programmeur.

Une donnée modifiée par l'algorithme en cours d'exécution ne peut servir au calcul d'une donnée suivante qu'à partir de sa « sortie » du dernier étage. Dès que l'ordonnement des éléments dans le flux de données est incompatible avec le traitement voulu, le chaînage des unités doit être rompu en attendant la sortie d'une valeur à réutiliser. Le rythme de fonctionnement optimal est alors perdu et les performances chutent dramatiquement. Pour pallier cet effondrement, les langages vectoriels proposent des directives de compilation qui permettent au programmeur de forcer le fonctionnement pipe-line, en indiquant qu'il assure que le flot d'entrée est compatible avec le traitement effectué dans le pipe-line.

D'importants travaux visent à soulager le programmeur de ce travail supplémentaire par une phase de pré-compilation qui permet d'extraire du code Fortran les informations nécessaires à la génération d'un exécutable comportant une gestion pipe-line sans interruption. Ces informations sont parfois très difficiles, voire impossibles à retrouver dans un code source, car souvent dynamiques. Le manque d'informations lors de la compilation, en particulier des domaines de variations des variables pendant l'exécution, ne permet pas toujours d'assurer la continuité du flot de données. Le programmeur doit donc intervenir sur la phase de génération de code par l'intermédiaire de directives autorisant le compilateur à considérer le flot de données comme satisfaisant aux conditions requises, assumant encore une part prépondérante dans l'écriture d'un programme efficace. De plus, certaines optimisations sont possibles à condition de connaître certains paramètres matériels (entre autres la taille des registres vectoriels).

Le modèle de programmation n'est pas modifié et consiste toujours à appliquer un traitement identique sur des données différentes ; une architecture vectorielle est donc programmée suivant le modèle à parallélisme de données. C'est l'introduction du parallélisme dans le modèle d'exécution qui induit les problèmes d'écriture et de compilation efficace du code. En reprenant le cas d'école de la figure I.2, on s'aperçoit bien que c'est la dimension temporelle qui devient primordiale : une dépendance de données ne doit pas « remonter » le déroulement du temps établi par l'ordre d'entrée des données dans le pipe-line.

2.3 Le modèle parallèle synchrone

2.3.1 Modèle SIMD

Parallèlement au développement effectif des machines vectorielles, une autre voie d'étude est explorée dans le but d'obtenir de fortes puissances de calcul. On s'oriente vers un autre type de parallélisme, déjà évoqué en 1958 : celui que Flynn appelle SIMD.

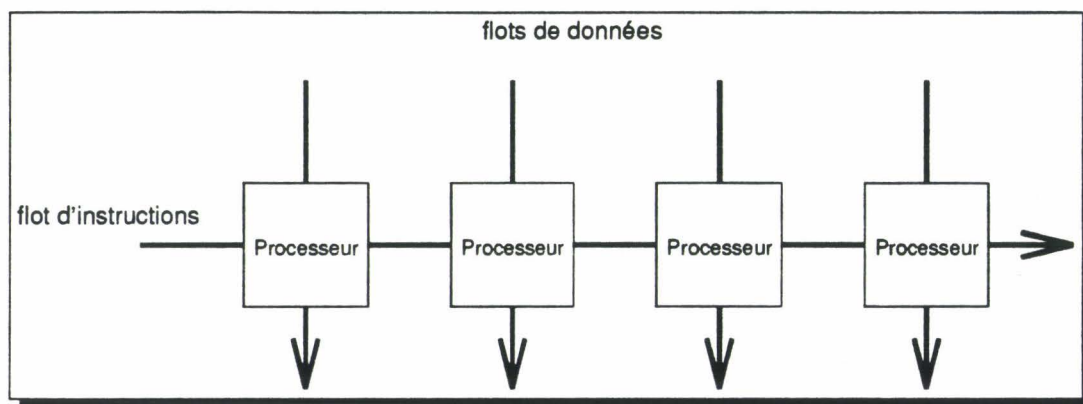


FIG. I.5 - *Le modèle d'exécution SIMD*

Ce modèle d'exécution, aussi appelé *modèle synchrone*, consiste à exécuter à un même instant un flot d'instructions unique, sur un certain nombre de données différentes. Le taux de parallélisme induit par ce fonctionnement est égal au nombre d'unités de traitement de la machine.

Chacune de ces données est traitée par un processeur dont l'architecture peut être simplifiée, du fait de l'absence de gestion des instructions par une unité de commande. Le programme à exécuter est chargé dans un processeur hôte qui pilote l'ensemble des processeurs élémentaires par diffusion des instructions, qui sont généralement décodées et envoyées sous forme de micro-code (cas de la MasPar [Mas90]) ou d'appel à des primitives de base (cas de la CM-2 [Thi89]).

Cette simplification permet une intégration plus importante qui autorise la présence au sein de ces machines d'un nombre très important de processeurs élémentaires. De par cette intégration possible, et conjointement aux progrès technologiques, sont apparues les premières machines pouvant comporter jusqu'à plusieurs centaines, voir milliers de processeurs. Cette nouvelle génération d'architectures est couramment identifiée sous le nom de machines « massivement » parallèles ; elles sont le point de départ de la deuxième révolution des supercalculateurs.

2.3.2 Les machines synchrones

L'idée de coupler plusieurs unités de calcul est apparue très tôt dans l'histoire. Par exemple, dès 1958, deux machines sont apparues dans les travaux de recherches : une machine dédiée à l'étude de problèmes discrets à deux dimensions [Ung58], et SOLOMON [BBJ⁺62, SBM62] conçue par Daniel Slotnick. Les impératifs de construction et l'avancement des connaissances technologiques n'ont pas permis, à cette époque, de lancer l'exploitation effective de cette idée. La machine SOLOMON, pourtant annonciatrice d'un important bouleversement architectural (elle était prévue pour contenir 1024 processeurs) sera spécifiée mais ne sera jamais construite.

Dans les années qui suivirent, plusieurs projets reprenant le modèle SIMD sont étudiés, et aboutissent à la construction de machines expérimentales. Citons ILLIAC-IV (toujours proposée par Slotnick [BBK⁺68] en 1968), STARAN (dédiée au traitement de signal, 1970), PEPE (1975) et DAP (avec ses arithmétiques variées, 1976). On peut remarquer que toutes ces machines possèdent une topologie *tableau*. Les puissances de ces machines ne permettent pas de rivaliser avec les calculateurs vectoriels, mais ont l'avantage de montrer par anticipation les problèmes d'exploitation du parallélisme des architectures SIMD.

La première machine massivement parallèle suffisamment généraliste et performante pour concurrencer le monopole établi du calcul vectoriel, et par conséquent pour être commercialisée largement, arrive dans la fin des années 80 : c'est la CM-2 de chez Thinking Machine Corporation, successeur de la CM-1 [Hil85, KH89, Thi90]. Les processeurs élémentaires sont au nombre maximal de 65 536, ils ont une architecture limitée au traitement de données codées sur 1 bit [DKV88], le réseau de communication entre les processeurs est un hyper-cube.

L'organisation mémoire devient une caractéristique importante des machines. On ne peut, à l'époque de la construction de ces machines, proposer une mémoire accessible à tous les processeurs. La mémoire totale de la machine est répartie entre les nœuds (processeurs) qui ont le privilège exclusif des accès à chaque partie de mémoire. Une adresse mémoire est donc composée d'un numéro de processeur et d'une adresse locale. Pour qu'un processeur accède à la mémoire réservée d'un autre processeur, il est impératif d'opérer des communications sur le réseau de données. Ces accès distants, largement plus coûteux en terme de temps d'accès, sont à la base de problèmes importants de programmation efficace sur architectures parallèles : il faut favoriser la localité des données par l'allocation des données traitées par un processeur dans la mémoire locale au processeur.

L'arrivée de cette machine sur le marché est accompagnée par une offre logicielle importante capable de séduire de nombreux scientifiques. On assiste alors au déclenchement d'un subit engouement de la part de nombreux laboratoires scientifiques pour ce nouveau type prometteur de parallélisme. La communauté informatique est amenée à résoudre les nouveaux

problèmes posés par ces architectures, nombreuses étant les applications réelles qui doivent être totalement réécrites pour atteindre des performances honorables sur CM-2, démontrant ainsi le besoin flagrant de nouveaux outils de programmation parallèle.

Les portes ouvertes par la construction effective de ce nouveau type d'architectures couvrent un éventail élargi qui va du développement des applications scientifiques, à la modélisation théorique du parallélisme synchrone, en passant par les langages de programmation.

2.3.3 Une philosophie nouvelle

Dans les années qui suivent l'apparition de ce type de parallélisme, la philosophie de construction ne consiste plus à obtenir un processeur surpuissant mais à intégrer un maximum de processeurs suffisamment généralistes pour proposer une machine dont la puissance théorique surpasse celle des calculateurs vectoriels. Pour construire une de ces machines, certains choix fondamentaux d'architecture sont possibles (réseau de communications) mais on observe la tendance générale des constructeurs à favoriser la puissance théorique maximale par le nombre plutôt que par la puissance des processeurs.

Le processeur élémentaire est souvent de capacité faible pour permettre une intégration maximale. Il est en effet difficilement concevable de fabriquer une machine qui comporterait une carte par processeur, ce qui lui imposerait d'être construite au-dessus d'un fond de panier pour le moins gigantesque. La solution adoptée consiste alors à implanter plusieurs processeurs élémentaires dans le même circuit intégré, regroupant ainsi plusieurs points vitaux comme l'alimentation du circuit, les chemins de diffusion des instructions, celui des données...

La construction d'une machine synchrone massivement parallèle passe donc impérativement par la définition et la construction d'un processeur élémentaire dédié. C'est sûrement principalement pour cette raison que ce type de machines n'est aujourd'hui plus construit. Nous verrons dans la section suivante le choix plus récent d'intégrer au sein des machines des processeurs courants.

Au-delà du processeur élémentaire, les machines synchrones se différencient par l'organisation globale des processeurs. Il est évident que ceux-ci doivent communiquer entre eux au cours d'un traitement. Entrent alors en jeu les capacités du (ou des) réseau de données de la machine massivement parallèle. Plus la machine offrira la possibilité d'effectuer simultanément un grand nombre de communications de processeur à processeur, plus le temps global dédiée aux communications sera réduit. L'influence néfaste de ces communications quant à la puissance effective d'une machine s'en trouve diminuée.

Le nombre de liens de communication reliés à chaque processeur élémentaire et la géométrie mise en place par ces liens définissent la *topologie* de la machine, qui est souvent le critère

de base pour distinguer les différentes architectures SIMD.

La topologie idéale est un réseau complètement connecté dans lequel tout processeur peut communiquer directement avec n'importe quel autre processeur. Pour une machine comportant n processeurs élémentaires, le nombre de liens bi-directionnels nécessaires est donc égal à $n(n-1)/2$. Il est bien évident qu'il n'est pas possible d'intégrer dans ces conditions un grand nombre de processeurs.

La solution qui tend actuellement à se répandre (pour les machines massivement parallèles synchrones comme pour les machines asynchrones) est d'organiser la machine en grille torique à deux ou trois dimensions, comme les machines CRAY-T3D [Oed93], Intel PARAGON [Int91], MasPar MP-1 et MP-2 [Bla90]... Ces topologies dispensent une bande passante acceptable et possèdent la bonne propriété d'être extensibles. En effet, l'ajout de processeurs supplémentaires se fait par l'augmentation de la taille de la machine suivant une ou deux dimensions, sans changer la connectivité des processeurs déjà présents. Cette propriété n'est pas retrouvée pour un hyper-cube.

2.3.4 L'exploitation logicielle

À chaque opération calculatoire, le parallélisme SIMD fournit une accélération théorique égale au nombre de processeurs. Cependant, le déroulement du programme nécessite certaines opérations indépendantes du flot parallèle de données mais qui traitent les variables de contrôle du programme. Ces variables (comme les variables d'itération de boucles) sont gérées par un processeur de type *von Neumann*⁵; le temps d'exécution de ces instructions est donc équivalent à celui d'un processeur séquentiel. De plus, ces machines proposent impérativement un mécanisme d'inhibition de certains processeurs dans le but de contrôler l'activité de calcul sur certaines valeurs des flux d'entrée suivant l'algorithme à appliquer⁶, réduisant ainsi le degré de parallélisme du programme. Enfin, certains calculs (comme le tri, la somme des éléments...) nécessitent des communications entre les processeurs, opérations généralement coûteuses. Le gain effectif apporté par ce modèle d'architectures est donc toujours nettement inférieur au nombre de processeurs. Pour tenter de réduire l'écart constaté entre la puissance théorique et la puissance effective d'une machine, il est d'une importance primordiale de proposer des outils performants de génération de code.

La programmation des machines synchrones comporte plusieurs points communs remarquables avec la programmation de machines vectorielles. Les problèmes d'accès mémoire ou de dépendance de données s'y retrouvent. Par exemple, dans le cas d'école du décalage des élé-

5. Processeur qualifié de *processeur hôte*, ou de *séquenceur*.

6. Par exemple, lors d'une division, on inhibera les processeurs qui possèdent dans le flux d'entrée un diviseur dont la valeur est nulle.

ments d'un vecteur, la dépendance qu'il existe entre l'affectation d'un élément et celle de son voisin dans le cas du traitement vectoriel, se retrouve au niveau des processeurs élémentaires. En effet, si deux éléments se trouvent projetés sur deux processeurs différents, le décalage temporel nécessaire entre la lecture et l'écriture d'un élément se retrouve au niveau du besoin de communications entre les deux processeurs. Tout processeur devra envoyer la valeur de son élément à son voisin (ou recevoir de son voisin) avant de traiter leur affectation. Dans le cas où le nombre de processeurs physiques permet d'allouer un élément par processeur, la boucle de décalage pourra être traduite et exécutée en un cycle : communications puis traitement.

Le décalage historique important entre le début de la généralisation du Fortran et la diffusion des machines massivement parallèles a sûrement été à l'origine de l'utilisation d'un tel langage sur ce nouveau type d'architectures. Les nombreux développements d'applications scientifiques intensives effectués en Fortran sur machines vectorielles doivent s'adapter facilement sur ces nouvelles architectures. On comprend qu'un physicien qui a supporté Fortran depuis son premier développement d'application n'en reste pas totalement indemne, et revendique la possibilité d'atteindre plus de performances par un changement de machine qui ne compromet pas ses travaux antécédents. Fortran accède alors à un statut de langage intouchable, on le retrouve sur quasiment toutes les machines massivement parallèles synchrones.

Comme pour l'adaptation au vectoriel, l'univers parallèle devra amener à Fortran quelques directives supplémentaires, prenant en compte les nouveaux problèmes spécifiques comme le placement des données et les optimisations de temps de communication⁷.

Ainsi, le modèle de programmation sous-jacent au traitement des données homogènes par plusieurs processeurs est commun aux deux modèles d'exécution, correspondant aux deux modèles d'architecture (vectorielle ou parallèle). Le parallélisme de données est bien un modèle indépendant de l'architecture cible, seule la phase de compilation varie d'un modèle d'exécution à l'autre.

2.4 Le modèle parallèle asynchrone

2.4.1 Modèle MIMD

Plus récemment, les machines massivement parallèles sont construites suivant un autre mode de fonctionnement que Flynn appelle : *Multiple Instruction stream, Multiple Data stream*⁸. Schéma en figure I.6.

7. À noter tout de même l'adoption d'une instruction déjà présente dans d'autres extensions data-parallèles de Fortran : `forall`. Nous la détaillerons dans la description d'HPF (cf 2.1)

8. « *multiples flots d'instructions, multiples flots de données »*

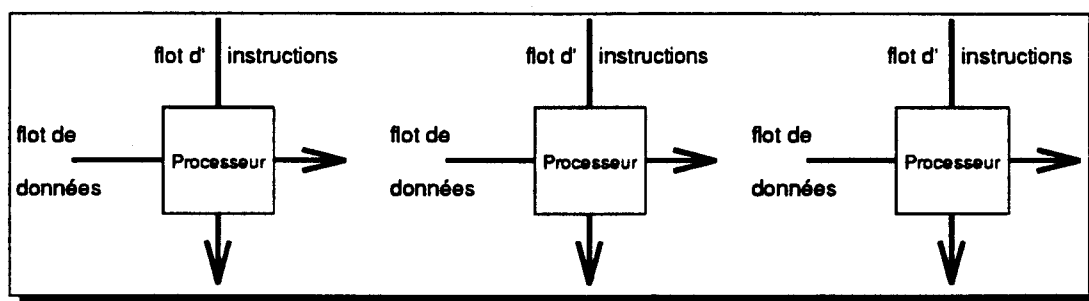


FIG. I.6 - *Le modèle d'exécution MIMD*

Ce modèle de fonctionnement est déjà établi depuis plusieurs années, mais les machines reposant sur ce principe ne pouvaient pas être qualifiées de « massivement » parallèles, elles comportaient généralement un nombre moins importants de processeurs.

Aujourd'hui, les progrès technologiques ont permis l'avènement de machines comportant un nombre de processeurs beaucoup plus important, et toute nouvelle machine massivement parallèle suit ce modèle d'exécution. L'adoption de ce type d'architecture dans le développement peut s'expliquer d'une part du point de vue de la construction, et d'autre part du point de vue de la programmation.

2.4.2 Les machines asynchrones

La définition et la construction d'un processeur performant est un travail extrêmement lourd, souvent très long, et parfois même de résultat incertain⁹. Son développement dans le but unique de l'intégrer dans une machine parallèle précise ne peut pas permettre à ce jour d'atteindre la rentabilité. Les constructeurs de machines parallèles préfèrent donc, depuis quelques années, réutiliser un processeur existant dans le monde du séquentiel, largement diffusé et qui comporte de ce fait un avenir à beaucoup plus long terme.

L'utilisation de processeurs « scalaires » permet non seulement d'éviter le développement d'un processeur élémentaire dédié à la machine, mais aussi d'exploiter rapidement les progrès technologiques intégrés à ces processeurs. Citons l'exemple récent des processeurs alpha [Alp93a, Alp93b] repris par CRAY dans la machine T3D [Oed93]. Notons enfin, que cette tendance a pour effet d'inciter les concepteurs de processeurs à prendre en compte leur possible intégration dans une machine parallèle par l'ajout de certaines fonctionnalités propres.

Depuis peu, de part l'importante progression technologique appliquée à la construction des

9. Toute ressemblance avec des processeurs...

processeurs et des machines séquentielles, certains constructeurs s'orientent vers la conception de machines parallèles constituées d'un certain nombre de stations de travail. On ne réutilise plus uniquement un processeur, mais une machine séquentielle complète, comme brique de base de machine parallèle. Ces machines, principalement IBM [DD90] et Digital [SKz⁺94], sont couramment appelées *ferme* ou *cluster*. Du fait du nombre de processeurs limité (généralement n'excédant pas quelques dizaines), mais néanmoins puissants, le réseau de communication est de type cross-bar et permet l'obtention d'une bande passante importante qui permet de considérer ces édifices comme de réelles machines parallèles.

Enfin, il est nécessaire de différencier les deux organisations mémoire possibles. Certaines machines associent exclusivement à chaque processeur une partie de la mémoire totale de la machine, on parle alors de mémoire *distribuée*. La difficulté de programmation (et généralement de compilation) de cette organisation réside dans le fait qu'un processeur voulant accéder à la mémoire d'un autre processeur doit impérativement communiquer avec ce second processeur. Le code généré doit alors inclure des synchronisations entre les processeurs lors de chacun de ces accès mémoire distants.

La seconde organisation mémoire ne considère qu'un espace d'adressage pour toute la mémoire de la machine parallèle. Un processeur peut accéder n'importe quel emplacement sans perturber les autres processeurs. Dans ces machines massivement parallèles, les bancs mémoire sont tout de même physiquement répartis à travers le réseau, pour permettre de rendre plus efficaces les accès : la notion de localité peut y être retrouvée et les temps d'accès locaux sont toujours plus optimisés par rapport aux temps que l'on obtiendrait en regroupant la mémoire totale de la machine.

Pour ces machines, le routage des données à travers le réseau est assuré par la partie matérielle. La problématique de la programmation s'en trouve diminuée, mais il n'en est pas forcément de même pour le temps d'accès distant. En effet, il existe toujours une grande différence d'efficacité entre un accès local et un accès distant. On retrouvera inévitablement les problèmes d'optimisation des placements de données sur les différentes mémoires de la machine.

Les deux systèmes de répartition de la mémoire sont aujourd'hui présents dans le parc des machines parallèles. La construction d'une machine à mémoire partagée est plus difficile mais une partie importante du travail de compilation est déléguée au matériel, et par conséquent probablement rendu plus efficace¹⁰.

10. On ne peut pas formellement comparer les deux systèmes : aucune machine ne permet la comparaison exclusive de ce critère.

2.4.3 Programmation des machines MIMD

Une observation possible lors de l'utilisation des machines synchrones est le faible degré de parallélisme observé pour de réelles applications scientifiques. L'algorithmique mise en place n'est en effet pas toujours compatible avec un fonctionnement synchrone. L'idée simple de désynchroniser les processeurs permet d'espérer l'augmentation du taux d'activité des processeurs : un processeur élémentaire inactivé par un constructeur quelconque du flot d'instructions peut exécuter une autre partie du code, sans attendre que le programme soit déroulé jusqu'à sa réactivation.

On arrive ainsi à un modèle d'exécution dit « SPMD »¹¹ (par exemple [LER92]), dans lequel le programmeur n'écrit qu'un programme (une tâche) qui est ensuite exécutée par une machine parallèle. Ce modèle d'exécution est alors théoriquement plus rapide qu'une exécution synchrone. Néanmoins, pour obtenir de meilleures performances, il est nécessaire de disposer de mécanismes de synchronisation rapide des processeurs. À défaut, ces opérations demandent un temps d'exécution qui pénalise ce type d'architectures.

Le nouveau modèle d'exécution n'induit pas forcément un nouveau modèle de programmation. Bien qu'il existe d'autres modèles, comme le développement de tâches indépendantes communiquant par messages¹², les applications numériques suivent souvent le modèle à parallélisme de données : le programmeur considère un ensemble de données sur lesquels il applique un traitement unique. Seul ce modèle de programmation à parallélisme de données permet de développer facilement des applications qui s'exécutent sur un grand nombre de processeurs.

Ainsi, le langage le plus utilisé reste, sans surprise, Fortran. Malgré les problèmes de compilation fortement éloignés de ceux rencontrés pour le modèle d'exécution SIMD, les machines massivement parallèles MIMD sont encore programmées par le modèle à parallélisme de données. Il va de soit que ces problèmes de génération de code ont amené la communauté Fortran à réfléchir sur l'opportunité de se définir de nouveaux concepts de programmation.

Un forum¹³ regroupant les principaux constructeurs et utilisateurs des calculateurs scientifiques s'est formé en ce sens. Un nouveau Fortran a été défini ; ou plutôt, une nouvelle rafale de directives : HPF [For93]. Ce langage est reconnu comme pouvant être candidat à un standard de programmation à parallélisme de données, qu'il soit compilé pour les machines séquentielles, vectorielles ou parallèles. C'est pour cette raison que nous lui consacrerons une part importante de la présentation des langages à parallélisme de données qui existent aujourd'hui (cf. chapitre suivant, 2.1).

11. « *simple programme, multiples flot de données* »

12. Généralement ce modèle de programmation fait interagir des tâche à gros grain, ce qui nécessite souvent autant de développements de codes sources.

13. HPFF : High Performance Fortran Forum [Hpf92]

3 Les super-calculateurs

IL EXISTE des ouvrages qui répertorient les super-calculateurs installés dans le monde et qui proposent souvent un classement de ces machines suivant leurs performances réelles ou théoriques. C'est le cas du TOP500, réalisé à l'université de Mannheim en collaboration avec Jack Dongarra. Ce classement est établi sur la base de résultats aux tests de performances sur l'exécution des programmes LINPACK qui renferment une modélisation des principales opérations numériques régulièrement rencontrées dans les diverses applications¹⁴.

Nous avons synthétisé les derniers résultats en date dans le tableau de la figure I.7. Dans ce récapitulatif de machines ne figurent que des machines parallèles ou vectorielles dont les mesures de performances sont issues de tests effectifs réalisés par les auteurs de ce classement ; cette liste n'a donc aucun caractère exhaustif. On se contente de donner une idée sur le type des machines rencontrées ainsi que sur leur puissance effective (notre étude se focalisant surtout sur les machines parallèles, les machines vectorielles n'y figurent qu'à titre de comparaison de puissances observées.)

machine	type	mémoire	S/M-IMD	nb proc ¹⁵	nœud	réseau ¹⁶	Mflop/s
Paragon XP	//	dist	MIMD	3680	i860XP	tore 2-D	143400
Fujitsu	Vec			140			124500
TMC CM-5	//	dist	MIMD	1056	sparc	fat-tree	59700
NEC SX-3	Vec			4			23200
Cray T3D	//	part	MIMD	256	alpha	tore 3-D	21400
Cray Y-MP	Vec			16			13700
TMC CM-200	//	dist	SIMD	2048	Weitek	hyper-cube	9800
Hitachi	Vec			4			7016
KSR2	//	part	MIMD	128	64 bits	à anneaux	6923
TMC CM-2	//	dist	SIMD	65 536	1 bit	hyper-cube	5200
IBM SP1	//	dist	MIMD	128	RS6000	cross-bar	4800
Fujitsu VP2600	Vec			1			4009
KSR1	//	part	MIMD	128	64 bits	à anneaux	3380
Intel iPSC/860	//	dist	MIMD	128	i860	hyper-cube	2600
MasPar MP-2	//	dist	SIMD	16384	8 bits	tore 2-D	1600
MasPar MP-1	//	dist	SIMD	16384	4 bits	tore 2-D	473

FIG. I.7 - Quelques machines massivement parallèles

Ce classement montre qu'il existe deux grandes catégories de machines actuellement en exploitation pour le calcul scientifique, qui reflètent les deux types d'architectures présentées plus haut : les machines à base de processeurs vectoriels, dont le nombre de processeurs est

14. Le lecteur peut aussi se reporter à [BBS94] pour une évaluation des performances de ces machines sur plusieurs tests « grandeur réelle », ou à [CD94] pour d'autres machines.

15. Le nombre de processeurs indiqué est le nombre maximal pour une machine installée.

16. Certaines machines comportent plusieurs réseaux de données, seul le principal figure ici.

limité, et les machines massivement parallèles, dont la technologie de construction est moins performante (donc moins exigeante) mais dont le nombre de processeurs est nettement plus élevé. Notons aussi, en marge de ce classement, que les stations de travail actuellement produites ont vu leur puissance largement décuplée, ce qui aura pour effet certain de réduire le champ des domaines d'applications nécessitant l'utilisation de ces super-calculateurs.

Ces chiffres confirment que les machines MIMD sont plus performantes que les machines SIMD, probablement car les machines SIMD ne sont plus développées aujourd'hui. Les constructeurs préfèrent la réutilisation de processeurs puissants comme l'i860 ou l'alpha, plutôt que le développement d'un processeur dédié. On ne parle ici que de machines généralistes, en dehors de toute machine dédiée, par exemple au traitement ou à la synthèse d'images, qui peuvent faire l'objet de développements spécifiques de circuits matériels synchrones.

On remarque, enfin, que les machines à mémoire partagée atteignent « facilement » de bonnes performances avec un « faible » nombre de processeurs. Les constructeurs semblent mettre ainsi en valeur la facilité de programmation (et de développements d'outils) en proposant une solution équivalente à une machine séquentielle, en terme de vision pour le programmeur uniquement. Les problèmes de distributions efficaces des données persistent tout de même.

Les calculateurs vectoriels sont en exploitation depuis plus d'une vingtaine d'années, alors que les machines parallèles datent de la fin des années 80. Pourtant, les études de performances montrent que les puissances effectives de calcul se situent aujourd'hui dans les mêmes ordres de grandeur. Par contre, la progression des puissances est plus rapide pour les calculateurs parallèles.

Comme nous l'avons évoqué en introduction, la technologie est limitée dans sa progression. Les constructeurs de machines vectorielles, qui sont indiscutablement à la pointe des techniques de construction, ont maintenant tendance à accroître le nombre d'unités vectorielles et produire ainsi des machines parallèles dont les nœuds renferment le modèle d'exécution pipeline. La programmation de telles architectures s'oriente en conséquence vers le même type de parallélisme que les machines massivement parallèles, la problématique de la programmation vectorielle se retrouve aujourd'hui au niveau des nœuds de ces machines vectorielles.

En projetant ces constatations dans l'avenir, il est possible de penser que les machines massivement parallèles constituent l'avenir du calcul scientifique¹⁷. Nous nous intéresserons donc principalement à ces machines massivement parallèles.

17. Nous ne parlons ici que du critère scientifique. Encore faudra-t-il que les constructeurs de machines massivement parallèles outrepassent certains problèmes financiers...

4 Le Grand Challenge

PRÉVOIR les évolutions technologiques à venir est toujours un exercice qui comporte une grande part d'incertitude. Par contre, il est intéressant de se pencher sur les besoins connus en terme de puissance de calcul. Certains problèmes que l'on ne peut pas résoudre en un temps acceptable par l'intermédiaire de la technologie actuelle, ont été regroupés dans ce que l'on appelle le *Grand Challenge* [Wil91, HPCC94].

Aux États-Unis, le Grand Challenge est une organisation qui fédère de nombreux laboratoires dans le but de concentrer les travaux d'informatique scientifique sur certaines applications identifiées. Au delà du côté indiscutablement politique de cette fédération, il est instructif d'étudier les projections dans l'avenir issues des rencontres entre ces équipes.

Dans cette section, nous montrerons qu'une grande partie des besoins logiciels sont d'ores et déjà caractérisés comme devant être supportés par le parallélisme et que le fondement même des applications orientera leur résolution vers l'adoption impérative du modèle à parallélisme de données.

4.1 Environnement logiciel

En mai 1993, 34 équipes de recherches se sont réunies pour la durée du « Workshop and Conference on Grand Challenge Applications and Software Technology ». Principalement issues des laboratoires publics américains, ces équipes cherchent à résoudre une partie des applications regroupées au sein du Grand Challenge.

Pour les résoudre toutes, il a été mise en évidence la nécessité de proposer une nouvelle chaîne d'outils de développement devant comporter :

- des systèmes d'exploitation capables de gérer plusieurs dizaines de processeurs et leur mémoire, et pouvant supporter un environnement hétérogène ;
- de nouveaux langages de programmation permettant d'exprimer le parallélisme ;
- des mécanismes d'expression, dans un environnement parallèle, de manipulations de données codées suivant différents formats ;
- des outils de parallélisation et d'optimisation automatique de code ;
- des compilateurs capables de s'occuper de la distribution des données ;
- des outils de débogage ;
- des analyseurs de performances à grain fin ;

- des outils logiciels et matériels de communication entre les machines, par exemple par satellite ;
- des nouveaux outils de visualisation de données ;
- des logiciels de communication permettant une large diffusion d'informations ;
- des environnements de production capable de faciliter la mise en œuvre de plusieurs processus ;
- des outils pédagogiques rendant facilement accessibles les nouvelles applications.

Il apparaît clairement que tous ces outils ont été rendus indispensables par la volonté affichée, et par ailleurs incontournable, d'utiliser le parallélisme pour tenter de résoudre ces problèmes. Ainsi, l'étendue des fonctionnalités de ces outils tend à montrer que le besoin de puissance de calcul ne sera pas comblé entièrement tant qu'il n'existera pas un environnement logiciel capable de supporter, de rendre conviviales de telles architectures.

Notons que cette liste comporte deux principaux types d'outils : les outils de développements (langages, compilateurs, débogueurs...) dédiés à l'exploitation de la puissance de calcul ; et des outils d'environnement (systèmes d'exploitation, communications, diffusion d'informations, visualisation...) dédiés à la convivialité, mais d'un apport moins primordial que la première catégorie, en terme d'accès aux performances : ils ne sont pas directement liés à l'apport de puissance, mais en permettent la mise en œuvre effective.

Nous pouvons déjà conclure que le parallélisme est à la base des travaux du Grand Challenge. Bien évidemment, c'est la seule voie qui permettra d'atteindre des performances supérieures au Téra-flops, eldorado du calcul scientifique pour cette décennie.

4.2 Les applications

Le Grand Challenge regroupe plusieurs domaines, qui nécessitent tous une partie des impératifs logiciels cités plus haut. Après un parcours rapide des applications¹⁸, nous allons montrer qu'on devra mettre en œuvre le parallélisme de données lors de leurs résolutions.

Aéronautique Les problèmes de simulation pour la création de nouveaux véhicules doit mettre en œuvre d'importantes capacités de calculs. En particulier, ils demandent la résolutions de problèmes d'éléments finis, de mécanique des fluides et de visualisation graphique des données.

18. qui permettra de mettre en valeur l'utilité de notre domaine

Recherches énergétiques Les modèles de dynamique des fluides peuvent être appliqués à la modélisation des combustions. D'autre part, il existe des besoins dans les domaines des particules physiques à haute énergie, de la simulation d'écoulement de fluides à travers divers éléments et de la simulation de la fusion nucléaire.

Sciences environnementales La partie environnement du Grand Challenge regroupe les problèmes de prédiction du climat sur la globalité de la Terre et les études sur l'influence des polluants et événements naturels dans le but de prévoir les modifications futures de l'environnement, et d'adapter en conséquence certaines politiques de préventions.

Notons que les capacités de calculs offertes permettent aujourd'hui de commencer à fusionner les modèles existants qui traitaient séparément la dynamique des liquides et celle de l'air, apportant ainsi une vue globale de la simulation. Les algorithmes utilisés sont principalement issus de la résolution des équations de la dynamique des fluides (Navier-Stokes).

Biologie et médecine De la biologie moléculaire au décodage du génome, les applications médicales sont multiples. C'est certainement dans ce domaine que l'apport d'environnements informatiques puissants ouvrent le plus de portes aux scientifiques.

Beaucoup de ces applications comportent une partie importante de visualisation tridimensionnelle des données. D'autres consistent à modéliser tel ou tel organe, pour mieux en comprendre le fonctionnement.

Notons aussi l'intérêt de l'informatique temps-réel dans ce domaine, que ce soit pour la transmission d'image entre deux centres concernés, ou pour le traitement d'images médicales en temps-réel et leur interprétation immédiate pour la prise de décisions rapide, voire automatique.

Dans ce domaine, peut-être plus que dans les autres domaines du Grand Challenge, on imagine bien la nécessité de développer des applications totalement fiables ; d'où l'importance ici de proposer des environnements capables de résoudre avec une grande précision les problèmes posés.

Espace De même, il existe des simulations de dynamique des fluides dans la modélisation, par exemple, de l'activité solaire. Des grands besoins de puissance se font ressentir aussi dans l'exploration des galaxies et dans l'étude de leur formation.

Ici aussi, le temps-réel est utilisé : il permet l'écoute des ondes radios sur un spectre très large.

Synthèse des besoins Dans le cas général, les objectifs sont principalement :

- une meilleure résolution, par des données discrétisées plus finement, souvent en prenant une échelle spatiale plus fine, ou/et un pas de temps plus court ;
- un temps d'exécution plus court permettant aux scientifiques d'explorer plus facilement de nombreux paramètres de simulation ;
- une mise en œuvre des problèmes physiques plus réaliste, en opérant moins d'approximations ;
- le développement de modèle plus généralistes.

Le but des progrès à réaliser n'est donc pas uniquement la réduction du temps de calcul, mais aussi la mise à disposition du scientifique d'outils capables de laisser libre cours à son imagination et de lui permettre d'élargir l'étendue de ses simulations.

4.3 Le parallélisme de données

Dans toutes les applications du Grand Challenge, nous pouvons montrer que les algorithmes mis en place reposent tous sur le modèle du parallélisme de données. La preuve de cette affirmation nécessiterait au moins le développement complet de ces algorithmes. À défaut, nous nous proposons simplement d'entrevoir ce que peuvent être leur résolution.

Dans le domaine de la dynamique des fluides, la discrétisation des données physiques amènent le physicien à manipuler des matrices (de taille souvent énorme) dont chaque élément représente l'état d'une petite partie du système physique à un instant donné. Les équations sont ensuite implémentées sur ces données, sous la forme de traitements sur les matrices de départ. La solution du problème est alors obtenue en intégrant un système d'équations différentielles, et dans de nombreux cas, ce travail consiste principalement à résoudre des problèmes d'algèbre linéaire sur des matrices importantes [Per92b]. Par exemple, et en simplifiant grossièrement, les équations de Navier-Stokes peuvent aboutir à l'inversion de matrices bande (généralement tri-diagonales). Ces traitements matriciels sont, à l'évidence, directement amenés à être résolus en adoptant le parallélisme de données.

Pour la visualisation de données, comme pour la synthèse d'images, le parallélisme de données permet d'atteindre le traitement en temps-réel, par le calcul en parallèle des pixels qui constituent l'image à afficher.

La nature même des problèmes d'éléments finis (découpage d'un corps en particules) fait apparaître le parallélisme de données comme un modèle naturellement adéquat à l'algorithmique développée dans ce cadre.

Des problèmes spécifiques, comme le décodage du génome, font appels à des volumes de données énormes. Le traitement sur ces données est relativement simple, il est donc efficace de traiter en parallèle plusieurs parties des données. Ce qui revient à mettre en œuvre le parallélisme de données.

Enfin, les problèmes difficiles à résoudre, comme les problèmes d'une complexité reconnue (problèmes NP-complets), peuvent être résolus par des méthodes de recherche opérationnelle qui traitent des matrices, ou par des algorithmes génétiques qui satisfont eux-mêmes au paradigme parallélisme de données, de part le traitement en parallèle des gènes faisant partie de la population à étudier.

Comme nous le décrirons dans le chapitre suivant en parlant de preuves de programmes, le séquençement linéaire des instructions d'un programme à parallélisme de données rend celui-ci lisible, et facilement abordable pour celui qui n'est pas spécialiste du parallélisme. On peut affirmer que le modèle à parallélisme de données possède conjointement de bonnes propriétés pédagogiques et une capacité prouvée quant aux performances des codes générés. Ces deux critères réunis lui permettront une large diffusion.

4.4 Un modèle universel?

Les applications scientifiques du Grand Challenge sont résolubles par l'adoption du parallélisme de données. Par contre, d'autres applications scientifiques ne sont pas, *a priori*, data-parallèles. Nous nous proposons ici d'étudier trois exemples de familles d'applications qui sont proches ou éloignées de ce modèle de programmation.

4.4.1 Les algorithmes à pile

Les algorithmes comme les méthodes de Montè-Carlo utilisées en physique des particules, ou le lancer de rayon en synthèse d'image, sont qualifiés d'algorithmes à pile en raison du modèle algorithmique qu'il représentent: le programme applique un traitement sur un ou plusieurs éléments de base du problème (une particule ou un rayon dans ces exemples). Ce traitement consomme l'élément, mais peut produire un ou plusieurs autres éléments à traiter. L'exécution du programme est achevée quand tous les éléments, pour la plupart créés dynamiquement, ont été traités.

L'écriture directe d'un algorithme de cette famille peut faire appel à la récursivité pour le traitement des éléments créés. Pour cela, le programme décrit un traitement sur une particule puis appelle la même fonction de traitement sur les éventuels éléments créés durant l'évaluation du père. Cette méthodologie de programmation conduit directement à l'adoption

du parallélisme de tâches. Néanmoins, une autre approche permet la parallélisation : le parallélisme de données [Dek86]. Le programmeur manipule explicitement la pile des éléments en attente de traitement qui est distribuée sur les nœuds de la machine parallèle. La fonction de traitement est ensuite appliquée en parallèle sur les fractions de piles réparties à travers le réseau.

Cette approche permet d'obtenir une bonne efficacité quant au taux de parallélisme effectivement obtenu, à la condition d'inclure dans le mécanisme de répartition un système de rééquilibrage dynamique des fractions de pile [Fon94]. On peut donc affirmer que le parallélisme de données est un modèle permettant la mise en œuvre des algorithmes à pile.

4.4.2 Programmation logique

Le modèle de programmation introduite dans le domaine de la programmation logique est très éloigné du modèle de programmation impératif. Le programmeur déclare un ensemble de clauses et un but à satisfaire à partir de ces clauses. Le système résout le problème en faisant appel à un moteur d'inférences qui parcourt l'arbre des solutions.

Certains travaux visent à paralléliser l'évaluation de programmes logiques. Une approche utilise le paradigme du parallélisme de données pour permettre une exploitation de machine parallèle pour ce type de programmes [SH94]. Le moteur évalue le premier atome du corps de la clause à satisfaire et produit un ensemble de solutions possibles pour cet atome. Parmi toutes ces solutions, l'extraction des solutions finales est ensuite réalisée par l'évaluation en parallèle de la conjonction des atomes restant dans le corps de la clause, sur ces solutions potentielles (cf. figure I.8). Cette étape est bien construite suivant le paradigme du parallélisme de données¹⁹.

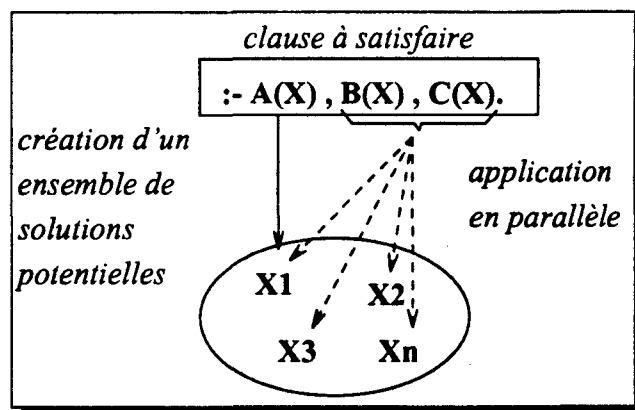


FIG. I.8 - Le parallélisme de données pour la programmation logique

19. On parle de parallélisme « ou » dans la terminologie de programmation logique.

Bien que ce type de problèmes ne suive pas le modèle de programmation data-parallèle, il constitue un exemple de l'utilisation de ce parallélisme à un niveau inférieur de la chaîne de développement d'un programme. Notons tout de même que ce nouveau mécanisme d'exécution d'un programme Prolog change l'ordre de parcours de l'arbre, et par conséquent la sémantique associée à un programme.

4.4.3 Simulations non numériques

Les simulations non-numériques traitent souvent de problèmes issus de la vie courante. Chaque entité qui fait partie d'un système complexe à étudier est modélisée sous forme d'un acteur par un programme qui reproduit son comportement réel. Les interactions entre acteurs sont reproduites sous forme d'envois de messages. L'étude du système se fait ensuite en « laissant » interagir les différents programmes entre-eux. Souvent, ce type de simulations est développé à l'aide de langages à objets [YT87, Laz90, CaP94].

L'asynchronisme des événements est à l'évidence sous-jacent à cette programmation. Les entités présentes ont un comportement indépendant les unes des autres. Un code doit être développé pour chaque type d'objet présent dans le système. Il n'y a donc pas de possibilité de paralléliser ce type d'applications en suivant le modèle à parallélisme de données²⁰.

Ce type de problèmes, et de développement de programmes, met en évidence les bonnes qualités du parallélisme de données vis-à-vis des problèmes de débogage ou d'analyse de comportement. Avec ce type de modèle d'exécution totalement incontrôlée par le programmeur, l'interprétation du comportement d'une application est souvent très difficile, et fait l'objet de très nombreuses recherches (trace d'un programme, ré-exécution, analyse fine de performances...).

5 Conclusion

CINQUANTE années nous séparent de la construction du premier ordinateur. Nous venons de montrer que depuis l'apparition du calcul automatique, on pouvait considérer le parallélisme de données comme déjà présent avec le regroupement de données homogènes dans des entités manipulées par le programmeur au cours du développement d'un algorithme.

Depuis l'apparition d'architectures de modèles d'exécution non-*von Neumann*, le parallélisme de données devient plus explicite. Les machines deviennent potentiellement capables

20. Sauf dans le cas particulier où un grand nombre d'objets d'un même type serait présent dans le système, auquel cas, le parallélisme de données pourrait être utilisé de façon locale à un acteur qui regrouperait une famille d'objets homogènes.

d'exploiter le parallélisme sur les données, transférant ainsi les concepts de modèle de programmation vers le modèle d'exécution.

Avec l'arrivée de machines massivement parallèles, synchrones et asynchrones, le parallélisme de données est souvent une solution de mise en œuvre simple et efficace du parallélisme matériel. Il permet alors d'unifier, pour les différents modèles d'exécution, un seul et même modèle de programmation.

Chapitre II

Modèles de programmation à parallélisme de données

DANS ce chapitre, nous proposons une caractérisation du modèle à parallélisme de données par l'étude de la problématique de base relative à la définition des langages adoptant ce modèle. Nous nous pencherons aussi sur leur compilation, étape primordiale dans le processus permettant d'atteindre des performances correctes vis-à-vis de la capacité des architectures cibles.

Nous avons choisi de rediriger le lecteur vers d'autres ouvrages complets pour satisfaire sa soif de connaissances précises sur de multiples langages. À titre d'illustration, nous présenterons uniquement les principales fonctionnalités d'HPF, langage apparu en 1993 sous l'égide d'un collectif regroupant les principaux constructeurs de machines.

Ensuite, nous allons extraire à partir des divers langages dédiés au parallélisme de données, les différents modèles de programmation sous-jacents. Au delà de la syntaxe propre d'un langage, il est en effet intéressant de se pencher sur les concepts fondamentaux mis en jeu lors de la programmation par tel ou tel langage. Nous avons identifié deux grandes catégories de modèles.

Le premier ensemble de modèles de programmation traite du modèle à flot de données. On l'appelle plus couramment le modèle *data-flow*. On peut distinguer deux approches sensiblement différentes : la programmation par les langages systoliques, et la programmation par les langages issus du formalisme mathématique à base d'équations récurrentes, qui peuvent aboutir eux-mêmes au modèle d'exécution systolique.

L'autre ensemble de modèles, plus étendu en nombre de langages que le précédent et plus utilisé dans les domaines du calcul scientifique, est directement issu de l'idée de base du data-parallelisme, telle que nous l'avons présentée au chapitre précédent. Il regroupe les langages

qui mettent en œuvre les modèles que nous appellerons les *modèles de programmation par structures globales*. Parmi eux, nous distinguerons le modèle de programmation irrégulière proposé par certains de ces langages.

Enfin, nous présenterons le modèle géométrique. À partir de constatations simples sur l'usage courant du parallélisme de données, nous caractériserons ce que doit proposer un modèle proche de l'algorithmique. Nous appliquerons ces choix dans la définition du modèle HELP, que nous proposerons dans la deuxième partie de ce chapitre.

Avertissement Nous avons choisi de sélectionner quelques langages pour illustrer chacun des modèles. Notre choix s'est souvent porté vers les langages et environnement développés par des équipes françaises.

1 Le parallélisme de données

LA PRÉSENTATION des différents modèles d'exécution a montré que le parallélisme de données semble être une notion qui a vu le jour en même temps que les premiers langages de programmation. Nous nous attacherons dans cette section à caractériser plus précisément ce modèle de programmation.

Dans un premier temps, après une mise au point sur le terme « modèle de programmation », nous montrerons que le fait de garder un déroulement séquentiel du code, en appliquant le parallélisme au niveau de l'instruction primaire, permet d'obtenir de bonnes propriétés, non seulement pour le développement et la lisibilité des programmes, mais aussi pour la possibilité offerte au programmeur de formaliser clairement une preuve de ses programmes. Pour cela, nous référencerons les travaux théoriques de l'équipe de recherche du LIP qui a montré les bons fondements sémantiques du parallélisme de données.

Nous proposerons ensuite un aperçu des problèmes de base du parallélisme de données, quant à l'expression dans le langage des opérations de base et à leur génération de code. Nous verrons ainsi qu'un langage data-parallèle doit proposer des constructions syntaxiques permettant d'exprimer les informations issues de l'algorithme et de diriger ensuite le compilateur afin de générer un code « le plus efficace possible ».

Enfin, nous étudierons un langage mettant en œuvre ce paradigme. À défaut d'un panorama complet des langages data-parallèles existants, nous avons choisi de sélectionner HPF, dernier Fortran en date¹, pour son universalité promise. Le lecteur peut se reporter à de nombreux ouvrages qui présentent de non moins nombreux autres langages : [HJ88, Ker92, Per92a, Mar93a].

1. en attendant HPF-2!

1.1 Programmation et exécution

Certains utilisateurs de machines parallèles opposent parfois le paradigme « parallélisme de données » à celui de « passage de messages ». Il est important de mettre en valeur les deux niveaux différents auxquels se situent ces deux notions, le modèle d'exécution et le modèle de programmation.

Le **modèle de programmation** est le cadre dans lequel peut s'exprimer un algorithme répondant au problème posé. Ainsi, le parallélisme de données, par l'intermédiaire des langages qui y sont associés, est une boîte à outils permettant la description d'un algorithme. Nous verrons par la suite ce que renferme cette boîte.

Par contre, le **modèle d'exécution** est la façon dont un programme sera implémenté et dont les concepts nécessaires au modèle mis en place par l'algorithme vont être réalisés au niveau du système d'exploitation et de l'architecture de la machine. Ainsi, dire qu'un programme va utiliser le passage de messages est une conséquence de l'utilisation d'une machine asynchrone ; ce n'est pas forcément un modèle de programmation. Dire qu'un programme est développé au-dessus d'une couche de communication comme PVM, ne renseigne pas sur le modèle de programmation utilisé à un niveau plus haut dans la chaîne de développement.

La parallélisme de données est un modèle de programmation qui consiste à effectuer un traitement unique sur un certain nombre de données homogènes. Ce paradigme est indépendant du modèle d'exécution ; sur machine SIMD, les traitements seront effectués suivant le modèle d'exécution synchrone ; tandis que sur machine asynchrone, les échanges de valeurs entre processeurs et les synchronisations nécessaires au parallélisme pourront être mis en œuvre sous forme de passage de messages.

De plus, la notion d'exécution « parallèle » peut être distinguée du modèle à « parallélisme » de données. On dit « parallélisme » de données car les traitements d'un élément de l'ensemble des données est équivalent au traitement d'un autre élément, peu importe qu'il y ait simultanément entre ces deux opérations. On retrouve d'ailleurs la non-simultanéité dans les langages data-parallèles de bas niveau, non-virtuels (ou dans les projections des langages virtuels) : un processeur peut posséder plusieurs éléments du même ensemble à traiter, et par conséquent, ces éléments seront traités séquentiellement alors qu'il s'agit pourtant d'une application du modèle à parallélisme de données. L'ambiguïté du terme « parallélisme » ne doit pas cacher ce principe. Par extrapolation, on peut aussi dire qu'un programme écrit avec un langage classique² est « data-parallèle » dès qu'il met en jeu un ensemble de données homogènes, et qu'il opère un traitement indépendant sur chaque élément de cet ensemble.

Pour prendre en compte cette distinction, on pourrait parler de « parallélisme potentiel de

2. classique dans le sens « non-parallèle »

données » pour le modèle de programmation et de « parallélisme de traitement de données » pour l'exécution simultanée d'un même code sur plusieurs données.

1.2 Sémantique et preuves de programmes

Un langage data-parallèle met à disposition du programmeur diverses constructions qui ont pour but d'exprimer, le plus simplement possible, les caractéristiques de base du parallélisme de données. Le langage \mathcal{L} défini par Luc Bougé est un noyau minimal regroupant cinq constructions qui peuvent modéliser à elles seules les autres possibilités d'expression des langages data-parallèles existants [Bou93, BLVU94].

Les cinq constructions de \mathcal{L} sont :

La séquence décrit la succession des traitements ;

L'affectation parallèle donne aux éléments d'une variable parallèle la valeur résultante de l'évaluation d'une expression locale.

Le conditionnement permet de gérer l'activité des processeurs en fonction des différentes valeurs du flot de données. Certains processeurs peuvent ainsi être inhibés le temps de l'exécution d'une partie du programme.

Les communications sont effectuées par des processeurs actifs qui lisent une donnée dans la mémoire d'autres processeurs (pas forcément actifs pour leur part).

L'itération établit un déroulement en boucle du programme. À chaque itération, un test est effectué sur chaque processeur encore actif. Quand un de ces processeurs ne satisfait pas au test, il est inactivé jusqu'à la sortie de boucle. Quand tous les processeurs deviennent inactifs, la boucle est achevée et l'activité des processeurs est restaurée avec sa valeur en entrée de boucle.

Les travaux de l'équipe de Luc Bougé³ ont abouti à la preuve de l'expressivité du langage \mathcal{L} par la mise en forme d'une sémantique opérationnelle et d'une sémantique dénotationnelle. Ils montrent ensuite l'équivalence de plusieurs langages⁴ par équivalence de leurs sémantiques opérationnelles [Lev93].

D'autres résultats sont issus de l'extension de la notion d'assertion selon la méthode axiomatique de Hoare au cas du data-parallélisme. Une assertion est composée d'un ensemble de

3. Laboratoire d'Informatique du Parallélisme, Lyon.

4. en particulier des langages \mathcal{L} , MPL [Mas91b] et POMPC [Par92].

données (comme dans le cas du séquentiel) et d'un ensemble de valeurs booléennes représentant l'activité des processeurs. La sémantique dénotationnelle est définie pour chacun des cinq constructeurs. Un système de preuve est mis en place pour chacune des constructions de base du langage, et la particularité de la sémantique du conditionnement data-parallèle est mise en évidence par un résultat relatif aux pré-conditions les plus faibles [BLVU94].

Il est beaucoup plus complexe d'arriver à de tels résultats en ne considérant pas le paradigme du parallélisme de données. Par exemple, modéliser l'exécution parallèle d'un programme multi-tâches par la donnée d'une sémantique clairement établie doit prendre en compte le non-déterminisme d'exécution pour l'ordre des réceptions de messages, entre autre. Pour ce domaine, en particulier pour la modélisation théorique de l'asynchronisme, nous ne ferons qu'évoquer les travaux initiés par Petri [Pet62, Bou88]. Disons simplement que les langages « non data-parallèles » couramment utilisés n'offrent pas la possibilité de formaliser un déroulement séquentiel du programme. La preuve des programmes est alors souvent beaucoup plus difficile à exprimer.

En conclusion, on comprend l'intérêt, au regard des impératifs sémantiques, de garder une exécution déterministe comme dans le cas du parallélisme de données. C'est le seul paradigme qui permet conjointement de prouver qu'un programme est correct et d'obtenir de bonnes performances sur des applications réelles.

1.3 Les problèmes de base

Nous proposons dans cette section de mettre en évidence les problèmes cruciaux de la définition d'un langage data-parallèle ; et d'en tirer une classification suivant 5 points importants :

- langage virtuel ou non-virtuel ;
- langage explicite ou implicite ;
- type de machine virtuelle défini dans le langage ;
- accès aux données par les indices ou par un référentiel ;
- directives de distribution des données.

1.3.1 Virtualisation

Les ensembles de données à traiter en parallèle ont une taille dépendante du problème et de l'algorithme qui le résout. La machine cible ne possède pas forcément le nombre exact de

processeurs permettant d'associer à chacun des éléments une unité de traitement. Généralement, le nombre de processeurs est malheureusement inférieur au nombre de données⁵. Deux solutions existent pour palier ce déficit.

Langages non-virtuels La première des deux solutions consiste à imposer au programmeur de manipuler des données dont la taille correspond à la machine cible. Ce sont les langages non-virtuels. Efficaces, mais difficilement utilisables pour de grosses applications, ils sont réservés aux spécialistes de la programmation parallèle. Les données doivent être découpées en plusieurs tranches correspondantes à la taille de la machine cible. Un programme écrit pour un certain nombre de processeurs ne sera donc pas portable vers la même machine dont la taille est différente.

Par conséquent, ces langages n'apportent aucune portabilité. Considérons les comme des macros-assembleurs, et non comme des langages de calcul scientifique.

Langages virtuels Les données manipulées sont de taille quelconque. Le programmeur peut considérer que, quelque soit la taille de ses données, il existe un processeur par élément. Ce sont les langages virtuels. Beaucoup plus faciles à utiliser que les précédents, ils requièrent l'utilisation d'un compilateur beaucoup plus perfectionné, qui prend en charge le travail de virtualisation [Par93a]. Les performances de tels langages sont généralement plus basses, mais rendent l'univers du parallélisme accessible à tout programmeur.

Définitions

On appellera *virtualisation* le découpage des données de l'algorithme en plusieurs ensembles (ou *couches*) de tailles correspondantes à la taille de la machine. Comme nous l'avons montré, cette étape est laissée à la charge du compilateur pour les langages virtuels, du programmeur pour les langages non-virtuels.

On appellera *projection* l'association des données issues de la virtualisation à la machine cible. Voir figure II.1.

5. à l'image des crédits alloués par rapport aux crédits réclamés.

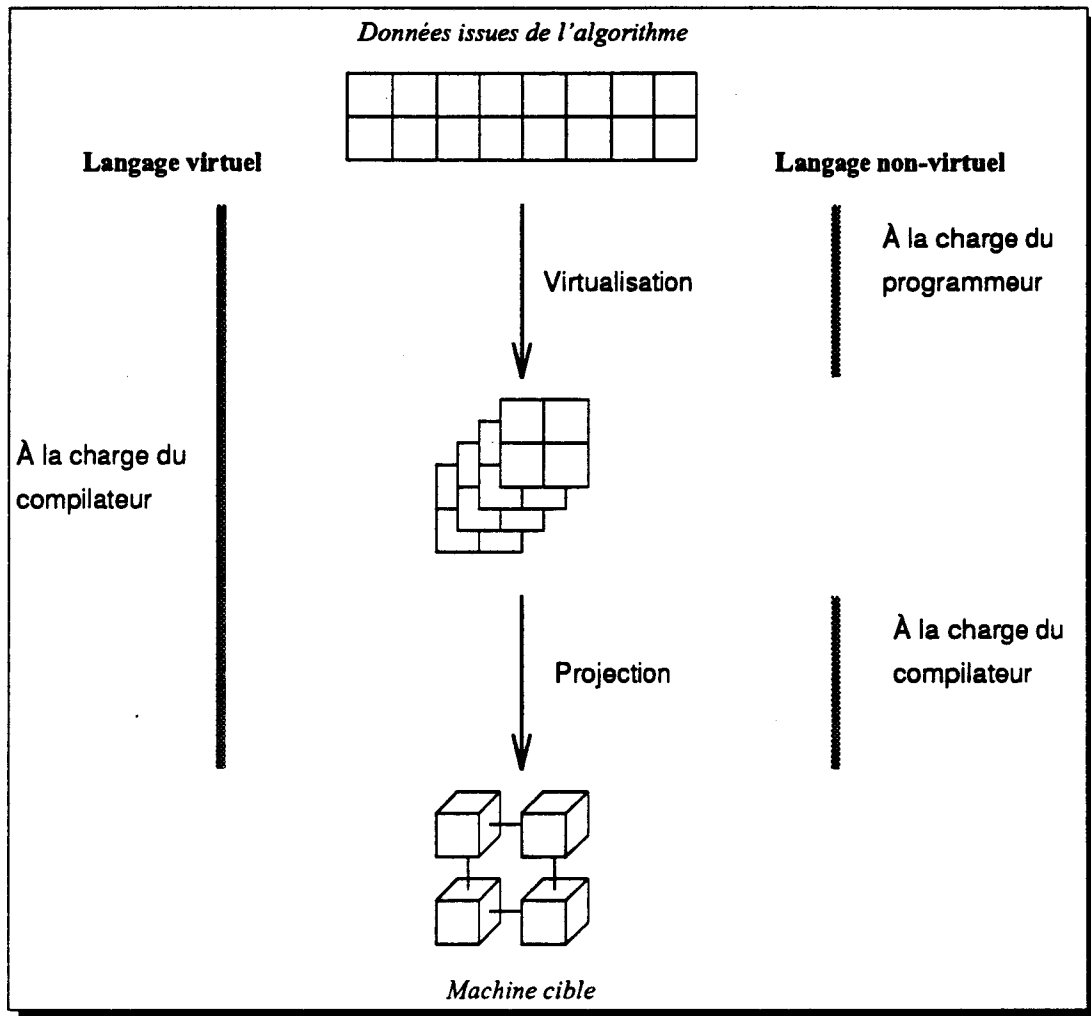


FIG. II.1 - *Les langages virtuels ou non-virtuels*

1.3.2 Déclarations d'objets

Opérer un traitement sur un ensemble de données homogènes nécessite avant tout de pouvoir déclarer dans le langage ces ensembles homogènes. Déjà sur ce point, on peut discerner deux familles de langages : ceux qui imposent de caractériser explicitement les ensembles qui vont faire l'objet de traitements parallèles, et ceux qui ne considèrent pas que le fait d'appliquer un traitement parallèle à un objet impose de caractériser explicitement cet objet.

Langages implicites Dans le cas d'objets non spécifiquement data-parallèles, c'est le compilateur qui est en charge d'allouer ces données de telle façon que le parallélisme potentiel soit exploité. On se trouve alors plutôt dans la philosophie « parallélisation automatique ». Cer-

tains langages, comme beaucoup d'extensions parallèles de Fortran, fournissent pour ce faire des directives de compilation pour orienter la génération de code vers le parallélisme. Nous pouvons regretter que ces directives ne puissent offrir aucune sémantique dans le langage. En effet, puisque les directives ne sont pas impérativement prises en compte par le compilateur, le langage sans directive doit posséder la même sémantique qu'avec la présence de ces directives. Par conséquent, un tel langage ne peut pas faire l'objet de l'élaboration d'une modélisation théorique dans sa version complète.

De plus, il n'y a pas de modèle impératif de programmation proposant une façon de procéder pour obtenir effectivement le parallélisme. Le caractère optionnel des directives cachant souvent les capacités (ou incapacités) du compilateur, les informations exprimées par le programmeur doivent être retrouvées et exploitées durant la phase de compilation.

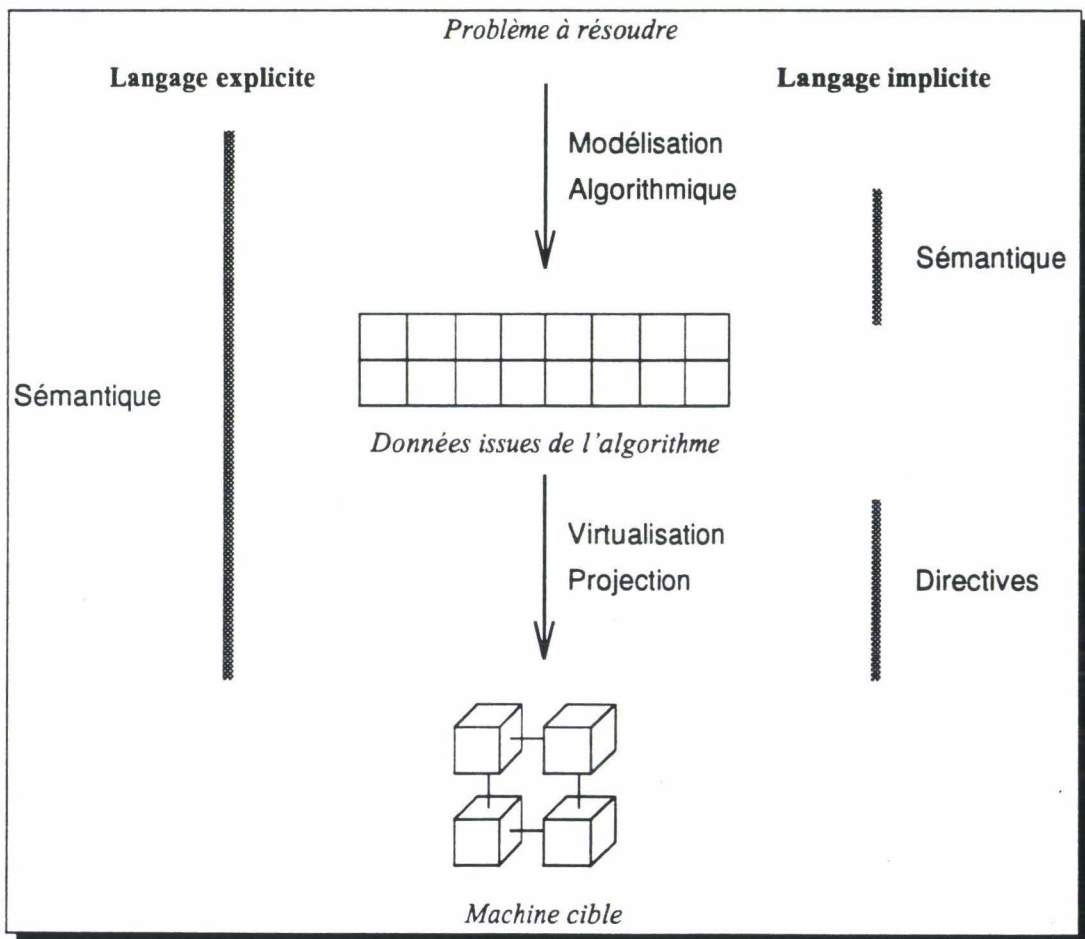


FIG. II.2 - Les langages explicites ou implicites

Langages explicites Le second type de langage comportent les langages dits *explicites* qui offrent tous la possibilité de déclarer qu'un objet se verra appliquer des traitements data-parallèles. Généralement, surtout dans le cas des extensions de C, cette déclaration est faite par l'intermédiaire de la création d'une *machine virtuelle*. Les objets issus de l'algorithme sont alors projetés sur cette machine virtuelle et la phase de compilation (virtualisation + projection) sera opérée sur la machine virtuelle. Voir figure II.2.

Un langage explicite permet donc au programmeur d'exprimer complètement les traitements et la projection physique. Son contrôle est plus important que pour un langage implicite dans lequel seul le compilateur maîtrise la phase de projection, sans intervention de la sémantique du langage.

1.3.3 Machines virtuelles

Nous avons distingué deux types de machines virtuelles suivant la raison de déclarer dans un même programme plusieurs de ces machines. Le premier type de langages dispense au programmeur la possibilité de définir des machines virtuelles qui regroupent les objets de même caractéristiques. Les autres langages permettent le regroupement d'objets indépendamment de leurs caractéristiques, en précisant l'alignement de ces objets par rapport à la machine virtuelle.

Machines virtuelles d'instanciation Un langage peut définir une machine virtuelle pour regrouper les objets de mêmes caractéristiques. Par exemple, C* [Thi91] définit des *shape*, POMPC [Par92] définit des *collection*, PARALLAXIS [Brä89] définit des *architecture*. Ces objets ont tous la même taille, et peuvent interagir à condition qu'ils soient déclaré « au sein » de la même machine virtuelle. Les interactions entre deux objets de deux machines virtuelles différentes (donc de deux tailles différentes) se font par des appels explicites à des primitives de transfert ou à des conversions de type.

Machines virtuelles d'alignement L'autre raison de déclarer plusieurs machines virtuelles est d'ordre algorithmique. Le programmeur dessine un programme et le découpe en plusieurs parties relativement indépendantes. Il décide alors de ne pas projeter tous ses objets dans la même machine virtuelle, mais plutôt d'associer une machine virtuelle à une partie de son algorithme. Les objets qui interagissent sont alors alloués dans la même machine virtuelle qui doit, dans ce cas, être capable d'accueillir tout objet, quelles que soient ses caractéristiques. C'est le cas des *template* d'HPF ou des *hspace* de C-HELP (cf. chapitre III). Voir figure II.3.

On appelle cette catégorie *machines virtuelles d'alignement* car le langage doit fournir la possibilité d'exprimer l'alignement des objets par rapport au référentiel de la machine virtuelle.

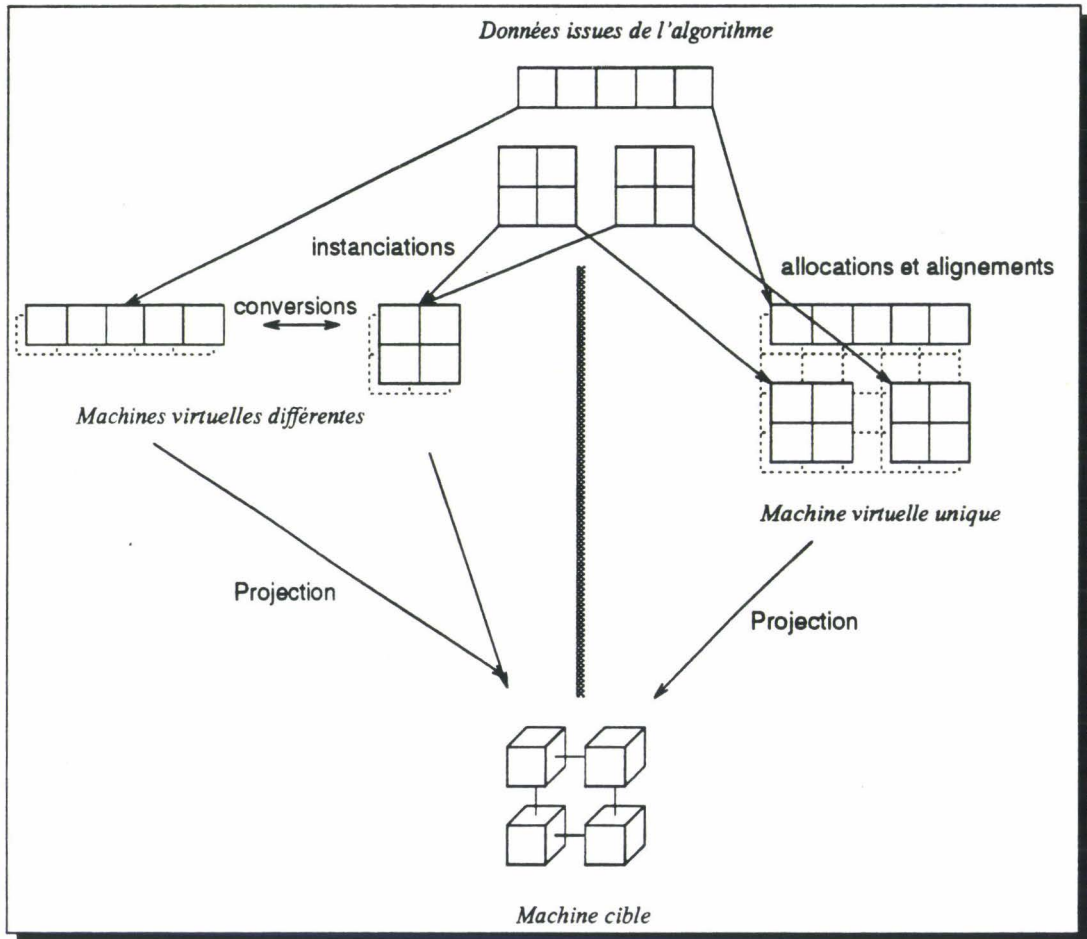


FIG. II.3 - Les deux types de machines virtuelles

Définitions

On appellera *instanciation* la déclaration d'un objet dans une machine virtuelle quand celle-ci entraîne la fixation de la taille de l'objet à celle de la machine virtuelle (premier type de machines virtuelles).

On appellera *allocation* la déclaration d'un objet dans une machine virtuelle quand cela n'entraîne pas de détermination de taille (seconde solution). Le langage doit permettre dans ce cas l'explicitation des caractéristiques de taille de l'objet au moment de sa déclaration.

On appellera *alignement* le positionnement d'un objet dans la machine virtuelle. Dans la

première solution, l'alignement est fixé pour tous les objets instanciés dans la même machine virtuelle. Cette remarque peut nous amener à penser que les langages qui offrent la seconde solution sont plus souples car permettent ces alignements et l'interaction directe entre deux objets de caractéristiques géométriques différentes.

1.3.4 Expressions data-parallèles

On s'intéresse maintenant à l'écriture dans les langages data-parallèles d'une expression faisant intervenir plusieurs objets data-parallèles.

La syntaxe couramment reconnue, et adoptée par tous les langages data-parallèles évolués, permet de raccourcir les notations d'accès aux structures de données parallèles, en référant uniquement le nom d'une variable pour considérer implicitement l'accès à chaque élément. On appellera *expression data-parallèle* une expression faisant intervenir plusieurs objets eux-mêmes data-parallèles.

Ainsi, si **A** et **B** sont deux objets data-parallèles à une dimension, l'expression **A+B** est équivalente à une boucle qui balayerait tous les éléments de **A** et **B** en déclenchant les évaluations successives des additions des éléments correspondants de **A** et de **B**⁶.

Cette notation n'est pas ambiguë dans le cas de deux objets d'une machine virtuelle d'instanciation car ces deux objets ont impérativement la même taille et sont alignés de la même façon (deux éléments de même position par rapport à l'origine de l'objet, sont projetés sur le même processeur virtuel). Par contre, dans le cas d'une interaction possible entre deux objets de forme ou de positionnement différents dans la même machine virtuelle, il existe deux interprétations possibles, donnant à l'expression deux sémantiques différentes.

Les langages basés sur le référentiel La première solution consiste à utiliser la machine virtuelle comme support de l'expression de l'activité. En fixant certaines conventions, un langage qui adopte cette solution détermine pour chaque expression un ensemble de processeurs de la machine virtuelle qui effectueront l'opération. Cet ensemble est appelé *domaine d'activité*. L'interprétation d'une expression se fait maintenant localement aux processeurs virtuels actifs, indépendamment de l'indice de chaque élément d'un objet. Voir figure II.4.

Une importante caractéristique de chacun de ces langages est la façon dont est exprimée (ou calculée à partir du programme source) la géométrie du domaine d'activité.

On peut d'ores et déjà remarquer qu'une expression évaluée avec cette interprétation ne peut pas engendrer de communication entre les processeurs de la machine virtuelle car

6. on ne considère aucun ordre de parcours pour cette boucle.

les éléments ne sont pas indicés par leur position par rapport à l'origine de l'objet mais sont considérés localement aux processeurs virtuels. Les communications devront donc être explicitées par des constructions spécifiques à chacun de ces langages.

Cette interprétation est adoptée par tous les langages proposant une machine virtuelle d'instanciation. De plus, le langage \mathcal{L} dont nous avons vu les bonnes propriétés du point de vue de la sémantique se base sur cette interprétation, par la prise en compte d'expressions qui interdisent les communications implicites aux expressions. Seuls les langages basés sur un référentiel sont donc mis en valeur par ces travaux théoriques. L'extension de ces résultats fait l'objet de travaux dont le but est d'intégrer les communications implicites au formalisme théorique.

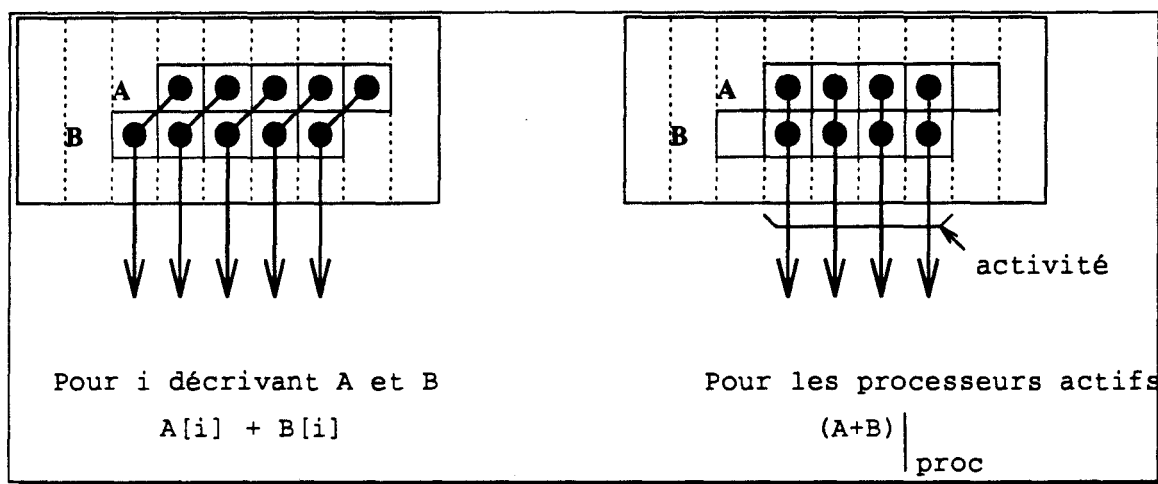


FIG. II.4 - Les deux interprétations sémantiques

Langages basés sur les indices L'autre interprétation est directement issue de la notion de tableaux. À toute expression data-parallèle, on donne la sémantique d'une boucle englobante (ou de plusieurs boucles si les objets sont multi-dimensionnels), qui décrit les objets intervenant dans l'expression, et qui opère le traitement élément par élément à chaque itération. On peut entrevoir les problèmes liés à l'interprétation d'expressions dont tous les membres ne possèdent pas le même nombre d'éléments.

Le plus important reproche que l'on peut faire à cette interprétation sémantique est de cacher des communications. On parlera dans ce cas de *communications implicites*. Comme le montre la figure II.4, pour une expression arithmétique simple, deux éléments appartenant à des objets différents peuvent être référencés par le même indice implicite, alors qu'il ne se trouvent pas sur le même processeur virtuel. Le compilateur doit donc assurer la migration

d'une donnée vers l'autre⁷. De plus, le langage ne permet pas d'expliciter laquelle de ces deux valeurs va faire l'objet de la communication. Cette décision est à la charge du compilateur, nous y reviendrons dans le chapitre consacré à la compilation. Cette incertitude quant au mode d'exécution rend difficile la résolution du problème des placements efficaces de données sur la machine [KLS90].

Cette interprétation est reprise inévitablement par tous les langages implicites, et en particulier par tous les Fortran.

1.3.5 De la machine virtuelle à la machine physique

La phase de compilation de plus bas niveau consiste à projeter la machine virtuelle sur la machine physique. Ici intervient fortement la topologie de la machine cible et le type de machine virtuelle à projeter. Dans le cas le plus courant, la machine virtuelle est un tableau multi-dimensionnel et la machine physique est une grille dont le nombre de dimensions est compris entre 1 et 3. Le langage utilisé permet généralement de diriger cette projection, par l'expression de directives souvent à la charge du programmeur. Un ordre de priorité est extrait du code source pour déterminer quelles seront les dimensions prioritaires en terme de parallélisme.

Une fois sélectionnées, ces dimensions doivent faire l'objet d'un « découpage » pour que chaque processeur virtuel soit associé à un processeur physique. Nous avons vu que le nombre de processeurs virtuels étant généralement supérieur au nombre de processeurs physiques, un regroupement de plusieurs processeurs virtuels sur un même processeur physique est nécessaire.

Suivant la topologie de la machine et la taille des dimensions virtuelles, on associera une (ou éventuellement plusieurs) des dimensions prioritaires à une dimension de la machine. Le regroupement linéaire des processeurs virtuels peut alors se faire sur la base de deux stratégies :

Découpage par blocs On peut regrouper les processeurs virtuels adjacents sur une dimension de la machine virtuelle par blocs de taille suffisamment grande pour que l'association du i^e bloc au i^e processeur physique permette de projeter en totalité la dimension virtuelle. La taille des blocs est égale à la longueur de la dimension virtuelle divisée par la longueur de la dimension physique, plus 1. Voir figure II.5.

Ce système de projection a l'intérêt de projeter, le plus souvent, deux processeurs virtuels logiquement voisins sur le même processeur physique, minimisant ainsi les temps de commu-

7. Voire même des deux opérandes vers un troisième processeur

nications entre deux processeurs virtuels voisins, ce qui est souvent rencontré par exemple pour le décalage des éléments d'un vecteur ou pendant des calculs de convolution.

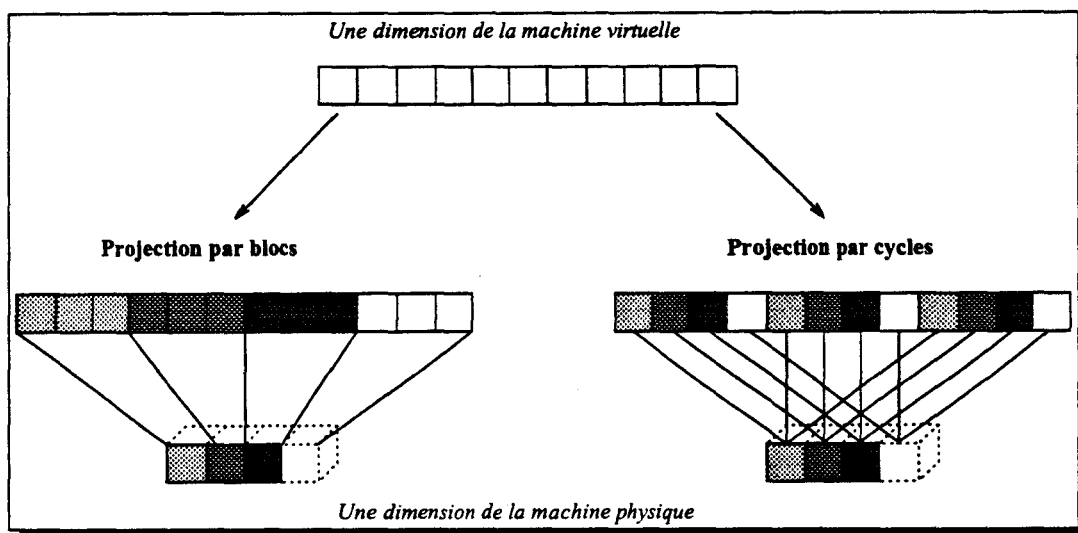


FIG. II.5 - *Projection par blocs ou par cycles*

Découpage cyclique La solution duale consiste à projeter cycliquement les objets sur la machine physique. Le i^{e} processeur virtuel est projeté sur le processeur physique de numéro i modulo le nombre de processeurs physiques (à 1 près, selon la convention de commencer la numérotation à 0 ou 1). Voir figure II.5.

Contrairement à la projection par blocs, deux processeurs virtuels voisins se trouvent sur deux processeurs physiques différents. On favorise ainsi le parallélisme effectif des opérations calculatoires déclenchées sur des processeurs virtuels adjacents. Les temps de communications entre processeurs virtuels adjacents sont, en contre-partie, plus élevés.

Découpages mixtes Des solutions intermédiaires peuvent être adoptées. Elles consistent à projeter cycliquement des blocs de processeurs virtuels. Ces blocs sont de taille plus petite que les blocs de la projection par blocs, et par conséquent, sont en nombre plus élevé. Généralement, le i^{e} bloc est projeté sur le processeur physique de numéro i modulo le nombre de processeurs physiques. Quand un langage permet ce découpage, il permet implicitement les deux autres, par la spécification par le programmeur de la taille des blocs (1 pour cycle, taille de la dimension divisée par nombre de processeurs pour le découpage en blocs). Notons l'originalité du langage HPF qui propose une distribution des blocs par cycles ou par blocs (cf 2.2.2.2)

2 Une illustration: HPF

À DÉFAUT de présenter finement plusieurs langages data-parallèles, nous avons choisi de focaliser les illustrations sur le langage HPF.

2.1 Vers un standard? Le forum HPFF

HPF est issu du « High Performance Fortran Forum » qui a réuni durant de nombreux débats physiques et cybernétiques, les principaux constructeurs de machines et beaucoup de personnes travaillant autour du développement de compilateurs ou du développement d'applications diverses. Le but de ce forum était de définir, pour la première fois dans l'histoire, une version **universelle** de Fortran. Les dernières années avaient bien vu quelques tentatives de normalisation du Fortran (en particulier Fortran 8x, devenu Fortran 90 [Met87]), mais l'importance du projet HPF par son nombre de participants et par la réelle volonté politique d'arriver à un standard, en ont fait *de facto* le langage à utiliser, ou au moins à connaître, pour les années qui viennent. C'est ainsi que les constructeurs de machines proposent, ou proposeront dans une avenir proche, leur compilateur HPF.

Notons que les spécifications du langage ont, pour une fois, précédé les premiers compilateurs, ce qui devait rendre plus libres (?) les intervenants pour l'élaboration du consensus.

Les nouvelles capacités de ce Fortran devaient:

- être compatible avec Fortran 77. On ne se sépare pas si facilement du bon vieux Fortran ;
- être suffisamment généraliste pour ne pas refléter uniquement un type d'architecture ou une machine précise. Il faut bien que chaque constructeur se sente intéressé par le projet ;
- pouvoir proposer à l'utilisateur des fonctionnalités facilement abordables pour mettre en œuvre le parallélisme, d'une façon à obtenir un minimum de performances ;
- être « compilable ». C'est assez utile pour un langage.

La première version des spécifications d'HPF a fait l'objet d'une publication des caractéristiques principales du langage en janvier 1993 [Hpf93]. La version définitive des spécifications du langage a été adoptée en mai [For93]. Notons tout de suite que malgré la volonté consensuelle au départ d'HPFF, le document présentant les spécifications du langage comporte une partie dédiée au noyau minimal que doit supporter un compilateur HPF. Ainsi, deux compilateurs HPF peuvent ne pas forcément compiler le même langage. On évoque couramment les *subsets* d'HPF, en attendant les compilateurs *full-HPF*.

Cet ensemble minimal facilite sûrement le travail de développement rapide d'une première version minimale, pas forcément très étendue mais constituant un noyau efficace, puis l'amélioration incrémentale vers un compilateur plus complet et un code généré plus efficace. Les utilisateurs n'ont plus qu'à espérer que les compilateurs développés vont converger vers le point commun avant que celui-ci fasse l'objet de nouvelles spécifications.

2.2 Le langage HPF

HPF a été identifié dès les premières conclusions du forum comme devant être un langage data-parallèle. Comme nous l'avons exposé au début de ce chapitre, le modèle de programmation est considéré comme primordial, quel que soit le modèle d'exécution découlant de l'architecture cible. Ainsi, les primitives explicitement teintées de « message passing » sont bannies rapidement des discussions.

HPF a été prévu pour engendrer un minimum de problèmes de compatibilité avec d'autres langages, en particulier avec les anciens Fortran. De plus, les passages de paramètres dans le développement des bibliothèques ont été pensés de façon à favoriser l'intégration de routines développées dans d'autres langages, en C par exemple.

Pour caractériser HPF en fonction des distinctions que nous avons exposées dans le paragraphe précédent, nous pouvons classer HPF dans les langages virtuels, implicites, proposant une machine virtuelle pouvant regrouper différentes formes d'objets, basés sur les indices et proposant une distribution mixte des données. Nous allons détailler ces points particuliers.

2.2.1 Déclarations d'objets

Comme dans tout Fortran, l'objet de base d'HPF est le tableau multi-dimensionnel. Il est déclaré avec la syntaxe Fortran 90 [Met87]. Il peut y avoir des tableaux `COMMON` mais ceux-ci semblent « dangereux » à utiliser, du fait de certaines restrictions sur les primitives qui peuvent y être appliqués (entre autre pour les alignements et ré-alignements dynamiques).

Les tableaux peuvent être déclarés dynamiques à l'aide d'une directive (`DYNAMIC`) pour autoriser des changements de leur allocation.

2.2.2 La machine virtuelle

2.2.2.1 La grille virtuelle : TEMPLATE De la structure de données du langage (tableau Fortran) à la machine physique, HPF propose un chemin qui comporte une étape de plus que

la majorité des langages.

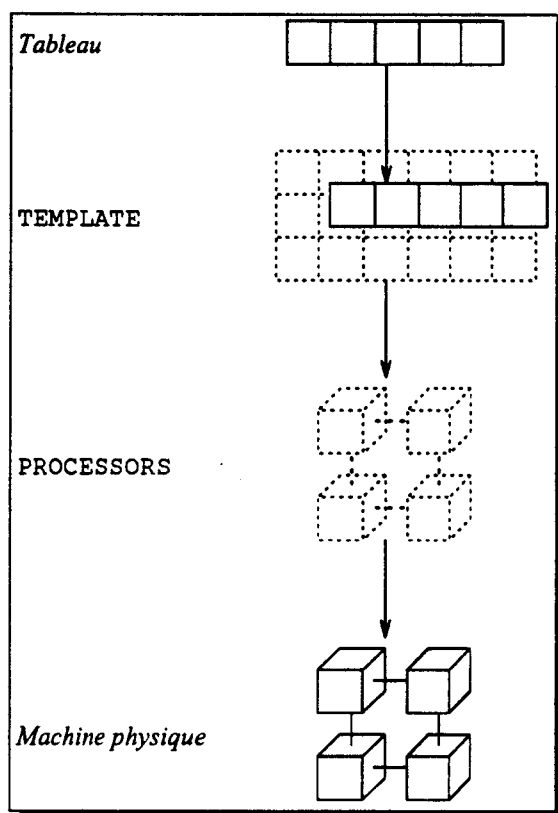


FIG. II.6 - Le modèle de programmation HPF.

Les données de base (tableaux Fortran) sont virtuellement allouées et alignées sur une grille virtuelle : le **TEMPLATE**, qui fait l'objet d'une déclaration de géométrie ressemblant en tout point à un tableau Fortran classique, sans type de données. C'est ce niveau d'abstraction que l'on avait qualifié de machine virtuelle dans la classification proposée en 1.3. Le **TEMPLATE** est un tableau d'éléments sans allocation, sur lesquels sont alignés, à l'aide de déclarations ou de directives statiques ou dynamiques, les éléments des données parallèles du programme.

Par exemple, le programme décrit dans la figure II.7 déclare un **TEMPLATE** de taille 6×4 , grille virtuelle permettant l'allocation et l'alignement de tableaux, comme le vecteur **B** qui sera mis en concordance avec la deuxième ligne de la matrice **A** par la directive d'alignement.

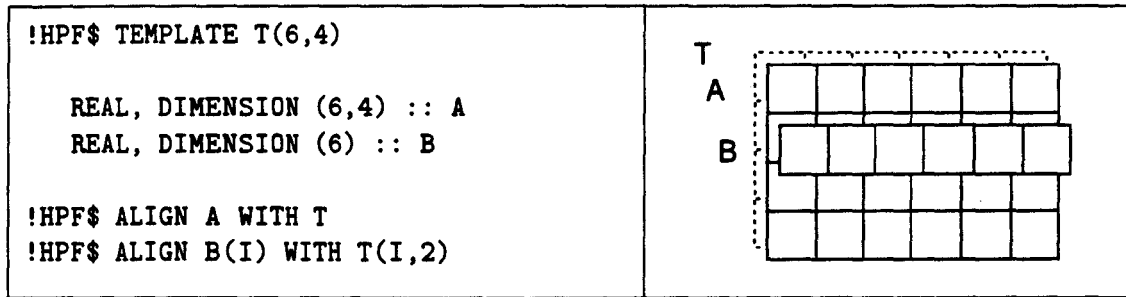


FIG. II.7 - HPF: un exemple de TEMPLATE

Notons que le code de cet exemple aurait pu être écrit sans l'intermédiaire du TEMPLATE T, car toute déclaration de tableau HPF est associée à la déclaration implicite d'un TEMPLATE de même caractéristique géométrique que le tableau. Il est ainsi possible d'aligner un objet par rapport à un autre, ou plus exactement par rapport au TEMPLATE implicite qui y est associé.

Enfin, remarquons que l'alignement de deux objets, par intermédiaire de TEMPLATE ou directement entre eux, n'a qu'un caractère optionnel et dédié à l'obtention de meilleures performances. Le programmeur est autorisé à faire interagir deux objets alignés sur des TEMPLATE différents: le TEMPLATE n'est qu'une directive du langage.

2.2.2.2 Topologie virtuelle Au delà des TEMPLATE, HPF possède une directive PROCESSORS, qui permet de définir une grille (1-,2-,3-...D) de processeurs virtuels. Cette topologie virtuelle n'est pas forcément concordante avec la topologie de la machine cible.

Il est possible de préciser comment seront projetés les TEMPLATE sur la grille virtuelle par l'utilisation de la directive DISTRIBUTE. Cette directive s'applique sur un TEMPLATE explicite ou implicite (on peut directement distribuer un tableau, en considérant le TEMPLATE qui lui est associé implicitement). Le programmeur précise s'il choisit une distribution par cycle (CYCLIC), par blocs (BLOCK), ou mixte en précisant la taille des blocs et le type de projection de ces blocs (BLOCK(N) ou CYCLIC(N)) [For93].

La figure II.8 montre un exemple simple des différentes distributions proposées. On remarque que la distribution est une caractéristique primordiale de l'algorithmique data-parallel. En effet, une mauvaise distribution peut allouer un grand nombre de données sur un processeur tandis que d'autres processeurs se trouvent totalement déchargés.

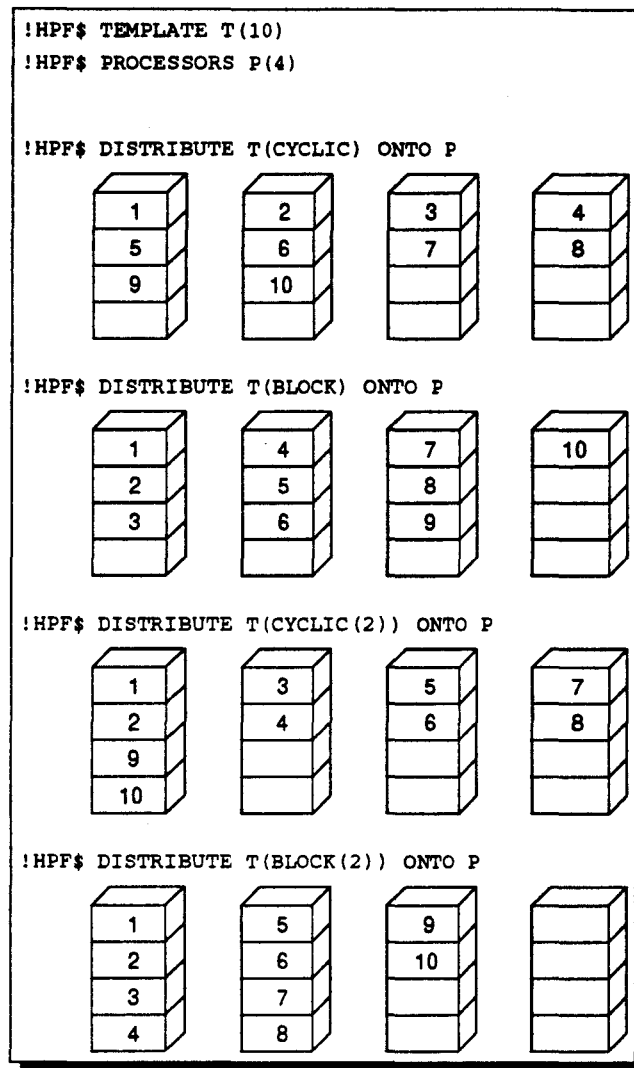


FIG. II.8 - Les différentes distributions d'HPF

Cette notion de PROCESSORS a pour but de se donner une structure virtuelle qui devrait assurer que deux éléments alignés dans le TEMPLATE se retrouvent alignés dans le PROCESSORS et, après la projection des PROCESSORS se trouvent effectivement alloués sur le même processeur physique, reflétant ainsi la volonté du programmeur de favoriser les communications entre deux données, par le regroupement, d'abord logique puis physique, d'opérandes au sein de la même mémoire.

L'avantage d'avoir ainsi défini deux étapes de virtualisation est la possibilité d'exprimer à la fois, au niveau des directives du langage, l'alignement entre les données au niveau des TEMPLATE, suivant des informations liées à l'algorithme mis en place ; et l'alignement entre les différents TEMPLATE sur les processeurs virtuels, suivant des informations liées à l'efficacité du programme.

Or, le nombre de **PROCESSORS** et le nombre de nœuds de la machine physique peuvent être différents. Dans la cas où le nombre de **PROCESSORS** est inférieur au nombre de processeur physique, le compilateur peut choisir d'éclater les données contenues dans un processeur virtuel sur plusieurs processeurs physiques, perdant alors la localité voulue par le programmeur. L'autre alternative consiste à laisser inoccupés certains processeurs physiques, pour assurer la localité des données, ce qui à l'évidence réduit le taux de parallélisme du programme.

Notons qu'HPF propose de nouvelles fonctions intrinsèques qui retournent la topologie de la machine physique, permettent ainsi la déclaration de **PROCESSORS** concordante avec la topologie cible. Dès lors, on peut penser que dans un cas général, la notion de **PROCESSORS** reflète exactement la topologie cible ; il n'y a alors plus qu'un niveau de virtualisation.

Une seule chose est garantie : le programmeur peut spécifier des alignements et expliciter des informations utiles à la projection efficace. Le compilateur peut ensuite en tenir compte, mais peut aussi ne pas les considérer...

2.2.3 L'instruction **forall**

Après les déclarations et alignements des tableaux, le programmeur fait interagir les objets par l'évaluation des expressions faisant intervenir ces objets. Pour exprimer le parallélisme de données, il peut employer les notations « résumées » des expressions en ne faisant apparaître que le nom des objets qui interagissent. Cette solution n'est valable qu'en présence de tableau de même taille. Ainsi, pour faire par exemple la somme de deux vecteurs **M** et **N** de 100 éléments, il suffira d'écrire **M+N**.

Dans le cas de deux objets de géométries différentes, l'écriture d'expressions contenant des descriptions d'objets à l'aide de triplets est possible. On pourra aussi utiliser l'instruction **forall** qui permet de spécifier l'accès aux éléments de chaque objet intervenant dans l'expression par un n -uplet, n étant le nombre de dimensions de l'objet. Cette description par l'intermédiaire d'indices de boucles permet d'exprimer des interactions beaucoup plus générales que les descriptions par triplets. Il est important de noter la sémantique associée à cette construction. Lorsqu'un programmeur utilise un **forall**, il doit assurer que le graphe de dépendances satisfait une certaine condition : il ne doit pas comporter de dépendances entre les données des instructions du blocs pour deux itérations différents. En d'autres termes, le programmeur assure que les calculs pour des valeurs différentes des indices du **forall** peuvent s'exécuter simultanément.

Il existe aussi des solutions syntaxiques (':') permettant de cacher de telles instructions. Pour plus de détails, le lecteur est invité à se reporter à [For93].

En reprenant l'exemple précédent, la somme sur le vecteur **B** des éléments de **A** et **B** peut

s'effectuer par la boucle forall :

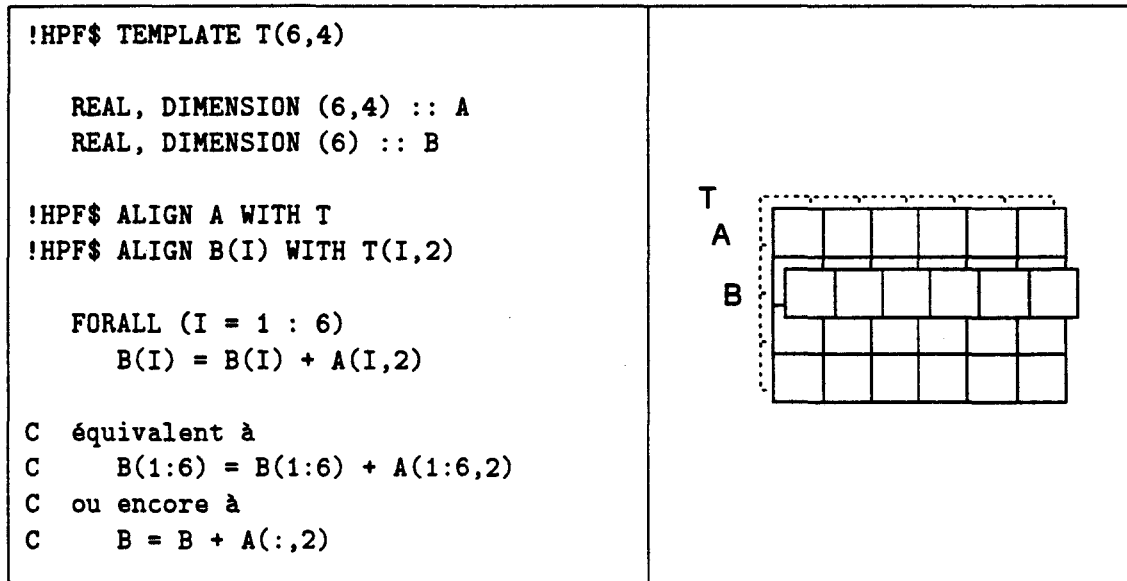


FIG. II.9 - HPF : langage basé sur les indices

L'instruction forall peut comporter plusieurs indices, décrivant ainsi un parcours de tableau à plusieurs dimensions. Elle peut aussi comporter des triplets, décrivant alors le parcours avec un pas différent de 1 pour une dimension donnée. Enfin, il est possible d'opérer un masquage de certains éléments en incluant dans les paramètres de l'instruction forall un test qui suspend l'exécution pour les valeurs pour lesquelles l'évaluation de ce test donne la valeur faux.

Comme nous l'avons évoqué dans l'histoire de l'évolution de Fortran, le fait d'exprimer les informations d'alignements sous forme de directives du langage, implique nécessairement que ces informations ne peuvent jamais être associée à une sémantique. En effet, un programme HPF auquel on soustrait toute directive doit posséder une sémantique équivalente.

Ainsi, dans cet exemple, l'information d'alignement de B sur la deuxième ligne de T est exprimée dans la directive !HPF\$ ALIGN B(I) WITH T(I,2), qui dirige le compilateur mais qui n'a aucune influence sur les données. Dès lors, pour opérer le traitement rudimentaire décrit plus haut, le programmeur doit nécessairement redonner l'information de placement lors de l'accès aux données : l'expression B(I) = B(I) + A(I,2) contient par conséquent la même information que la ligne précédente, il y a redondance.

On peut donc affirmer que le TEMPLATE d'HPF est une machine virtuelle mais qu'il ne constitue pas un référentiel de programmation. En effet, malgré le placement explicité des

objets par rapport à la machine virtuelle, toute expression est basée sur la description des vecteurs à la Fortran. HPF est un langage basé sur les indices.

2.2.4 Communications

Les communications engendrées par un programme HPF sont absolument invisibles, car généralement contenues dans les expressions parallèles. Il n'existe alors aucun modèle typique de communications ; car elles sont fortement dépendantes des données. Ainsi, une expression peut nécessiter, avant l'évaluation arithmétique proprement dite, des communications qui pourraient être modélisées, par exemple, par une translation de plusieurs données d'une rangée de processeurs à la rangée voisine. Ce type d'informations est caché dans les expressions d'indices de description des objets.

Par contre, certaines communications peuvent apparaître à un niveau supérieur quand le programmeur fait appel à des primitives de repositionnement dynamique des données. Ces communications explicites d'HPF ont comme point commun d'être proposées à un niveau haut de la chaîne de développement (au niveau du langage) mais, comme nous l'avons montré, ne sont pas forcément répercutées à un niveau inférieur (au niveau du réseau de communication de la machine physique).

La primitive **REALIGN** permet de changer l'alignement d'un objet par rapport à un **TEMPLATE** (ou par rapport à un autre objet) ; et la primitive **REDISTRIBUTE** effectue une redistribution des **PROCESSORS** sur la machine physique (ou est censée effectuer un tel traitement). Ces deux directives qui introduisent une grande dynamique des objets HPF, ne font pas partie du noyau minimal.

2.2.5 Conclusion sur HPF

HPF possède beaucoup de concepts fortement orientés vers le parallélisme de données et ses problèmes de compilation.

Regrettons d'une part, que l'héritage des Fortran précédents ait amené les concepteurs à proposer un langage figé dans le modèle basé sur les indices, ce qui engendre fatalement la présence de communications implicites qui handicapent le programmeur dans sa recherche d'efficacité.

Regrettons d'autre part, le côté « *on fait ce qu'on peut* » des spécifications de la phase de compilation. Quelques spécifications plus précises du langage (et surtout rendues impératives), comme la projection sur le même processeur physique de deux éléments alignés sur le même

élément d'un **TEMPLATE** ou comme l'alignement impératif de deux objets interagissant dans le même **TEMPLATE**, eurent l'avantage de respecter plus strictement la volonté du programmeur de donner des informations liées à la projection de ses données.

3 Les modèles à flot de données

AU DELÀ de l'existence de machines spécifiques [AKMS85, FLLQ89, QR91, Hug91], le modèle de programmation à flot de données s'adapte bien à de nombreuses architectures de modèle d'exécution différents parmi lesquelles figurent aussi bien des machines synchrones ou asynchrones. De ce fait, cette technique de développement de programme est adoptée non seulement pour les architectures dédiées mais se trouve également projetée efficacement sur d'autres machines.

Le développement peut s'opérer soit directement à l'aide d'un langage dédié à cette programmation (langage dit *systolique*) ou par l'intermédiaire d'un modèle déclaratif basé sur un systèmes d'équations linéaires récurrentes, établissant ainsi une distinction importante de modèles de programmation.

3.1 Le modèle de programmation systolique

Frédéric Raimbault décrit dans sa thèse [Rai94]: « Un algorithme systolique décrit une suite d'opérations s'exécutant sur un ensemble de processeurs identiques qui coopèrent à la réalisation d'un même calcul. Les processeurs du modèle systolique sont interconnectés localement avec un nombre restreint et fixe de voisins. L'ensemble des connexions forme un réseau régulier, de type linéaire, grille, hexagonal etc., où seuls les processeurs situés aux extrémités échangent donnés et résultats avec l'extérieur. »

Ainsi, le modèle de programmation est directement issu du modèle d'exécution systolique, ce qui en fait toute sa puissance et sa simplicité [QR89]. Un algorithme systolique est aisé à décrire, mais il est parfois difficile à un programmeur non averti de mettre au point un tel algorithme à partir d'un problème donné. Il existe donc certaines classes d'applications plus adaptées que d'autres à ce modèle de programmation.

En ce qui concerne la génération de code, on peut retrouver dans ce modèle les mêmes problèmes que ceux que nous avons mis en valeur au chapitre précédent. Ainsi, la phase de virtualisation qui effectue le repliement du réseau systolique est laissée au compilateur. On retrouvera aussi des solutions équivalentes, comme la projection par blocs ou par cycles⁸.

8. éventuellement sous d'autres appellations comme LSGP et LPGS dans [Rai94]

L'environnement ReLaCs

ReLaCS [RL93] a été conçu et développé à l'IRISA de Rennes, Frédéric Raimbault en a fait le cœur de sa thèse. Il s'agit d'un environnement de simulation parallèle pour les algorithmes systoliques. Le langage C-STOLIC issu du langage C est défini pour mettre en œuvre le modèle de programmation systolique à l'intérieur de l'environnement ReLaCS.

Un programme C-STOLIC (comme tout autre langage systolique) consiste surtout à définir un traitement de la cellule de base du réseau. Le point crucial de la mise au point d'un algorithme consiste donc à trouver la façon d'exprimer un traitement incrémental des données qui permette l'éclatement de ce traitement sur les cellules du réseau, reconstituant ainsi le flot global d'instructions dans l'espace traversé par chacune des données traitées.

Une fois ce traitement cellulaire défini, l'implémentation consiste à expliciter trois points :

- caractériser le flot d'entrée sur le réseau, pour permettre l'alimentation de la machine par les données en entrée;
- allouer sur chaque cellule les variables intermédiaires, et exprimer le traitement local à effectuer sur chaque cellule. C'est sur cette phase que le parallélisme de données est exprimé;
- enfin, caractériser le flot de sortie pour récupérer un résultat, généralement retrouvé à l'opposé du point d'entrée sur le réseau.

Le modèle de programmation introduit dans ReLaCS reflète donc bien le modèle systolique, par le développement d'applications souvent spécifiques d'une façon explicite et fortement contrôlée par le programmeur.

3.2 Le modèles basé sur les équations récurrentes

La modélisation mathématique de problèmes peut se faire par intermédiaire d'équations linéaires récurrentes [Mon94]. Certains projets ont consisté à proposer des environnements qui font de cette modélisation un modèle de programmation.

La programmation consiste dans ce cadre à transcrire un problème sous forme d'équations mathématiques qui font apparaître une récurrence et de quelques équations permettant de mettre fin à la récurrence. Généralement, cette première phase est directement issues du problème à traiter et ne présente guère de difficultés au programmeur. L'écriture du programme est ensuite immédiatement obtenue par la traduction de ces équations dans le langage.

L'environnement de programmation doit alors « digérer » ces équations par diverses transformations que l'on peut caractériser formellement pour arriver à un système particulier d'équations. Le but des manipulations, guidées par le programmeur, est de faire apparaître un système qui possède la bonne propriété de ne contenir qu'une unique équation bien particulière.

Cette équation dont une variable d'indice est supérieure à tous les indices des autres variables permet d'obtenir un programme exécutable : elle donne l'état du système à l'instant $t+1$ en fonction des variables à l'instant t . On considère ensuite le facteur temps extrait du système, et on opère la mise en concordance avec l'écoulement du temps inhérent au modèle d'exécution systolique. On obtient ainsi la possibilité de projeter le problème sur une architecture systolique.

Dans ce modèle de programmation, la virtuosité du programmeur se reconnaît à sa façon de guider les transformations du système d'équations pour arriver à en soustraire la forme « exécutable ». Cette phase cruciale fait l'objet d'études en vue de sa prise en charge par des outils de transformations automatiques. Dans l'hypothèse optimiste du développement d'un tel produit, on aboutirait à un modèle de programmation déclaratif dont les outils de mise en œuvre permettraient au programmeur de s'abstraire totalement du développement d'un algorithme pour résoudre un problème modélisé sous forme d'un système d'équations récurrentes.

Le parallélisme de données est sous-jacent aux données manipulées par ces langages. Les inconnues intervenant dans les équations récurrentes sont des données regroupant des éléments homogènes. Les équations qui doivent être satisfaites s'expriment donc globalement sur ces objets data-parallèles.

3.2.1 ALPHA

Le langage ALPHA [Mau89a, VMQ91] est intégré à l'environnement Centaur [Mau89b], tous deux développés à l'IRISA. Le programmeur est amené à utiliser le modèle que l'on vient de décrire.

Une variable ALPHA est un polyèdre convexe de \mathbb{Z}^n appelé *champ*. La dimension représentant le temps n'est pas distinguée de l'espace et les traitements effectués sur les variables sont définis génériquement. Ils peuvent alors s'appliquer sur une dimension spatiale ou sur la dimension temporelle.

3.2.2 PEI

L'environnement PEI est aussi dédié à la manipulation des équations récurrentes. Il permet l'application de règles de réécriture du système sous l'ordre du programmeur. Les données sont ici appelées *champs de données* et représentent à un instant donné une solution possible au système d'équations. Les champs de données ne sont pas forcément de forme particulière et sont totalement dynamiques, dans le sens où le programmeur n'a qu'une idée relative de sa topologie [VP92, Vie94]. L'environnement PEI propose une interface graphique sur laquelle nous reviendrons en présentant les intérêts du graphisme et de la géométrie dans le développement d'applications (cf. 5.1.3).

3.3 Le modèle 8 1/2

8 1/2 est un modèle de programmation abstrait et un langage déclaratif. Il a été conçu et développé par J.-L. Giavitto et J.-P. Sansonnet au LRI [Gia91]. La figure II.10 extraite de [GS94] montre bien l'idée forte d'orthogonalité entre le temps, les valeurs et l'espace. 8 1/2 est fortement inspiré par la donnée des équations modélisant un problème à résoudre. Les variations d'une valeur au cours du temps sont des *trajectoires*, tandis que les variations dans l'espace sont des *champs*. On retrouve cette distinction au niveau de l'implémentation informatique sous les termes respectifs de *streams* et de *collections*.

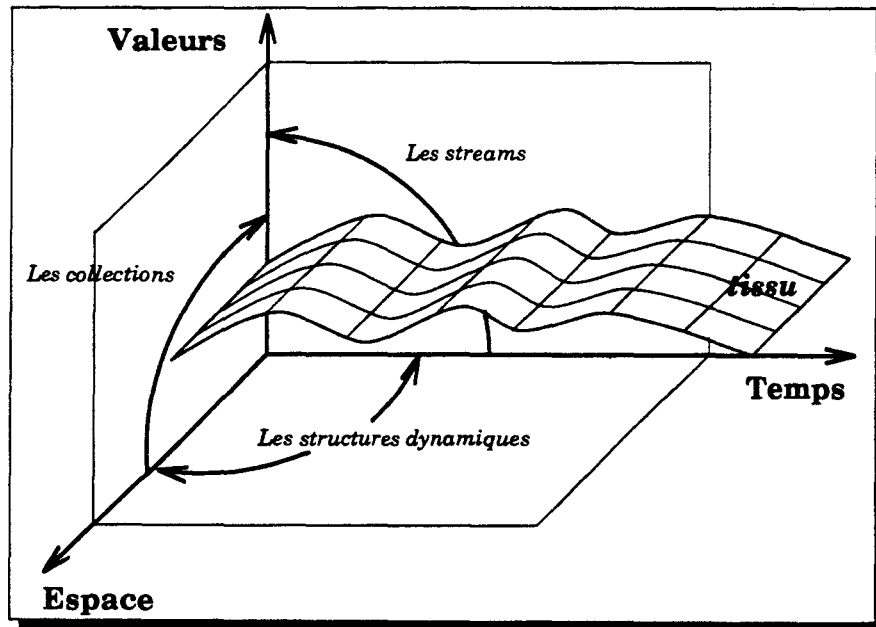


FIG. II.10 - Un tissu du modèle 8 1/2

La notion de temps est très particulière pour le modèle et c'est ainsi que sont définis les *tissus* qui combinent la notion habituelle de parallélisme de données avec la notion de temps, qui fait l'objet de constructions particulières. Ainsi, « un objet défini par un programme 8 1/2 est une surface dans le repère Temps-Espace-Valeurs (cf. figure II.10) » [GS94].

Le programme traite les données par un langage déclaratif dont la résolution consiste à trouver la plus petite solution au système d'équations linéaires [Mic94]. L'implémentation intègre à la fois le parallélisme de données dans le traitement des collections, et l'asynchronisme dans la gestion des streams sous forme d'automates.

4 Les modèles de programmation par structures globales

NOUS avons regroupé ici les langages plus classiques qui proposent un modèle de programmation impératif, qui est basé sur un contrôle des données et de leurs migrations par le programmeur.

Contrairement au modèle à flot de données, entrées et sorties ne sont pas définies dans le modèle, mais sont caractérisées par le programmeur, suivant l'application développée.

La programmation par ces modèles consiste à déclarer un certain nombre de structures de données, et d'y appliquer des traitements par déclenchement d'opérations sur les éléments composant ces données. C'est dans ce sens que nous avons qualifié les données de *structures globales*.

Parmi ces modèles, nous avons distingué dans un premier temps les modèles réguliers puis les modèles proposant une gestion de structures irrégulières, permettant de modéliser des maillages qui étendent la modélisation en grilles multi-dimensionnelles.

4.1 Les modèles réguliers

Ce modèle est à l'évidence le plus répandu car issu directement de l'historique des langages de programmation, comme nous l'avons vu au chapitre précédent. Les structures de données sont des grilles régulières, à l'image des premiers tableaux Fortran.

On doit distinguer deux modèles, suivant le choix effectué sur la sémantique adoptée pour l'expression du parallélisme de donnée (cf. chapitre précédent).

4.1.1 Le modèle régulier à sémantique de référentiel

C* [Thi91], comme beaucoup d'extensions data-parallèles du langage C comme Multi-C [Wav91] ou POMP-C [Par92], est un langage qui considère la sémantique basée sur le référentiel. La programmation par un tel modèle consiste à définir une architecture virtuelle sous forme d'un référentiel cartésien et de considérer chaque point de ce référentiel comme un processeur virtuel.

Comme nous l'avons vu en début de chapitre, le programmeur maîtrise à la fois l'activité des processeurs virtuels et tous les mouvements de données qui sont effectués sur la grille virtuelle.

Le modèle de programmation induit par ce choix de sémantique implique donc pour le programmeur la nécessité de se placer impérativement dans un cadre précis pour le développement de son algorithme. Le problème consiste ensuite à gérer les traitements et les communications à un niveau bas de l'architecture virtuelle.

Dans la cas d'une machine virtuelle d'instanciation, ces langages ne donnent pas la possibilité de faire interagir des données de formes différentes sans préciser explicitement la conversion. En effet, le programmeur se place bien au niveau de la machine virtuelle et les processeurs de cette grille exécutant le même programme, l'allocation d'un objet est uniformément opérée sur la totalité de la grille.

Nous exposerons dans la suite le modèle HELP, que nous avons défini suivant cette sémantique, mais en proposant une machine virtuelle d'alignement.

Ce modèle est certainement celui qui apporte le moins de surprise au programmeur du fait du contrôle pratiquement intégral de l'exécution. C'est sûrement cette raison qui en fait un modèle relativement simple à utiliser pour l'informaticien non initié au parallélisme.

Le compilateur s'occupe de la virtualisation (quand le langage est virtuel) et de la projection. Cette phase ne comporte pas de prise de décision de la part du compilateur quant au positionnement des objets, celui-ci étant issu du modèle de programmation.

4.1.2 Le modèle régulier à sémantique d'indices

Comme nous l'avons présenté, HPF est un langage manipulant des grilles virtuelles pour aider le placement des données les unes par rapport aux autres. Ce modèle est unanimement adopté par toutes les extensions data-parallèles de Fortran, ainsi que par HPC [VB94, VBF94] qui se veut l'équivalent d'HPF pour le langage C.

La sémantique data-parallèle mise en œuvre reste basé sur les indices des éléments traités par rapport à l'origine de l'objet. Par conséquent, les informations de placement sur la grille virtuelle ne servent qu'à guider le compilateur mais ne constituent pas réellement un modèle de programmation.

Pourtant, la possibilité offerte au programmeur d'aligner des objets de différentes formes au sein d'une même machine virtuelle doit permettre d'atteindre de bonnes performances. Le programmeur qui développe une application à l'aide de ces langages doit donc s'efforcer d'utiliser ces outils, suivant le modèle de la machine virtuelle. On peut donc conclure qu'HPF possède tous les éléments requis pour être un véritable modèle de programmation...

4.2 Les modèles irréguliers

Les données régulières sont généralement représentatives d'une partie du problème à traiter. Par exemple, pour un problème de dynamique des fluides, un élément d'une donnée parallèle représentera l'état d'un élément du maillage cubique de discrétisation.

L'expérience montre qu'il est parfois difficile de discrétiser diverses formes pour aboutir à des structures en forme de grille multi-dimensionnelles (en particulier des solides qui interviennent dans le problème, comme une aile d'avion par exemple. Certains ont alors proposé des modèles de programmation pouvant supporter des types de maillages sortant de ce cadre limitatif. Nous qualifions ces modèles de modèles *irréguliers*.

En dehors de la topologie virtuelle irrégulière introduite dans ces langages, l'idée de base du parallélisme de données est conservée: les éléments constituant une donnée sont traités en parallèles quelle que soit la « forme » de cette donnée. On peut donc voir ce modèle de programmation comme une généralisation du modèle précédent.

4.2.1 PARALLAXIS

PARALLAXIS a été présenté à partir de 1989 par Thomas Bräunl [Brä89, Brä90] de l'université de Stuttgart. Ce langage, et le modèle de programmation qui y est inclus, proposent une approche de la programmation data-parallèle originale avec l'absence de toute configuration *a priori* des données.

Le modèle de programmation est basé sur le principe des données allouées sur une machine virtuelle de topologie définie par le programmeur. Cette spécification est constituée d'abord du nombre de processeurs virtuels de la topologie (*configuration*) puis d'une déclaration de réseau virtuel de communications par la donnée des liens qui relient les processeurs virtuels entre

eux. De plus, ces liens peuvent être déclarés unidirectionnels ou bidirectionnels. Il est ainsi possible de travailler sur des architectures régulières ou irrégulières suivant les expressions données pour la création de ces liens (*connections*).

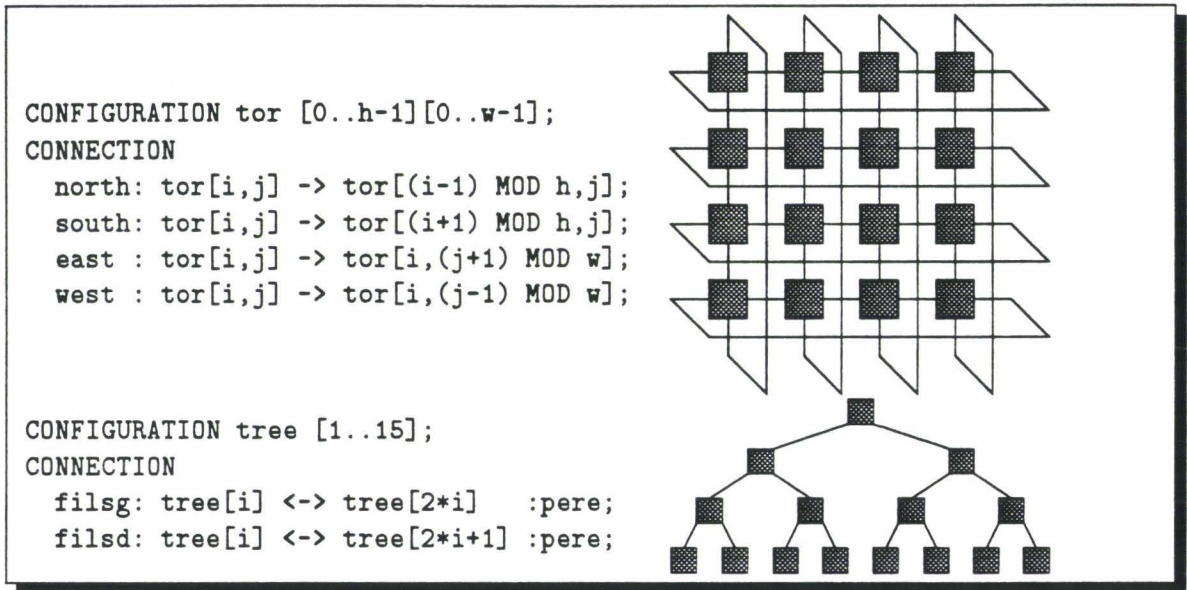


FIG. II.11 - Exemples de topologies PARALLAXIS

La figure II.11 présente deux exemples d'architectures virtuelles possibles à l'aide du modèle PARALLAXIS. La première est une machine dont la topologie est un tore bidimensionnel, la seconde architecture présentée est constituée de quinze processeurs virtuels disposés en arbre binaire.

Les données allouées sur ces machines seront alors elles-mêmes le reflet exact de leur architecture virtuelle. Les traitements calculatoires sont déclenchés sur chacun des nœuds de la topologie et les communications sont explicitées par l'appel de constructeurs qui reprennent le nom des liens pour effectuer des échanges à l'aide de ces liens virtuels.

La programmation d'algorithmes s'exécutant sur des topologies régulières mais non-polyédriques (comme l'arbre) est rendue beaucoup plus facile, mais aussi beaucoup plus lisible, qu'avec des modèles limités aux données en grilles.

Notons enfin que PARALLAXIS est projeté sur un nombre non négligeable de machines séquentielles, vectorielles et parallèles.

4.2.2 HyperC

HyperC [Hyp93] développé par HyperParallel Technologies est le descendant direct de POMPC [Par92]. Il reprend les concepts de base de ce langage régulier en le complétant notamment par la possibilité de travailler sur des données irrégulières définies par le programmeur.

Les *links* décrivent des structures de données homogènes dont la connectivité est explicitée par le programmeur. Ainsi, on a la possibilité de donner pour chaque élément de la structure une liste d'éléments accessibles par la topologie virtuelle ainsi construite. Deux éléments de la même structure n'ont pas forcément le même nombre de liens. Tous les maillages sont ainsi supportés par le langage.

Pour la mise en œuvre de cette irrégularité, les structures de contrôle du flot d'instructions ont été complétées par des constructions capables d'appliquer le parallélisme de données sur les *links* [Par93b]. La figure II.12 représente un exemple de programmation par les *links*. Une collection `cellule` est définie : elle représente une machine virtuelle 10×10 . Au sein de cette machine virtuelle, on définit un voisinage en hexagone par la déclaration d'un tableau de liens pour chaque éléments de la collection (le dernier lien est affecté à 0). Le constructeur `along` permet le parcours de tous les liens pour chaque cellule de la collection.

```

collection [10,10] cellule;
cellule int val,sum;

cellule link cellule VOIS[7];

VOIS[0] = [ . ,(.+1)%10] cellule;
VOIS[1] = [(.+1)%10,(.+1)%10] cellule;
VOIS[2] = [(.+1)%10, . ] cellule;
VOIS[3] = [ . ,(-1)%10] cellule;
VOIS[4] = [(-1)%10,(-1)%10] cellule;
VOIS[5] = [(-1)%10, . ] cellule;
VOIS[6] = 0;

along(VOIS) {
    sum+=[VOIS]val;
}

val=sum/6;
    
```

FIG. II.12 - Irrégularité en HyperC: calcul de convolution hexagonale

HyperC n'exige pas que les liens soient définis dans la même collection que celle des objets qu'ils atteignent (comme c'est le cas dans l'exemple). On peut ainsi décrire un nouveau mode d'accès à une collection, indépendamment de la géométrie de déclaration de la collection.

Enfin, HyperC offre la possibilité de travailler sur des maillages dynamiques (évoluant au cours du déroulement d'un algorithme). Il est alors permis de mettre en œuvre des maillages réellement irréguliers (qui sont parfois difficiles à décrire statiquement).

Le modèle de programmation induit par ces nouvelles constructions du langage n'est pas fondamentalement modifié par rapport au modèle régulier. Le programmeur se place toujours dans le cadre de l'application explicite des communications et des calculs sur des ensembles de données homogènes (*collections*).

5 Le modèle géométrique

DANS cette section, nous montrerons les principes de base du modèle de programmation géométrique que nous avons défini. Dans un premier temps, nous justifierons l'adoption de la géométrie comme support du modèle par son utilisation souvent cachée aux différents niveaux de la résolution de problèmes intensifs : de la pensée du numéricien qui met au point un algorithme au langage qui lui autorise la traduction de l'algorithme. Il semble alors fortement souhaitable qu'un modèle de programmation propose au programmeur des concepts synthétiques proches de cet état de faits.

Nous présenterons ensuite, plus précisément, les notions de base de modèle HELP, à savoir l'hyper-espace et les DPO. Nous étudierons enfin la répercussion des choix des définitions conceptuelles, et de la phase de projection telle que nous l'avons définie au cours de la présentation du parallélisme de données, sur la génération de code et son efficacité.

5.1 L'usage courant de la géométrie

Avertissement Notre but est ici de prouver que la pensée d'un numéricien suit un modèle géométrique et que toute étape de son algorithme est en rapport avec un concept géométrique. Le lecteur ne peut s'attendre à une preuve formelle puisqu'il paraît encore prématuré d'observer le fonctionnement précis d'un enchevêtrement de neurones (qui plus est appartenant à un numéricien !). Nous nous contenterons donc d'évoquer la présence indiscutable d'éléments qui peuvent laisser croire que le modèle géométrique est effectivement à la base de la pensée algorithmique.

5.1.1 Géométrie et bibliothèques numériques

Pour appuyer la thèse qui tent à prouver que l'univers géométrique est universellement répandu, il suffit au lecteur d'ouvrir quelques ouvrages parmi la multitude de ceux qui sont dédiés à l'algorithmique numérique [LPBAQVTSLM]⁹, d'assister à quelque exposé ou cours, ou de consulter les actes de conférences comportant quelques exposés dédiés à des problèmes d'algorithmes généraux ou particuliers ; pour s'apercevoir que de nombreux chercheurs et enseignants s'appuient couramment sur un schéma (plus ou moins soigné) représentant les objets intervenant dans l'algorithme décrit, suivant l'adage qui dit qu'un dessin est plus parlant qu'un discours. Quand le dessin est absent, l'auteur parle en général de notions géométriques simples comme par exemple une ligne ou une diagonale de matrice. Le modèle géométrique est directement issu de cette constatation : il paraît naturel de présenter (et donc certainement d'écrire) un algorithme par le côté géométrique des objets qu'il fait intervenir et par leur placement des uns par rapport aux autres.

En outre, nous avons montré lors du survol des problèmes du Grand Challenge que la mise en œuvre du parallélisme de données permet d'exprimer la résolution des systèmes importants voir complexes. L'usage de bibliothèques dédiées au calcul scientifique est courant. Parmi ces bibliothèques, celles qui se trouvent les plus répandues sont les BLAS-1¹⁰, BLAS-2 et BLAS-3. Elles comportent des primitives nécessaires pour le traitement de vecteurs (comme le produit scalaire), des opérations matrices-vecteurs (comme la multiplication) et des opérations entre deux matrices (comme le produit matriciel). On peut donc déjà justifier l'adoption de la programmation géométrique par le fait établi que les briques de base de la résolution de système linéaires sont elles-même basées sur cette présence de données géométriques.

En outre, les tests de performances des super-calculateurs se veulent le reflet le plus exact possible de l'adéquation des calculateurs à la résolution de problèmes divers. Le classement présenté au premier chapitre [GMS94] est établi sur les tests LINPACK. Ces tests de performances ont été mis au point par Jack Dongarra dans le but d'évaluer les performance réelles de machines diverses [Don93]. Ils sont constitués par la résolution d'un système dense d'équations linéaires. Là encore, on peut donc affirmer que ce genre d'algorithmes repose sur une système fortement orienté vers des algorithmes qui manipulent des données géométriques. La dimension géométrique est donc reconnue comme représentative d'une grande classe d'applications.

9. Le Premier Bouquin d'Algorithmique Qui Vous Tombe Sous La Main

10. Basic Linear Algebraic Systems

5.1.2 Géométrie et pensée algorithmique

Dans le chapitre précédent, nous avons montré que les algorithmes numériques fortement exigeants quant à la puissance de calculs mettaient principalement en œuvre des données homogènes issues du problème à traiter, souvent par une phase de discrétisation à la charge du physicien. Ainsi, un algorithme comporte une succession de phases importantes allant de la déclaration des variables aux traitements associés à la résolution proprement dite.

La majorité des applications exigent l'utilisation de structures de données régulières pour la mise au point d'un algorithme. L'utilisation de tout langage data-parallèle implique alors la déclaration de tableaux ou d'objets équivalents qui pourront être le support du parallélisme. Pour reprendre cette habitude incontournable en l'intégrant dans un univers géométrique, le modèle géométrique doit lui-même proposer d'appliquer le parallélisme sur des données régulières. Le choix d'un type d'objets explicitement parallèles permet une plus grande maîtrise du programmeur vis-à-vis de la distinction des parties parallèles ou séquentielles de son programme, et facilite le travail du compilateur qui est alors dispensé d'une phase de décision entre l'exécution séquentielle ou parallèle.

Une fois les objets parallèles déclarés, un algorithme décrit une succession de traitements qui font interagir ces données. Dans tous les cas, le programmeur connaît les traitements qu'il veut effectuer entre ces objets. De cette connaissance, il est capable de préciser la façon de placer les objets les uns par rapport aux autres pour les mettre en rapport afin de déclencher les opérations. Par exemple, faire interagir un vecteur et la première ligne d'une matrice permet de déduire l'information d'alignement entre ces deux objets.

Pour ce faire, un modèle de programmation doit définir un cadre virtuel qui servira de référence pour l'alignement des données. Les points de ce référentiel ne constituent qu'une entité relative au placement des objets et ne sont pas eux-même le support de données. On dissocie ainsi les données effectives du problèmes et le référentiel dans lequel vont s'effectuer les traitements.

Au delà de l'allocation et de l'interaction des objets, un algorithme numérique peut spécifier qu'une donnée parallèle fait l'objet d'une interaction par rapport à un objet puis par rapport à un autre objet (ou même une autre partie du premier objet). Par exemple, lors de la multiplication matrice/vecteur, le vecteur opérande peut successivement interagir avec la première colonne puis la seconde colonne d'une matrice, jusqu'à la dernière colonne. Le modèle qui supporte ces algorithmes doit donc impérativement proposer une possibilité d'exprimer des migrations d'objets à travers le référentiel. Pour garder constamment l'idée de géométrie cartésienne, on pourra modéliser ces migrations par des déplacements géométriques appliqués globalement aux objets, comme des translations ou des symétries.

Une fois le placement des données réalisé par le programmeur, l'écriture des expressions purement calculatoires, qui sont la base même du problème à traiter, doit pouvoir s'exprimer simplement. Pour cela, le modèle basé sur la sémantique du référentiel (voir précédemment) semble être le plus adéquat pour garder la cohérence avec la modélisation géométrique latente. De plus, nous avons montré que ce choix permet l'écriture de programmes en évitant les redondances d'informations dans le code source, en maîtrisant plus précisément les intermitences des phases de calculs et de communications. Rappelons que seule l'adoption de la sémantique du référentielle a permis de prouver des résultats théoriques dans le domaine de preuves de programmes data-parallèles.

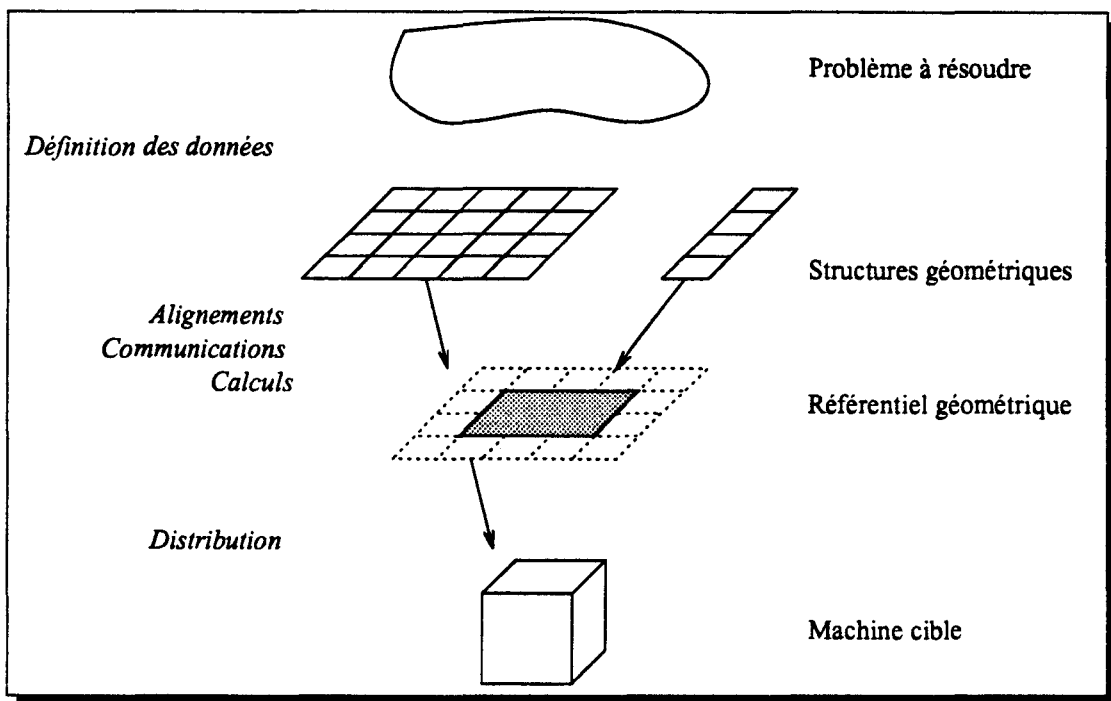


FIG. II.13 - Les différentes phases de l'élaboration d'un algorithme

En reprenant les définitions exposées dans le chapitre précédent, on peut dire que le modèle géométrique met à disposition du programmeur une machine virtuelle permettant l'allocation et l'alignement des objets les uns par rapport aux autres ; que les objets parallèles manipulés sont des objets explicites et que le modèle possède une sémantique basée sur le référentiel.

5.1.3 Géométrie et visualisation graphique

La mise au point d'un programme numérique nécessite souvent une visualisation des structures de données manipulées par une schématisation des objets parallèles du programme et

de leur placement sur la machine virtuelle¹¹. L'adoption d'un modèle explicite sur ce point permet le développement d'outils d'aide à la programmation.

C'est en ce sens qu'ont été entrepris des travaux relatifs à la création d'un environnement de développement de code qui permet à l'utilisateur d'écrire le code de son programme en visualisant graphiquement le placement et l'interaction de ces DPO. Ce sont les travaux d'Akram Benalia, exposés dans sa thèse [Ben95].

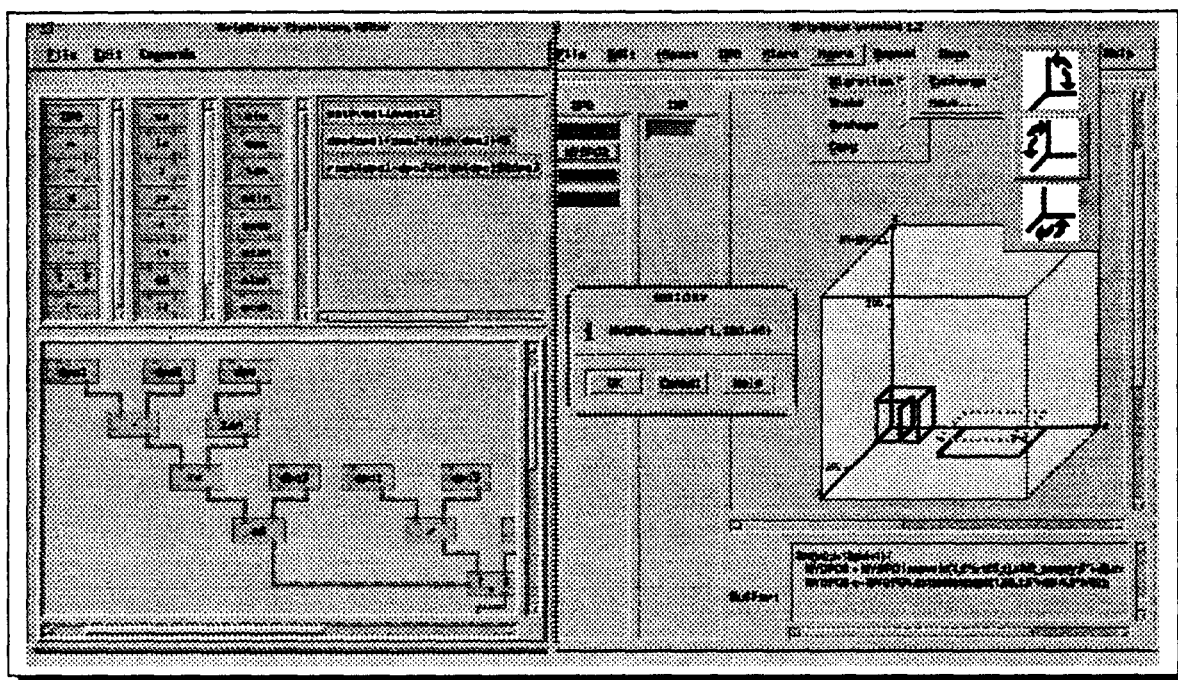


FIG. II.14 - Géométrie et environnement de programmation : HelpDraw

Dans l'environnement HelpDraw [BDLM92a, BDM93], la priorité est mise sur la possibilité donnée au programmeur de générer du code exécutable directement par la modélisation géométrique. On a ainsi l'espoir d'obtenir un environnement capable de rendre le parallélisme de données accessible à un numéricien, sans même que celui-ci est besoin d'apprendre un langage de programmation. Voir figure II.14. La partie génératrice d'expressions de l'environnement HelpDraw permet de compléter la construction des programmes par le graphisme.

Outre l'environnement HelpDraw, il existe d'autres travaux relatifs au développement d'environnements de programmation proposant une interface graphique d'aide au programmeur. Nombreux sont ceux, parmi eux, qui visent à proposer une interface de modélisation

11. ou même physique, dans le cas du développement d'un programme dans un langage non-virtuel

graphique du graphe de dépendances d'un programme Fortran. D'autres plus généralistes, et pas forcément dédiés à la programmation parallèle, traitent les flots de données, comme AVS. Même si ces travaux ne sont pas directement en relation avec un modèle géométrique de programmation, on voit bien que le graphisme ne constitue pas seulement un moyen d'accéder à la convivialité mais aussi un moyen d'exprimer des informations relatives à un programme en cours de développement.

Enfin, l'environnement PEI (cf. 3.2.2), proposé par l'équipe de Guy-René Perin à Besançon est un environnement dédié à l'aide à la résolution de systèmes d'équations récurrentes. Dans ce projet, le graphisme est utilisé pour donner au programmeur une idée intuitive, la plus naturelle possible sur l'état courant de ses champs de données. Là aussi, on estime qu'un dessin est une information importante qui permet de comprendre plus facilement l'état courant du développement d'un programme.

Notons aussi que certains des environnements dédiés aux langages que nous avons évoqué (environnement 8,5 pour le langage 8 1/2) proposent aussi une partie gérant la géométrie des objets. Il existe aussi d'autres environnements graphiques permettant la description géométriques des opérations d'un algorithme. C'est le cas, par exemple, de ALEX qui propose une manipulation des données par un environnement graphique de programmation associé à un langage fonctionnel [KTC⁺90].

5.1.4 Géométrie et lisibilité

Lorsqu'un programmeur doit faire face à un problème de lecture de code, le niveau de lisibilité de celui-ci peut faciliter ou rendre plus pénible la tâche de déchiffrement. Même si un programme écrit dans un langage à parallélisme de données comporte de bonnes propriétés vis-à-vis de la clarté d'écriture, le fait de manipuler des données sous forme de tableaux (ou équivalents) engendre un exercice périlleux de modélisation géométrique à la charge de celui qui veut comprendre le programme.

Nous venons de montrer que les environnements graphiques permettent un développement facile. Il en est de même pour la lecture d'un code existant. Les développements prévus de l'environnement HelpDraw, par son côté analyseur de code existant, doit permettre de pallier cette difficulté en se chargeant de la conversion du langage source vers la visualisation géométrique. On adapte alors le vieux principe connu : « un petit dessin vaut mieux qu'un gros programme Fortran ».

5.2 Adéquation de la géométrie aux machines parallèles

Le modèle géométrique dont nous venons de donner les caractéristiques majeures n'est pas uniquement un modèle de programmation pour exprimer simplement le parallélisme de données, mais peut aussi concerner le niveau inférieur du développement d'applications, à savoir la génération de code pour machines parallèles.

L'adoption d'une sémantique basée sur le référentiel sépare naturellement le cadre d'activité et les références aux éléments d'objets qui interagissent (on n'extrait plus le cadre d'activité en fonction des index de ces éléments). Le modèle d'exécution induit par cette sémantique consiste alors à définir dans un premier temps l'activité sur certains points du référentiel géométrique, puis d'évaluer localement à tous les points actifs l'expression courante. Ainsi, l'indice respectif des opérandes présents sur ce point n'a plus de signification, le processeur virtuel opère sur les données présentes. Dès lors, une expression calculatoire n'engendre aucune communication implicite. On « tombe » ainsi dans l'asynchronisme des processeurs virtuels qui sont capable d'évaluer leur propre instance de l'expression, totalement indépendamment des autres processeurs virtuels. Certains qualifient ce modèle d'exécution de SPMD [Hyp93].

C'est pour cette raison que l'on peut dire que le modèle géométrique est en adéquation avec le matériel du parallélisme. Une séparation communications/calculs permettra toujours une plus grande maîtrise de l'exécution et par conséquent l'obtention probable de meilleures performances. Par exemple, un problème souvent discuté dans la communauté des développeurs de compilateurs data-parallèles pour machines asynchrones est le recouvrement des communications et des calculs dans le code généré par le compilateur [TMP]. Il est évident qu'un tel problème ne pourra se résoudre qu'à partir d'une phase primordiale d'identification précise de ces portions de communications et de ces portions de calculs : on ne peut pas introduire d'optimisation si on ne connaît pas les endroits précis où elles doivent être introduites. Les langages explicites comme HyperC ou C* permettent, de par leur syntaxe (communications au voisinage virtuel), de retrouver facilement ces informations, mais les langages implicites (extensions de Fortran) sont sur ce point beaucoup plus difficiles à compiler. Le modèle géométrique est sur ce point on ne peut plus rationnel.

5.3 Le modèle HELP

Nous venons de montrer que le modèle géométrique est principalement orienté vers le développement d'applications numériques à l'aide de la géométrie des données et des traitements. Nous définissons maintenant un modèle à l'image des nécessaires caractéristiques que nous venons de mettre en évidence : le modèle HELP¹² dont la notion géométrique de base est l'hyper-espace.

12. Hyper-Espace et Langage Parallèle

5.3.1 L'hyper-espace

Le programmeur HELP peut déclarer un ou plusieurs hyper-espaces qui vont constituer le(s) cadre(s) de toutes les opérations de l'algorithme à mettre en place, conformément au modèle géométrique que nous venons d'exposer. Il va pouvoir ainsi se définir un repère cartésien de points de coordonnées strictement positives.

D'après le problème à résoudre, il peut choisir de faire évoluer ses données dans un espace à deux, trois dimensions, ou plus. De même, la taille de chacune des dimensions est définie d'après des critères dépendant de l'algorithme.

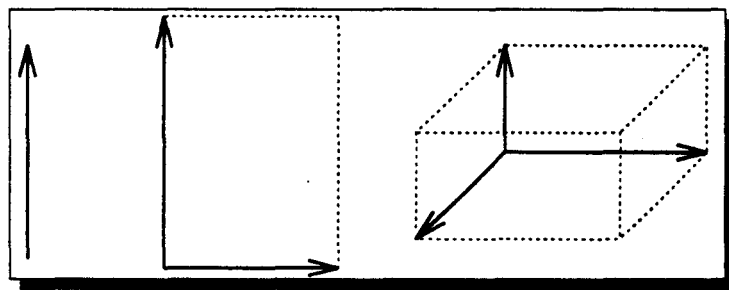


FIG. II.15 - Quelques hyper-espaces

5.3.2 Les objets data-parallèles

Les données de l'algorithme sont décrites sous forme de sous-espaces sur lesquels vont pouvoir s'effectuer les traitements. Le choix a été fait de caractériser explicitement le fait qu'un objet parallèle va faire l'objet du déclenchement d'opérations parallèles. Nous avons donc défini un nouveau type de données : les DPO qui sont des pavés compacts alloués sur les points de l'hyper-espace. Chaque point de l'hyper-espace est virtuellement le support mémoire d'un élément de l'objet. À la différence d'autres langages issus de C (comme C*), les objets ne sont pas forcément alloués sur tout l'hyper-espace : on se place dans le cadre d'une machine virtuelle d'allocation (et non d'instanciation).

Comme nous l'avons vu précédemment, le programmeur est capable de déduire de son algorithme un placement des DPO entre eux afin de mettre en concordance les éléments devant interagir sur les mêmes points de l'hyper-espace. Pour ce faire, un DPO possède une allocation dans l'hyper-espace. L'hyper-espace constitue donc une machine virtuelle qui offre la possibilité de spécifier les alignements de données. L'intérêt principal de ce choix réside en la possibilité offerte au programmeur de faire interagir des données de formes différentes au sein d'une même machine virtuelle, sans avoir à effectuer des transferts ou des conversions de type.

Au delà des objets parallélépipédiques, notons que le langage qui va reprendre les concepts du modèle HELP permettra la manipulation d'objets moins réguliers comme une diagonale de matrice (cf. chapitre suivant 1.2.3).

5.3.3 Domaine de conformité

Pour se donner un modèle de programmation concordant avec le modèle géométrique, nous avons choisi de distinguer au niveau du langage le domaine sur lequel les calculs vont être déclenchés des objets intervenant dans ces calculs. Ce domaine est appelé *domaine de conformité* et repose aussi sur la géométrie de l'hyper-espace. Ainsi, l'évaluation d'une expression devra se faire à la condition impérative de vérification de la règle de conformité introduite par le modèle :

Une expression C-HELP fait interagir des DPO qui doivent être impérativement alloués sur les points de l'hyper-espace associés au domaine de conformité.

Ainsi, quand deux objets interviennent dans une expression, ce ne sont pas ces objets qui constituent le support géométrique de l'activité, mais bien certains des points de l'hyper-espace sur lesquels sont alloués ces objets et qui constituent le domaine de conformité. Cette règle donne au modèle toute sa signification. Le référentiel de l'hyper-espace n'est pas uniquement une entité qui guide le compilateur (comme l'est le **TEMPLATE** d'HPF), mais constitue un réel support de l'activité du parallélisme de données. Dans le chapitre dédié à la compilation, nous nous attacherons à prouver qu'un tel modèle permet de simplifier et de rendre plus efficace la génération de code par le compilateur.

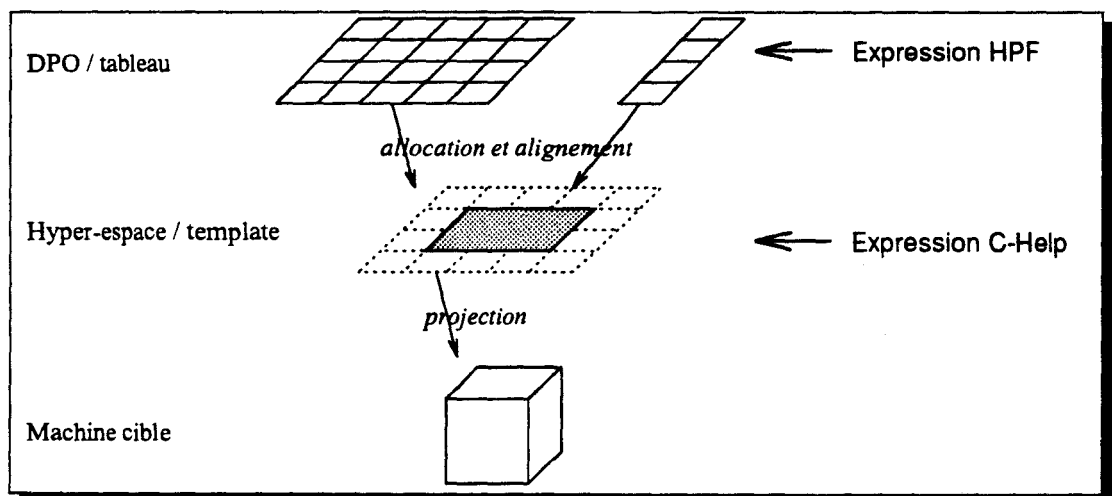


FIG. II.16 - Le référentiel support d'activité

5.3.4 Vue microscopique

Une expression fait interagir des données conformes (qui vérifient la règle de conformité). Le cadre d'activité est ainsi factorisé et se retrouve au niveau des points de l'hyper-espace. On a donc un modèle qui correspond, dans la classification proposée à la section précédente, aux langages basés sur un référentiel.

Lors de l'évaluation d'une expression, le cadre de l'activité est explicité par le programmeur, ou calculé par rapport à cette expression, mais indépendamment des indices de chaque élément appartenant aux DPO qui vont intervenir. Sur chaque point actif de l'hyper-espace, une référence à un objet sera donc traduite par l'accès à l'élément de cet objet qui se trouve sur le point, quelle que soit sa position relativement à l'origine de l'objet. Le référentiel de l'hyper-espace est bien considéré comme une machine virtuelle, contrairement à son homologue HPF, le **TEMPLATE** qui ne concerne que la phase de distribution des données sur les processeurs.

L'impérative nécessité d'aligner les DPO entre eux avant le déclenchement d'un traitement doit pouvoir être allégée par la possibilité donnée d'explicitier le cadre d'exécution d'une expression à évaluer. Le programmeur est en mesure de préciser que la portée d'une expression se limitera à une certaine zone de l'hyper-espace.

Pour ce faire, nous avons défini la notion de domaine contraint. Son explicitation est donnée par le programmeur sous forme d'une référence à un DPO. L'évaluation des instructions qui suivent est déclenchée exclusivement sur le domaine d'allocation de ce DPO. Il est ainsi possible d'explicitier une géométrie pour l'activité de la machine virtuelle.

Le modèle **HELP** offre ainsi le pouvoir de distinguer d'une part les expressions découlant du problème à résoudre, et d'autre part l'activité qui dépend de l'algorithme et des structures de données mises en place.

5.3.5 Vue macroscopique

La vue microscopique permet le déclenchement des phases calculatoires de l'algorithme. Pour les phases de communications, indispensables au développement d'une quelconque application, le modèle **HELP** propose une vue macroscopique sur les objets du programme.

Comme nous l'avons évoqué en début de chapitre, la modélisation géométrique des déplacements d'objets permet d'obtenir une facilité de programmation et une lisibilité du code écrit, ce qui est plutôt rare dans les nombreux langages scientifiques.

Le programmeur applique donc des primitives géométriques (translations, rotations...) à des objets et produit ainsi de nouveaux objets. Le modèle HELP propose, par ce biais, une dynamicité complète des objets : en position, taille, nombre de dimensions. Ce choix permet de substituer aux expressions manipulant des indices (comme on les rencontre en Fortran ou dans beaucoup d'autres langages) une expression géométrique qui rend compte de la vision globale sur la migration de l'objet. Voir figure II.17.

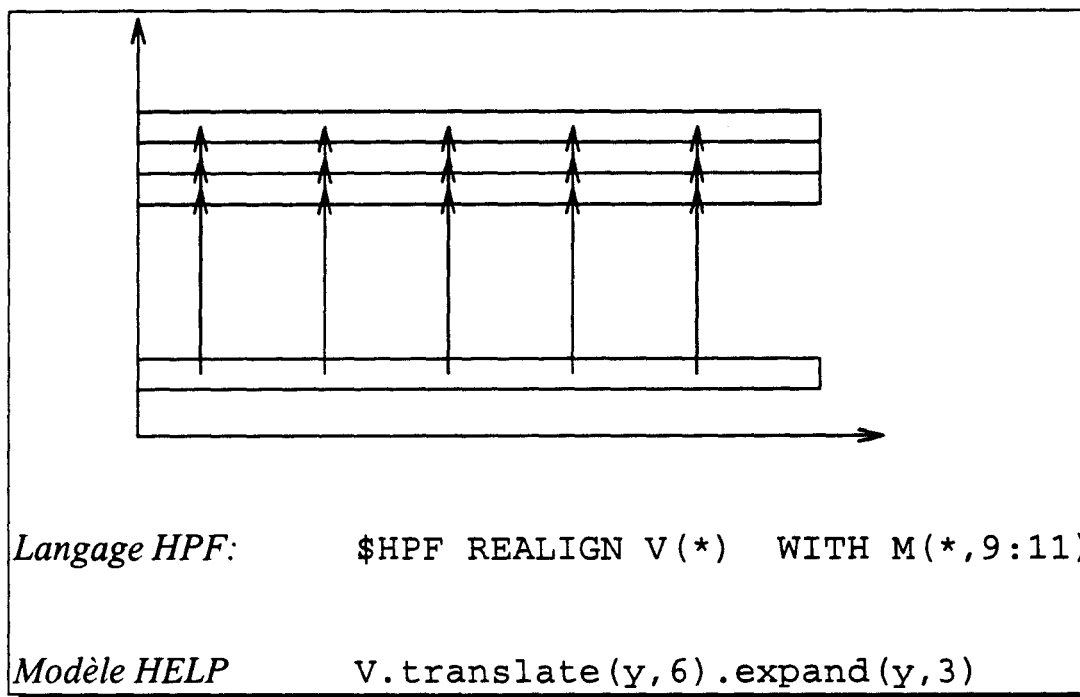


FIG. II.17 - La modélisation géométrique des communications

5.4 Projection

Après avoir exposé la philosophie de programmation HELP, nous exposons ici la justification du modèle et des contraintes de conformité d'un point de vue génération de code.

5.4.1 Stratégie de projection

Le programmeur doit spécifier un algorithme de projection de ses données par rapport aux processeurs de la machine cible. Pour cela, HELP fournit la possibilité d'exprimer simplement certaines informations qui ne peuvent être déduites du code que par l'intermédiaire d'outils de parallélisation automatique. Les travaux dans le placement automatique des données consti-

tuent à eux seuls un domaine particulier, dont les retombées sont primordiales quant à la réutilisation de codes existants [Fea93].

Pour notre part, nous avons choisi de garder l'idée de produire un environnement explicite de programmation en laissant à la charge du programmeur ces spécifications.

Outre le fait d'avoir choisi des modes de distributions réguliers, comme la distribution par cycles, par blocs de taille donnée, comme dans de nombreux langages, il est important de remarquer que ces directives de distributions sont attachées à l'hyper-espace, et non aux objets du langage. Le programmeur spécifie ainsi un mode de projection par dimension de son hyper-espace. Le modèle de la conformité logique est ainsi conservé lors de la phase de distribution.

Il a aussi la possibilité de préciser un ordre de priorité entre ces dimensions sous forme d'expression arborescente pour indiquer au compilateur la façon de projeter l'hyper-espace sur la machine cible. Par exemple, la projection d'un hyper-espace tri-dimensionnel sur une machine plane se fait par projection de la dimension la moins prioritaire en mémoire. L'attribution de la même priorité à deux dimensions pourra, suivant la topologie cible, autoriser le partage d'une dimension physique de la machine par ces deux dimensions logiques de l'hyper-espace. On obtiendra ainsi une utilisation optimale du parallélisme de la machine cible.

Ces directives permettent au programmeur de choisir une distribution parmi une grande variété de distributions différentes, selon son algorithme.

5.4.2 Influence de la conformité

La justification principale du modèle quant à l'efficacité réside dans la synthèse des trois éléments présentés jusque là : la conformité des objets, l'adoption d'une sémantique data-parallèle basée sur le référentiel et l'attribution des directives de projections à l'hyper-espace.

En effet, puisque la sémantique choisie pour l'évaluation des expressions entraîne l'association de l'activité à l'hyper-espace, les références aux opérandes se font localement au point de cet hyper-espace. D'autre part, la projection des données se faisant par l'intermédiaire des dimensions de l'hyper-espace, ce sont les points de cet hyper-espace qui sont projetés. Par conséquent, la conformité des données se retrouve physiquement sur la machine cible.

Une expression microscopique ne peut donc jamais engendrer de communications ni au niveau de l'hyper-espace, ni au niveau de la machine. Le modèle de programmation HELP propose ainsi une séparation effective entre les calculs (vue microscopique) et les communications

(vue macroscopique).

CONFORMITÉ + PROJECTION PAR POINTS = LOCALITÉ

6 Conclusion

LE PARALLÉLISME de données permet l'obtention de bonnes performances tout en favorisant l'écriture d'un programme par une personne non spécialiste du parallélisme, par le fait de garder un déroulement séquentiel du code. La simplicité de l'idée du modèle général de ce type de parallélisme nous a permis de mettre clairement à jour les problèmes de base quant à la compilation des langages l'utilisant.

Nous avons montré que Fortran a été complété par de nouvelles notions orientées vers la résolution de ces problèmes, mais que l'héritage de ses versions précédentes le confine dans un modèle basé sur les indices qui ne lui permet pas de proposer au programmeur un contrôle complet de l'exécution, notamment des communications engendrées implicitement.

L'étude des modèles à parallélisme de données a distingué deux grandes familles de langages : le modèle à flot de données et le modèle à structures de données globales.

Nous avons montré que la géométrie régulière des données pouvait constituer un modèle de programmation qui est en adéquation avec toutes les étapes du développement d'applications scientifiques. À partir de ces constatations, nous avons défini un modèle de programmation basé sur la notion de géométrie : le modèle HELP dont l'entité de base (l'hyper-espace) va permettre à la fois le développement aisé d'applications par une vision proche de l'algorithmique numérique, et l'obtention de bonnes performances par la séparation entre communications et calculs introduite dans le modèle.

Chapitre III

Le langage C-Help

LE LANGAGE C-HELP est une implantation du modèle à parallélisme de données géométrique HELP introduit dans le chapitre précédent. Il permet la mise en œuvre du parallélisme de données de façon explicite par la mise à disposition du programmeur d'un faible nombre de déclarations et de primitives spécifiques. Après la présentation du modèle, ce chapitre sera consacré à la présentation du langage qui intégrera les concepts du modèle, permettant au programmeur le développement d'un algorithme en suivant la méthodologie latente au modèle géométrique.

Dans un premier temps, nous présenterons les déclarations des objets virtuels (hyper-espace) et des objets data-parallèles (DPO) du langage. Nous présenterons ensuite la décomposition hiérarchique des expressions qui nous permet de distinguer les deux points de vue offerts au programmeur. Avec la vue *microscopique* des données, en particulier l'écriture de fonctions locales aux points de l'hyper-espace, le programmeur a la possibilité d'écrire des parties de code totalement asynchrones. La section suivante traitera de la vue *macroscopique* dans laquelle les déplacements de données sont modélisés sous la forme de migrations géométriques internes à l'hyper-espace. Enfin, nous présenterons les possibilités d'écriture de fonctions de différents types, notamment les fonctions à géométrie générique qui permettront, entre autres, l'écriture de bibliothèques C-HELP.

Pour des considérations techniques précises sur le langage, comme sa grammaire, le lecteur est invité à se reporter au manuel de référence C-HELP [Laz94a].

Le développement du langage C-HELP nous a permis de valider l'idée du modèle HELP par une production effective d'un langage et de son compilateur. La mise en œuvre de ces outils a été entreprise dans ce but unique. Le langage C-HELP peut être défini comme une extension du langage ANSI-C [KR88] aux concepts du modèle géométrique HELP.

Avertissement

Le symbole \triangle sera utilisé dans ce chapitre pour mettre en évidence les conventions de programmation imposées par le langage. Le programmeur devra utiliser les constructions du langage tout en s'assurant de la validité de certaines conditions. Le non-respect de ces règles de programmation entraîne généralement un comportement indéfini du programme.

1 Les objets du modèle HELP

PAR ANALOGIE à la définition de type du langage C, nous avons choisi de considérer, au **P** niveau du langage, l'hyper-espace comme une caractéristique d'ordre « information de type » attachée aux DPO (cf. figure III.1).

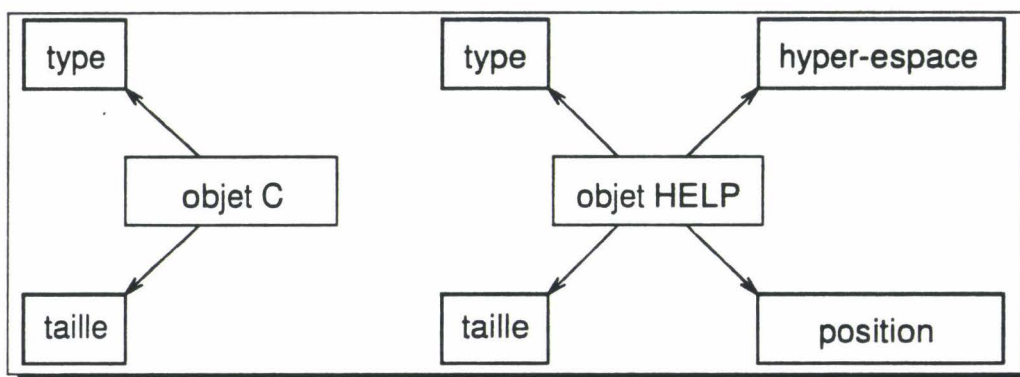


FIG. III.1 - Les objets du langage C-HELP

1.1 Hyper-espace

Comme présenté lors de la caractérisation du modèle HELP, un hyper-espace est un référentiel cartésien de points de coordonnées strictement positives. Sa géométrie est précisée par son nombre de dimensions, les caractéristiques de ces dimensions et l'ordre de priorité de parallélisme qui ordonne ces dimensions pour l'étape de projection sur la machine physique.

La déclaration est précédée du mot-clef `hspace` et comporte le nom de l'hyper-espace créé suivi des informations de géométrie et des informations de distribution des dimensions sur la machine physique. Ainsi, la déclaration contient une liste d'informations liées à chaque dimension : chacune d'entre elles comporte un identificateur pour pouvoir être référencée conformément au modèle géométrique (entre autres lors d'une migration géométrique, cf. 3.3), et ses caractéristiques propres suivant qu'elle soit primaire ou secondaire.

1.1.1 Dimensions primaires

Les dimensions primaires fournissent au programmeur un repère géométrique dans lequel les coordonnées cartésiennes d'un point servent d'accès aux données allouées sur ce point. La manipulation des structures régulières est disponible au programmeur par l'expression de notions géométriques simples, toujours basées sur cet hyper-espace.

La déclaration d'une dimension primaire doit comporter toutes les informations nécessaires à sa construction lors de la phase de compilation : l'hyper-espace est un cadre statique. Dès lors, le programmeur doit impérativement spécifier :

- le nom de la dimension ;
- la taille de cette dimension (en nombre de points) par une expression entière évaluable à la compilation.

La dimension primaire ainsi déclarée constitue un axe du repère cartésien de l'hyper-espace. Chaque point de l'hyper-espace comporte sur cet axe une coordonnée incluse dans l'intervalle [1..taille_de_la_dimension] (cf. figure III.2).

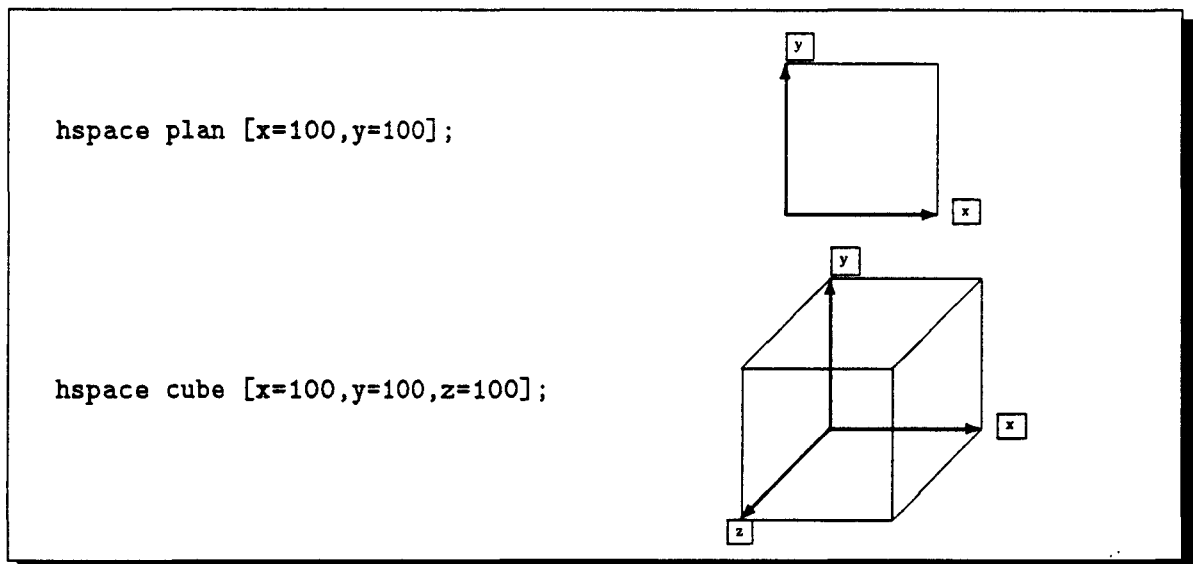


FIG. III.2 - Exemples de déclarations d'hyper-espaces comportant 2 ou 3 dimensions primaires

Au delà de ces informations purement géométriques, le programmeur peut préciser pour chaque dimension primaire une information appelée *facteur de bloc* qui sera exploitée par le compilateur pour la phase de projection (cf. 1.1.3).

1.1.2 Dimensions secondaires

Une dimension secondaire permet la manipulation d'objets par intermédiaire d'une diagonale principale de l'hyper-espace. Le programmeur a la possibilité d'allouer et de référencer les objets de son programme par rapport à ces diagonales. Cette possibilité offerte par C-HELP est souvent utilisée par les algorithmes d'algèbre linéaire.

La déclaration d'une dimension secondaire est écrite sous forme d'une liste de dimensions primaires déjà déclarées. \triangle Une dimension primaire ne peut apparaître qu'une seule fois dans cette liste. On donne de cette façon au programmeur la possibilité de définir un axe diagonal de l'hyper-espace, composé des axes sur lesquels s'effectue la construction. La figure III.3 présente des exemples de telles déclarations. Dans un hyper-espace cubique (cube), trois diagonales (e, f et g pour chacun des plans perpendiculaires) sont déclarées en composant deux à deux les dimensions primaires, et la diagonale principale (d) est déclarée par composition des trois dimensions de l'hyper-espace.

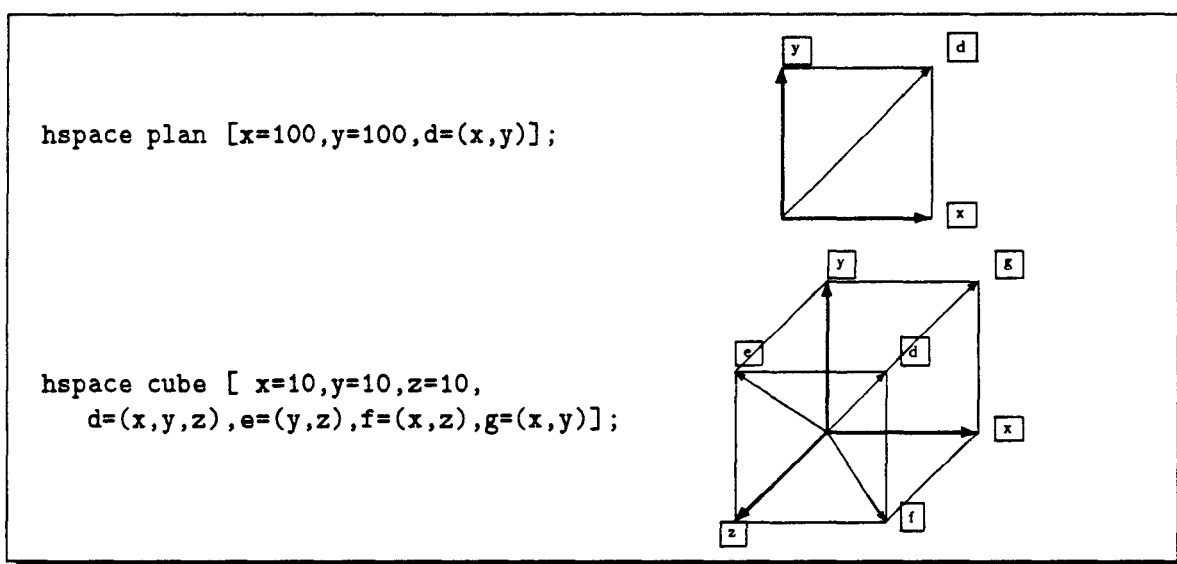


FIG. III.3 - Exemples d'axes secondaires

Une dimension secondaire ne possède jamais d'information de facteur de bloc, ni n'apparaît dans l'ordre de priorité des dimensions de l'hyper-espace. Ainsi, la phase de projection ne sera effectuée qu'en tenant compte des informations liées aux axes primaires. Un axe secondaire ne constitue qu'une possibilité supplémentaire pour référencer les points de l'hyper-espace. Les opérations d'allocation (déclarations) et de migrations de données suivant ces dimensions secondaires comportent néanmoins certaines restrictions nécessaires au respect de la validité géométrique des objets dans l'hyper-espace (cf 3.3.6.2).

1.1.3 Facteurs de bloc

Comme nous l'avons vu lors de la discussion sur les problèmes de base du data-parallélisme, l'efficacité d'un programme peut être fortement influencée par la façon dont le compilateur projette les entités du programme sur les processeurs de la machine cible. Pour le langage C-HELP, nous avons choisi de laisser le programmeur libre dans le choix des tailles des blocs de points à allouer sur chaque processeur.

Pour une dimension primaire, il est possible de préciser un facteur de bloc qui représente une information issue de l'algorithme. Le programmeur spécifie un facteur de bloc pour une dimension afin de privilégier les communications à l'intérieur de ces blocs. En contre-partie, le parallélisme exprimé pour les éléments appartenant au même bloc ne pourra être effectivement obtenu sur la machine cible.

Ce facteur est une expression entière, constante, évaluable à la compilation ou le symbole étoile '*'. La valeur par défaut du bloc de communications pour une dimension est 1. L'utilisation de l'étoile permet de fixer la taille d'un bloc de communications à la taille de la dimension complète.

On dit que deux points font partie d'un même bloc quand, pour toute dimension de l'hyper-espace, la division entière de la coordonnée du point moins 1, par la taille du bloc donne le même résultat pour les deux points.

En adoptant les notations suivantes :

\mathcal{D}_{hs} l'ensemble des dimensions primaires de l'hyper-espace hs ;

$x_d(p)$ la coordonnée sur la dimension primaire d du point p ;

bc_d la taille du bloc de communication déclaré sur la dimension d ;

deux points pt_1 et pt_2 font partie du même bloc si et seulement si :

$$\forall d \in \mathcal{D}_{hs}, (x_d(pt_1) - 1) \text{ div } bc_d = (x_d(pt_2) - 1) \text{ div } bc_d$$

Le compilateur assure que toute communication effectuée entre deux points d'un même bloc s'opère de façon interne à la mémoire d'un processeur physique, en évitant l'utilisation du réseau de communications de la machine physique. On remarque alors qu'une dimension dont le bloc de communication est spécifié par l'étoile est totalement projetée en mémoire. Le parallélisme d'exécution est alors abandonné au profit de l'optimisation des communications le long de cette dimension. D'autres langages, en particulier HPF, adoptent ce principe qui permet d'informer le compilateur pour lui permettre de rendre plus efficace le code généré.

1.1.4 Arbre de priorité

Lors du développement d'un algorithme, le programmeur ne connaît pas *a priori* la topologie de la machine cible. Il ne peut donc qu'exprimer des informations qui vont diriger le compilateur vers une projection efficace. Le langage permet de donner un ordre de priorité entre les dimensions qui permettra au compilateur de privilégier le parallélisme, ou les temps de communications, pour chaque dimension de l'hyper-espace. Conformément au modèle géométrique HELP, cette notion d'ordre est attachée à l'hyper-espace, et non aux objets manipulés par le programme.

Le choix de l'arbre de priorité est un facteur important dans l'obtention de performances. Le programmeur doit porter une attention particulière à son utilisation, en fonction des traitements qu'il désire opérer. Par exemple, si son hyper-espace comporte trois dimensions et qu'un traitement important est séquentiellement opéré sur des objets alloués sur des plans parallèles, le parallélisme d'exécution ne sera maximum pour une machine à deux dimensions que si la projection privilégie le parallélisme sur les deux dimensions définissant ces plans, la dimension d'itération étant projetée en mémoire.

L'ordre de priorité est une expression parenthésée qui peut être traduite sous forme d'arbre appelé *arbre de priorité*. **▲** Toute feuille de l'arbre comporte le nom d'une dimension primaire de l'hyper-espace déclaré. L'expression de priorité sur les dimensions secondaires est interdite. Une dimension primaire ne doit apparaître qu'une seule fois au maximum dans l'arbre. Une dimension primaire qui n'apparaît pas dans l'arbre de priorité est considérée comme possédant une priorité infiniment basse.

Le niveau le plus prioritaire en terme de parallélisme est le premier niveau de l'arbre (profondeur 1). Lors de la compilation, le placement des blocs de l'hyper-espace sur les processeurs de la machine physique est calculé de façon à favoriser le parallélisme sur les dimensions primaires qui apparaissent à la plus petite profondeur dans cet arbre de priorité.

On dit qu'une dimension est projetée en parallèle quand il existe au moins deux blocs voisins sur cette dimension projetés sur des processeurs physiques différents. Le compilateur assure que lorsqu'une dimension primaire est projetée en parallèle, toutes les dimensions primaires plus prioritaires (dont le nom apparaît à un niveau supérieur de l'arbre) sont projetées en parallèle.

Quand plusieurs dimensions primaires ont le même niveau de priorité, le compilateur associe une dimension physique du réseau de la machine à chacune de ces dimensions de l'hyper-espace. Quand le nombre de dimensions de l'hyper-espace est plus important que le nombre de dimensions de la machine, un ordre de priorité a été choisi: les dimensions sont considérées dans l'ordre d'apparition sur le niveau de l'arbre, de gauche à droite. Dans le cas

où toutes les dimensions équi-prioritaires ne peuvent être projetées en parallèle, il se peut donc que certaines de ces dimensions soit projetées en parallèle alors que d'autres sont projetées en mémoire.

L'exemple de la figure III.4 montre la finesse de projection que l'on peut atteindre avec l'arbre de priorité. Sur une machine tableau contenant six processeurs (disposés en grille 3×2), on projette un hyper-espace à deux dimensions de taille 6×2. Deux couches sont nécessaires à la projection de tous les points de l'hyper-espace. Trois arbres de priorité sont successivement utilisés pour favoriser le parallélisme sur la première, la seconde dimension ou équitabement les deux dimensions (dans ce dernier cas, chaque dimension est associée à une dimension de la machine).

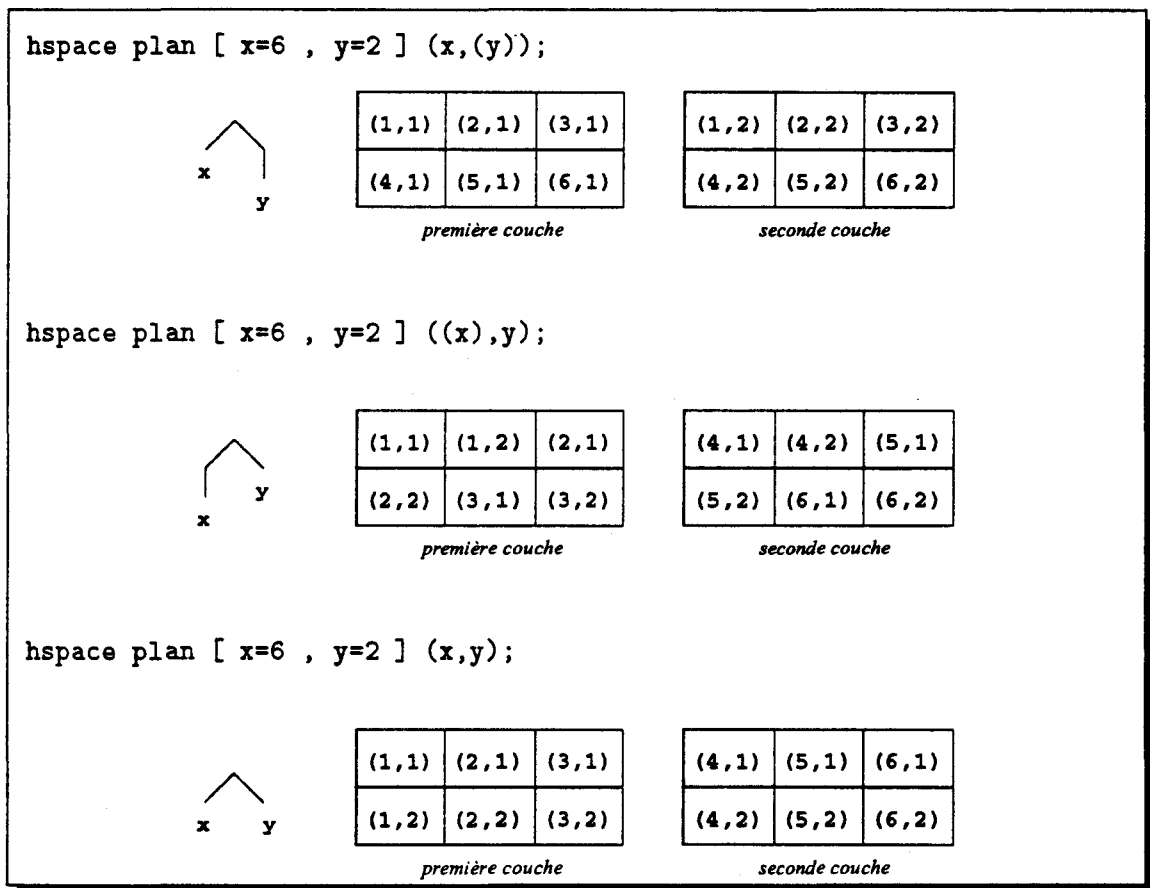


FIG. III.4 - Exemples d'utilisation des arbres de priorité


1.2 Les variables parallèles : DPO

La notion d'objet parallèle permet l'unification de toutes les variables parallèles d'un programme C-HELP, quels que soient leur taille, leur forme, leur position et le type de leurs éléments. Par l'association d'un hyper-espace d'allocation à tout DPO, le modèle géométrique d'interaction entre objets est possible pour des ensembles hétérogènes de DPO.


La déclaration d'un DPO dans un hyper-espace se fait par la donnée du type de ses éléments, de son hyper-espace d'allocation et de sa géométrie dans cet hyper-espace¹. L'hyper-espace doit, évidemment, être visible au moment de la déclaration du DPO.

Un DPO est alloué sur les points de l'hyper-espace qui font partie de son *domaine d'allocation*. Ce domaine est le cadre géométrique explicité par la donnée des intervalles sur chaque dimension de l'hyper-espace. Sur chaque point de ce polyèdre ainsi déclaré, un élément du type donné dans la déclaration est alloué.

1.2.1 Dynamicité

Un DPO est par défaut dynamiquement alloué lors de l'exécution. Son domaine d'allocation est susceptible de varier au cours de la vie de ce DPO par l'application d'une association (cf. 2.1).  Le programmeur a pour charge d'assurer que le domaine d'allocation reste inclus dans le domaine de coordonnées de l'hyper-espace. Il y a erreur de programmation lorsque le programmeur tente d'allouer dynamiquement un objet en dehors du cadre de l'hyper-espace.

Le langage C-HELP permet donc la dynamicité en position, en taille et en forme d'un objet parallèle, permettant au programmeur de faire évoluer ses données librement à travers le temps et l'hyper-espace. Dans tous les cas, l'association DPO/hyper-espace est impérative et figée. Pour que des DPO alloués dans des hyper-espaces différents interagissent, le programmeur est amené à utiliser les primitives de téléportation (cf. 4.4), à l'exclusion de tout autre système de communication (autre que le passage par le monde scalaire).

Lors de la déclaration d'un DPO, le mot-clef `steady` permet de rendre le domaine d'allocation du DPO constant.  L'utilisation de l'association sur ce DPO est par conséquent interdite (cf 2.1), et le domaine d'allocation doit impérativement être explicité lors de la déclaration (cf. 1.2.2).

1. Nous reviendrons plus loin dans ce chapitre sur les deux cas particuliers de déclarations de DPO :

- déclaration le long d'une dimension secondaire, voir 1.2.3 ;
- déclaration d'un DPO en paramètre formel d'une fonction, voir 4.3.

1.2.2 Domaine d'allocation

Lors de sa déclaration, l'intervalle d'allocation d'un DPO pour une dimension primaire est par défaut [1..1]. Il est possible d'exprimer le domaine d'allocation pour une ou plusieurs dimensions de l'hyper-espace. Pour cela, on exprime l'intervalle d'allocation sur chaque dimension par la donnée des bornes de chaque intervalle, ou par d'autres constructions syntaxiques équivalentes² [Laz94a]. Le programmeur peut aussi déclarer un DPO dont le domaine d'allocation est une copie de celui d'un DPO visible (mot-clef `idem`). Cette copie effectuée lors de l'exécution permet l'allocation dynamique de DPO à l'identique d'autres DPO, en particulier par rapport à un DPO passé en paramètre.

Quand le programmeur utilise le symbole '*' en guise d'intervalle d'allocation pour une dimension, le DPO est alloué sur toute la longueur de la dimension de l'hyper-espace.

Quand le domaine d'allocation d'un DPO n'est pas précisé (`[]`), le DPO n'est pas alloué lors de sa déclaration. En dehors de l'allocation, aucune opération n'est alors valide sur ce DPO. Son allocation est obtenue grâce à une association (cf. 2.1), suivant le modèle de dynamicité que nous venons de décrire.

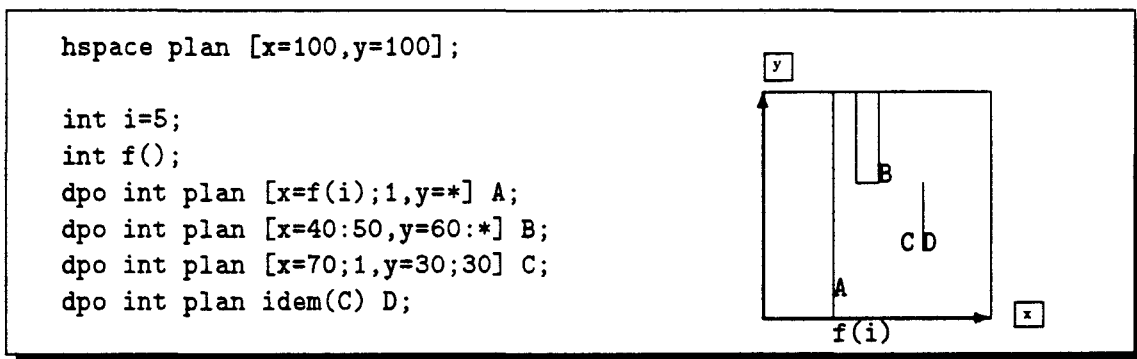


FIG. III.5 - Déclarations de DPO

L'allocation d'un DPO déclaré global est effectuée en entrée de la fonction `main()`, après l'initialisation des scalaires globaux; dans l'exemple de la figure III.5, l'affectation de la variable globale `i` est effectuée avant l'évaluation des expressions entières constituant la géométrie des objets, en particulier du DPO A.

2. Les intervalles s'expriment par le couple de bornes (:) ou par la borne inférieure et la longueur (;).

1.2.3 DPO alloués sur un axe secondaire

Les axes secondaires sont construits à partir d'une liste de noms de dimensions primaires. L'allocation et la déclaration d'un DPO le long d'une telle dimension se fait suivant une syntaxe équivalente à la déclaration sur un axe primaire, moyennant quelques restrictions dues à la volonté de garder des objets de formes raisonnables³. **⚠** Ainsi, si un DPO est alloué sur un axe secondaire, il ne peut comporter qu'une taille de 1 sur les dimensions primaires entrant dans la composition de cette diagonale.

Le programmeur peut manipuler des DPO alloués le long d'une diagonale mais d'origine quelconque. Ainsi, uniquement dans le cas d'une déclaration ne contenant pas le symbole '*', on peut en plus de la taille du DPO sur la dimension secondaire, préciser une origine du DPO pour chacune des dimensions primaires à partir desquelles est construite la dimension secondaire (cf. figure III.6). Il est possible de déclarer un plan diagonal à l'intérieur d'un hyper-espace comportant plus de deux dimensions, comme en figure III.7.

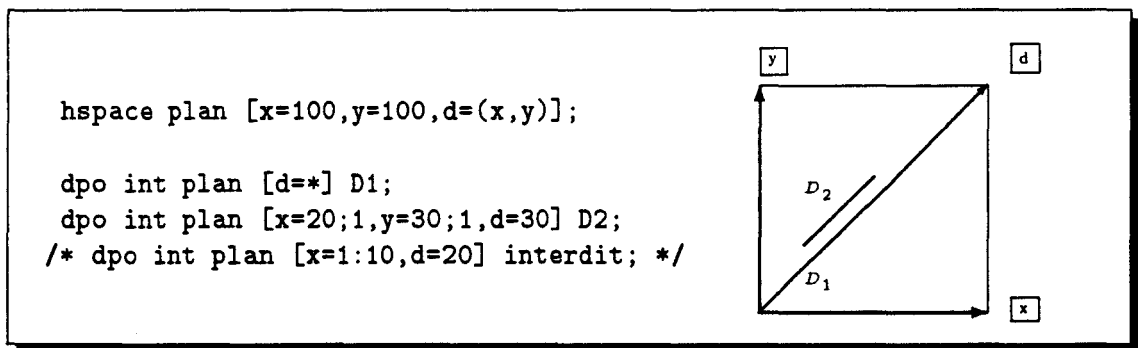


FIG. III.6 - Allocation sur une dimension secondaire

3. Nous avons décidé, pour la version actuelle du langage, de ne pas proposer des DPO « parallélogrammaux » ou triangulaires. Le modèle géométrique pourrait tout de même aboutir à de tels objets, et nous envisagerons ces possibilités dans le chapitre de conclusions et perspectives.

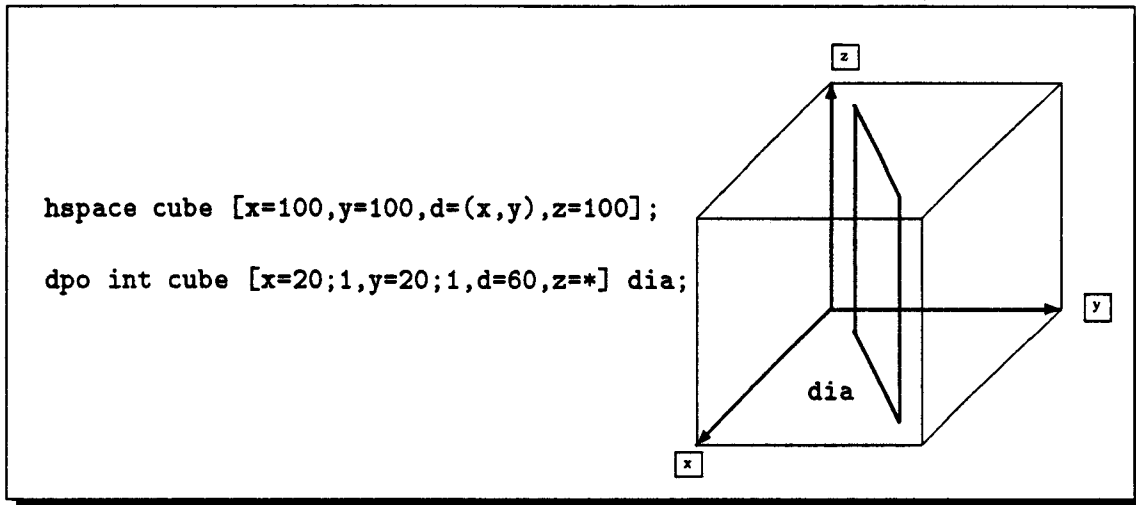


FIG. III.7 - Allocation d'un plan diagonal

1.2.4 Origine et longueur d'un DPO

On définit l'*origine* d'un DPO comme étant le point du domaine d'allocation du DPO le plus proche⁴ de l'origine de l'hyper-espace.

De même, on appelle *extrémité* d'un DPO, le point le plus éloigné de l'origine de l'hyper-espace.

La *longueur* d'un DPO est définie par rapport à une dimension et représente la largeur de l'intervalle d'allocation sur cette dimension.

1.2.5 Type d'un DPO

Dans tous les langages à parallélisme de données, les éléments d'un même objet sont tous du même type. Pour C-HELP, les types de base sont directement issus du langage C. On manipule ainsi des variables parallèles dont les éléments sont des caractères, des entiers ou des réels⁵.

4. au sens de la distance euclidienne : racine carrée de la somme des carrés des différences de coordonnées sur chaque dimension des deux points entre lesquels on calcule la distance.

5. Il existe aussi, pour le compilateur générant du code exécutable pour MasPar, le type de base **complex** du langage MPL[Mas91b]. Pour des raisons d'implémentation, les constructions de type structures ou unions ont été ignorées dans le développement des premières versions du compilateur C-HELP. L'apport de telles constructions ne perturberait pas les éléments spécifiques au modèle introduits dans le langage ; ces structures ne feraient que modifier l'allocation en mémoire des éléments sur les points de l'hyper-espace.

1.2.6 DPO de type void

Le langage C-HELP définit les DPO dont les éléments sont du type `void`. L'utilisation de tels DPO permet de manipuler des cadres géométriques pouvant être, en particulier, utilisés pour les constructions de domaine contraint (cf. 3.2.1). Cette caractéristique du langage propose ainsi au programmeur de séparer la gestion du domaine d'évaluation des expressions qui apparaissent dans l'écriture de son programme.

La particularité de ces objets est le fait qu'ils ne possèdent pas d'allocation mémoire effective sur les points de l'hyper-espace (ils sont par conséquent, absents de la mémoire parallèle de la machine physique). Dès lors, le compilateur ne génère que le code nécessaire aux manipulations de leur géométrie. Ces objets ont les mêmes caractéristiques de géométrie que tout autre DPO, en particulier, ils supportent la dynamique des objets (ils peuvent apparaître en partie gauche d'une association, cf. 2.1).

1.3 Variables scalaires

Une variable scalaire est une variable issue du langage C. Contrairement aux DPO, elle ne contient qu'une valeur. Toute variable scalaire est accessible en lecture à partir de n'importe quel point de n'importe quel hyper-espace, à la condition de la visibilité dans le code source de ces variables. Le programmeur est toutefois averti que les adresses scalaires n'ont aucune signification pour la mémoire des points de l'hyper-espace, du fait de la séparation des deux espaces d'adressage (cf. 1.4).

L'affectation d'une variable scalaire est opérée par les opérateurs d'affectation du C (`=`, `+=`, `--`, ...). Dans ce cas, si l'opérateur est diadique, la partie droite de l'expression doit donner un scalaire, en résultat de son évaluation. **⚠** L'affectation d'un scalaire par un DPO est interdite, même dans le cas d'un DPO qui n'est alloué que sur un point de l'hyper-espace. La primitive `scalar` permet la conversion d'un élément de DPO en scalaire (cf. 4.6.1).

1.4 Pointeurs et tableaux

Il existe deux espaces mémoires (et donc deux espaces d'adressage) différents dans le modèle d'exécution de C-HELP. Le premier espace est scalaire, il permet la gestion des variables scalaires (cf 1.3) de façon identique à leur usage classique en langage C. En particulier, les pointeurs de variables scalaires sont eux-mêmes des variables scalaires allouées dans l'espace mémoire scalaire et leur exploitation est régie par les fonctionnalités du C.

Le second espace mémoire est le cadre du parallélisme de données, l'hyper-espace. Un

point de cet hyper-espace est considéré, lors d'appels microscopiques, comme un processeur indépendant, possédant sa propre mémoire, partie de la mémoire globale de l'hyper-espace. Ainsi, C-HELP définit les DPO de type pointeurs comme des pointeurs internes aux points. Ces pointeurs sont donc des références à des adresses mémoire du point sur lequel ils sont définis (cf. figure III.8).

L'action de dépointer ou le passage à l'adresse sont donc des opérations microscopiques évaluées à l'intérieur du point actif, sans référence aux autres points (cf. 3.2).

La gestion de tableaux d'éléments suit ce modèle microscopique. L'adresse de base d'un tableau est valide uniquement au niveau de la mémoire du point.

Il n'existe pas, dans la première version du langage C-HELP, de pointeur scalaire d'objets parallèles explicitement manipulable par le programmeur. Nous montrerons lors de la discussion sur le passage de DPO en paramètre que le passage par adresse est tout de même possible. Les deux espaces mémoire sont ainsi conservés indépendants.

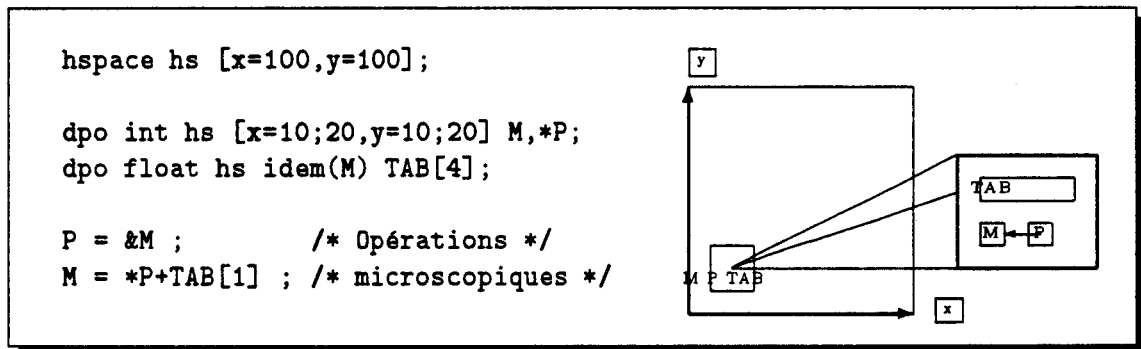


FIG. III.8 - Les pointeurs en C-HELP

2 Sémantique des opérations d'affectation C-HELP

LES AFFECTATIONS C-HELP ne sont pas des expressions mais des instructions. Δ Leur apparition au sein d'une expression est interdite, c'est une restriction introduite par rapport au langage C.

Nous avons défini deux types d'affectations : l'association qui introduit la dynamique des objets du langage, et les affectations injectives qui seront effectuées sans modification des domaines d'allocation.

2.1 Association

Une caractéristique intéressante pour un langage data-parallèle est de proposer au programmeur de manipuler des objets dont l'allocation (la taille, la forme ou la position) puisse changer au cours de l'exécution. Un opérateur est intégré pour donner une nouvelle allocation pour l'objet. C'est dans ce but que l'opérateur d'association est apparu dans le langage EVA [DMP90, Mar92].

En C-HELP, la dynamicité des objets permet leur évolution dans le temps selon des critères géométriques, conformément au modèle exposé au chapitre précédent. Pour affecter un nouveau domaine d'allocation à un DPO, le programmeur est amené à utiliser l'opérateur d'association '<-' qui associe un nom de DPO (partie gauche de l'affectation) au résultat de l'évaluation d'une expression produisant un DPO (partie droite de l'opérateur).

Le domaine d'allocation du DPO ainsi affecté devient égal au domaine d'allocation du DPO résultant de l'évaluation de l'expression de la partie droite. L'ancienne valeur du domaine d'allocation est perdue, et la mémoire correspondante est libérée. Naturellement, l'association '<-' est interdite sur un DPO déclaré avec le mot-clef **steady**.

Les valeurs issues de l'évaluation de la partie droite sont recopiées dans le nouveau DPO en partie gauche, localement à chaque point de l'hyper-espace sur lesquels une valeur a été calculée.

⚠ Pour garder l'intégrité du modèle, l'apparition d'une variable scalaire en partie gauche d'une association a été décrétée interdite. De même, l'association d'un DPO par une expression dont l'évaluation produit un scalaire est aussi interdite.

2.2 Affectations injectives

Les autres opérations d'affectation ne modifient pas le domaine d'allocation du DPO en cours d'affectation. Une copie du résultat de l'évaluation de l'expression en partie droite est effectuée sur chacun des points qui ont fait l'objet de cette évaluation. ⚠ Le DPO en partie gauche doit donc obligatoirement englober l'ensemble de ces points. On parle ainsi d'injections.

Dans cette catégorie d'opérateurs, on retrouve l'affectation '=' et les opérateurs d'incrémentations comme '+='.

2.2.1 Opérateur d'injection

L'opérateur d'affectation '=' est dit *opérateur d'injection*. Le DPO en partie gauche doit englober le résultat de l'évaluation de l'expression en partie droite (cf. figure III.9).

En cas d'expression scalaire en partie droite, l'affectation est réalisée par défaut sur le domaine d'allocation du DPO présent en partie gauche.

Le domaine d'allocation du DPO en partie gauche n'est jamais modifié par l'opérateur '='.

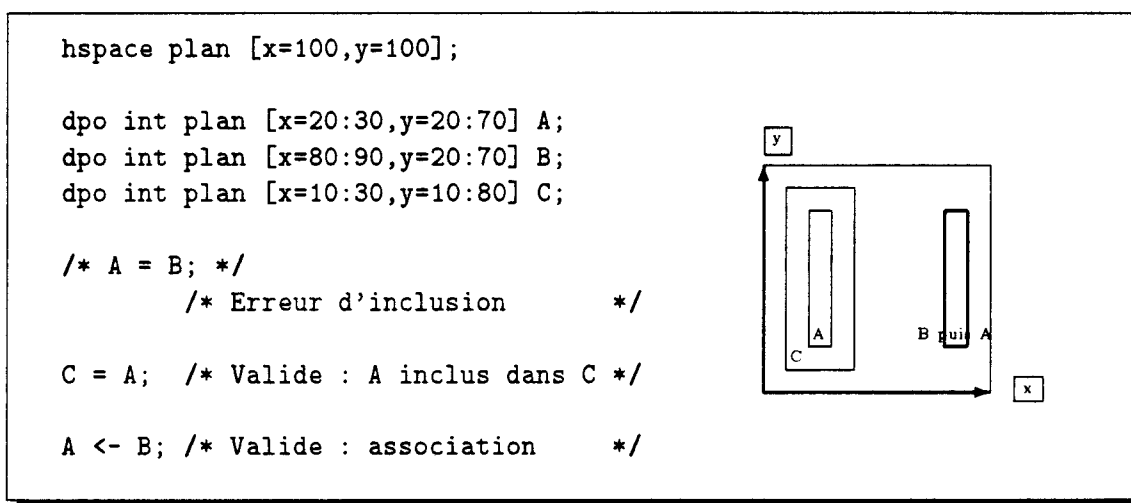


FIG. III.9 - Les deux types d'affectations

2.2.2 Opérateurs d'incrémentement

Les « opérateurs d'incrémentement » (comme '+=') issus du langage C sont évalués sur le même principe que l'affectation injective : le DPO en partie gauche doit englober les points sur lesquels l'évaluation de l'expression en partie droite a été effectuée.

Il est important de remarquer que ces opérateurs ne possèdent donc pas la même sémantique que leurs « extensions naturelles » ; par exemple, $A+=B$ n'est pas équivalent à $A=A+B$ (cf. figure III.10).

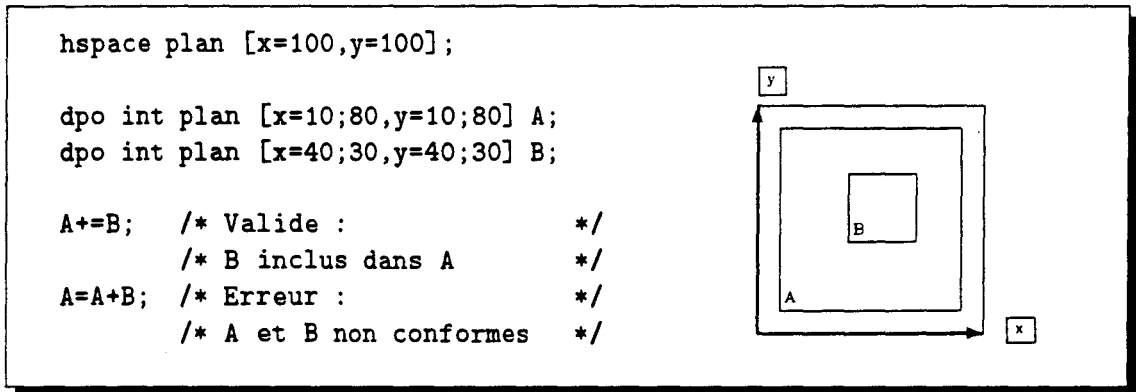


FIG. III.10 - Opérateurs d'incrémentation

3 Expressions data-parallèles C-HELP

COMME dans tout langage data-parallèle, c'est au niveau des expressions C-HELP que le parallélisme est exprimé. Une expression C-HELP fait interagir des DPO et des scalaires localement aux points actifs de l'hyper-espace. Tous les opérateurs du langage C sont étendus, à l'exception des opérateurs d'affectation, qui ne font pas partie des expressions C-HELP.

3.1 Découpage hiérarchique

Le modèle HELP met en valeur l'importance de la séparation des phases de calculs et des phases de communications. De plus, le programmeur a la possibilité de déterminer un cadre géométrique pour l'évaluation de tout ou partie de ses expressions. Pour intégrer ces concepts dans le langage, nous proposons la hiérarchisation des expressions.

⚠ Un *segment conforme* est une séquence d'opérations data-parallèles calculatoires qui vérifie la règle de conformité:

La règle de conformité est respectée quand tous les DPO (variables ou temporaires) qui interviennent dans le segment recouvrent un même ensemble de points. Cet ensemble est nommé *domaine de conformité* du segment, il est déterminé par l'une des deux possibilités:

- le programmeur l'explícite par le constructeur de domaine contraint
- il prend une valeur par défaut, sinon.

Une expression C-HELP est définie comme un arbre de segments conformes. Cet arbre est

construit suivant les règles (cf. figure III.11) :

- une expression est un arbre de segments. On appelle *segment conforme principal* d'une expression le segment conforme qui apparaît au niveau haut de l'arbre de segments conformes formant l'expression ;
- l'appel de primitives de communication crée un segment conforme de niveau inférieur ;
- un appel de fonction crée un segment conforme pour chacun de ses paramètres, à l'exclusion des fonctions intrinsèques et microscopiques (cf. 4.2) ;
- un domaine de conformité est associé à tout segment conforme.

La figure III.11 présente un exemple du découpage d'une expression en segment conformes. Le lecteur peut se reporter à la suite de ce chapitre pour le détail des primitives du langage. Les premiers segments conformes sont limités par les sous-expressions $a+b$, $c+d$ et x car ils font chacun l'objet d'une application d'un appel macroscopique. Un autre segment conforme est créé par l'appel de fonction $f()$ sur le segment conforme composé de l'addition des deux résultats précédents. Enfin, le segment conforme principal est défini sur la multiplication des deux termes de l'expression. \triangle Pour chacun des segments conformes, le règle de conformité doit être vérifiée, indépendamment des autres segments conformes.

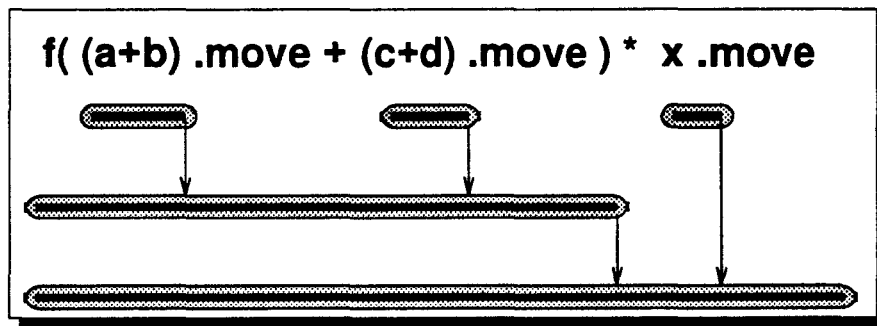


FIG. III.11 - Le découpage hiérarchique des expressions

3.2 Opérations microscopiques

Un segment conforme est exclusivement composé d'opérateurs microscopiques. La vue microscopique permet au programmeur d'expliciter les opérations calculatoires entre les DPO.

Une expression faisant intervenir des variables scalaires et des DPO est syntaxiquement construite de la même façon en C-HELP qu'en ANSI-C, à l'exclusion des opérateurs d'affec-

tation. En particulier, l'ordre de priorité entre les opérateurs logiques et arithmétiques est conservé.

Lorsqu'elle fait intervenir des DPO, une expression est appelée « microscopique » et le compilateur génère le code pour son évaluation en chaque point du domaine de conformité.

3.2.1 Domaine de conformité explicite ou contraint

On appelle *domaine de conformité* le domaine (ensemble compact de points de l'hyper-espace) sur lequel une expression microscopique est évaluée.

Pour préciser le domaine de points sur lequel un segment conforme est évalué, le programmeur spécifie le cadre géométrique de cette évaluation à l'aide du constructeur `on`.

3.2.1.1 Le constructeur `on` Le constructeur `on` a pour effet de contraindre le domaine de conformité (et par conséquent le domaine d'activité, cf. 3.2.4) au domaine d'allocation du DPO argument, pour le segment conforme dans lequel il apparaît.

Avec un domaine contraint, les DPO intervenant dans l'évaluation du segment conforme doivent uniquement englober le domaine. On peut ainsi déclencher des opérations microscopiques sur des DPO de domaines d'allocation différents en restreignant le domaine de conformité à tout ou partie de l'intersection géométrique des domaines d'allocation.

Ce constructeur peut être utilisé avec les DPO de type `void` pour fournir un moyen explicite de placer le domaine de conformité sur un ensemble précis de points, conformément au modèle géométrique. La manipulation explicite du domaine de conformité à l'aide de ces DPO rend le programme plus lisible, comme nous le présenterons dans le chapitre dédié aux exemples d'applications C-HELP.

Le programmeur a la possibilité d'utiliser le constructeur `on` sur une instruction d'affectation. Dans ce cas, le domaine contraint ne s'applique que sur le segment principal de l'expression en partie droite de l'opérateur d'affectation (cf. figure III.12).

```

hspace plan [x=100,y=100];

dpo int plan [x=10;80,y=10;80] A;
dpo void plan [x=40;30,y=40;30] B;
dpo int plan [x=30;50,y=20;80] C;

on(B)  A=C+1;

/* équivalent à */
A= (on(B) C+1);
    
```

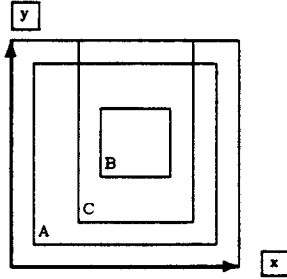


FIG. III.12 - *Domaine contraint*

Le constructeur on peut encore être étendu à un bloc d'instructions. Pour chaque instruction du bloc, le segment conforme principal (de la partie droite quand l'instruction est une affectation) est alors contraint au domaine de conformité explicité. (cf. figure III.13).

<pre> on (DPO) { _____ segments _____ conformes _____ principaux } </pre>	<p><i>Équivalents</i></p>	<pre> TMP <- DPO ; on (TMP) _____ on (TMP) _____ on (TMP) _____ </pre>
---	---------------------------	---

FIG. III.13 - *Constructeur on sur un bloc*

```

hspace plan [x=100,y=100];

dpo int plan [x=10;80,y=10;80] A;
dpo int plan [x=40;30,y=40;30] B;
dpo int plan [x=30;50,y=20;60] C;

on(C) {
  C+=A;
  on (B) /* B inclus dans C */
    C=B+A;
}

/* équivalent à */
/* (distribution sur le bloc) */
on(C) C+=A;
on(C) on(B) C=B+A;

/* équivalent à */
/* (imbrication de deux on) */
on(C) C+=A;
on(B) C=B+A;

/* équivalent à */
/* (passage à l'expression) */
C += on(C) A;
C = on(B) B+A;

```

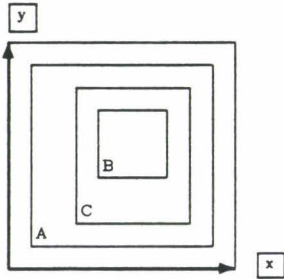


FIG. III.14 - *Imbrication de domaines contraints*

3.2.1.2 Restrictions successives Plusieurs constructeurs `on` peuvent être imbriqués afin de réduire successivement le domaine de conformité à des domaines de plus en plus petits, au sens de l'inclusion. Pour cela, l'argument de ce constructeur doit aussi être inclus dans l'argument du `on` englobant.

Quand un segment conforme comporte plusieurs restrictions de domaine de conformité, les domaines d'allocation des DPO arguments des `on` doivent donc être imbriqués les uns dans les autres (de droite à gauche). Le domaine contraint résultant est égal au domaine d'allocation du DPO argument du dernier `on` (le plus à droite). (cf. figure III.14).

3.2.2 Domaine de conformité par défaut

L'explicitation du domaine de conformité pour un segment conforme peut être omise dans le langage. Dans ce cas, tous les DPO qui apparaissent dans le segment doivent être alloués sur les mêmes points de l'hyper-espace. Ils sont fortement conformes. Le domaine de conformité est alors égal au domaine d'allocation de ces DPO.

3.2.3 Règle de conformité

La règle de conformité, déjà présentée lors de la décomposition hiérarchique, peut maintenant s'exprimer en prenant en compte les deux types de conformité. Voir figure III.15.

Tous les DPO visibles dans un segment conforme vérifient la règle de conformité si :

- ils englobent le domaine de conformité du segment, si celui-ci est explicité par un constructeur `on` ;
- ils sont alloués sur les mêmes points d'un même hyper-espace, sinon ^a.

^a on parle alors de conformité forte

FIG. III.15 - Règle de conformité

Le programmeur assure que la règle de conformité est vérifiée pour faire interagir des DPO. Aucun comportement n'est garanti pour le déclenchement de l'évaluation d'un segment conforme sans vérification de la règle de conformité.

Il existe une option du compilateur `-RTCC`⁶ pour générer, dans une phase de développement d'un code C-HELP, le code nécessaire à la vérification, durant l'exécution, de la conformité des DPO qui interagissent. Cette vérification ne peut être systématique qu'au moment de l'exécution, à cause de la dynamique des objets. Une vérification à la compilation pourrait néanmoins être effectuée sur les parties de code dans lesquelles on connaît les domaines d'allocation des objets.

3.2.4 Domaine d'activité

L'évaluation d'un segment conforme peut être soumise à l'évaluation d'une expression conditionnelle qui inhibera, en cas de résultat nul, une partie des points du domaine de

6. *Run-Time Conformity Check*

conformité associé à cette expression. Cette caractéristique que l'on retrouve dans tous les langages data-parallèles permet de contrôler finement l'activité en fonction du contexte.

On définit le domaine d'activité comme un sous-ensemble du domaine de conformité.

3.2.4.1 Constructeur where Le constructeur `where(DPO_expr)` s'applique sur un segment conforme. Le segment conforme principal de l'expression argument du `where` doit vérifier la conformité du segment dans lequel le constructeur apparaît.

Dans le cas général, un segment conforme est donc composé d'un constructeur `on` de définition du domaine de conformité, puis d'un constructeur `where` définissant le domaine d'activité comme sous-ensemble du domaine de conformité. Les DPO apparaissant dans le segment conforme doivent englober le domaine de conformité (cf. figure III.16).

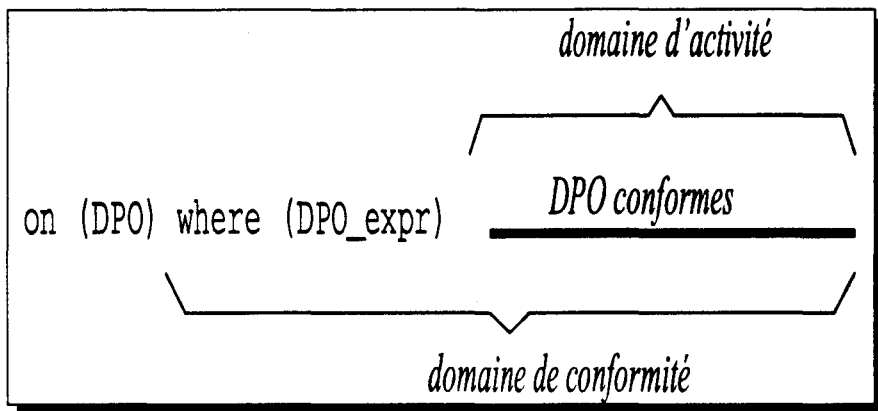


FIG. III.16 - Un segment conforme complètement spécifié

Lors de l'évaluation d'un segment conforme, le compilateur évalue, sur les points du domaine de conformité, l'expression du constructeur `where`. Le domaine d'activité est l'ensemble des points du domaine de conformité pour lesquels le résultat est vrai.

L'évaluation du segment se fait sur le domaine d'activité. Les points de l'hyper-espace qui ne font pas partie du domaine d'activité sont inhibés pour cette évaluation.

Le constructeur `where` n'a aucune influence sur les segments conformes inférieurs, dont le résultat intervient dans le segment conforme soumis au constructeur `where` (cf. figure III.17). La portée d'un masque est ainsi limitée au segment conforme dans lequel il apparaît.

Suivant le même principe que pour la restriction du domaine de conformité par le constructeur `on`, le constructeur `where` peut être appliqué à une instruction d'affectation. Dans ce cas,

le masque ne s'applique que sur le segment conforme principal de l'expression en partie droite de l'affectation. Les affectations se font alors sur les points du domaine d'activité de cette évaluation du segment principal de l'expression.

Là encore, le constructeur **where** peut être factorisé pour un bloc d'instructions. Dans ce cas, il s'applique sur chacune des instructions du bloc. (cf. figure III.17).

Quand plusieurs masques sont opérés sur un même segment conforme, seuls les points du domaine de conformité du segment, satisfaisant à toutes les expressions booléennes, font partie du domaine d'activité.

```

where ( A != 0 ) { /* A domaine de conformité forte */
  B = C + D;
  /* A, C et D fortement conformes */

  B = ( E + F ).transrel(x,1);
  /* transrel est une opération macroscopique, son */
  /* appel crée un nouveau segment. (cf 3.3.4) */

  /* E et F non nécessairement conformes à A */
  /* mais fortement conformes entre eux. */
  /* résultat de (E+F).transrel(x,1) conforme à A */
}

```

FIG. III.17 - Masquage et conformité

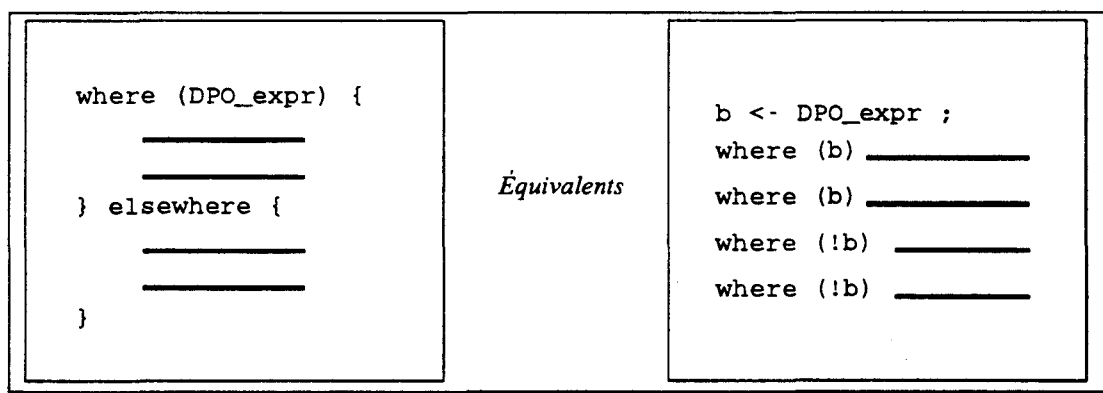


FIG. III.18 - Sémantique du elsewhere

Avec l'utilisation du **where** sur un bloc d'instructions, le programmeur a la possibilité

d'utiliser un bloc **elsewhere**. Pour ce bloc, les instructions sont masquées par l'inverse de la condition du **where** (cf. figure III.18).

Quand un segment conforme est entièrement masqué (qu'il ne reste aucun point actif), il n'est pas évalué. Pour les instructions scalaires qui sont éventuellement intégrées dans un segment conforme entièrement masqué, l'évaluation n'est pas déclenchée par le compilateur. L'adoption de cette convention est influencée par la sémantique définie pour le langage intermédiaire choisi (cf. chapitre 4).

3.2.4.2 Opérateur de fusion L'opérateur conditionnel du langage C est étendu à la notion de DPO pour donner le constructeur de fusion qui s'applique suivant la syntaxe :

```
dpo_expr_bool ? dpo_expr_if : dpo_expr_else
```

Cet opérateur est microscopique. C'est-à-dire que les trois expressions apparaissant dans ce constructeur constituent un segment conforme. La règle de conformité s'applique donc indépendamment sur chacun d'eux (ils peuvent utiliser des résultats d'évaluation de segments conformes inférieurs), mais les trois résultats doivent vérifier la conformité entre eux pour l'évaluation de l'opérateur de fusion.

Le DPO résultant de l'évaluation de cet opérateur est conforme au domaine de conformité. Pour chaque point actif (satisfaisant les conditions des **where/elsewhere** englobants) sur lequel le résultat de l'expression conditionnelle donne *vrai*, l'élément résultant est le résultat de l'expression **dpo_expr_if**; pour les autres points, le résultat de l'évaluation de **dpo_expr_else**.

L'évaluation d'un opérateur de fusion est sémantiquement équivalent à l'évaluation sur tout point actif du domaine de conformité de l'opérateur conditionnel du langage C construit avec les valeurs résultantes de l'évaluation des expressions DPO sur le point (ces expressions étant évaluées dans l'ordre : **dpo_expr_bool**, **dpo_expr_if** puis **dpo_expr_else**). L'exclusion des opérateurs d'affectation dans les expressions permet d'éviter les effets de bord et donc d'obtenir cette sémantique (cf. 2).

3.2.5 Ordre de priorité pour évaluation microscopique

La priorité des opérateurs du langage C est conservée pour l'évaluation d'un segment microscopique. Les constructeurs **where** et **on** ont une priorité inférieure aux opérateurs classiques.

Par exemple, pour le segment conforme **on(A) B+C**, l'addition est déclenchée pour le

domaine contraint A, les DPO B et C doivent englober le domaine d'allocation de A. Par contre, lors de l'évaluation du segment conforme $(on(A) B) + C$ l'addition est déclenchée sur deux DPO fortement conformes, C doit être alloué sur le même domaine d'allocation que A.

L'opérateur d'application d'une primitive de migration (cf. 3.3.2) a une priorité supérieure à tous les opérateurs. La figure III.19 montre un exemple de découpage hiérarchique issu de l'adoption de cet ordre de priorité.

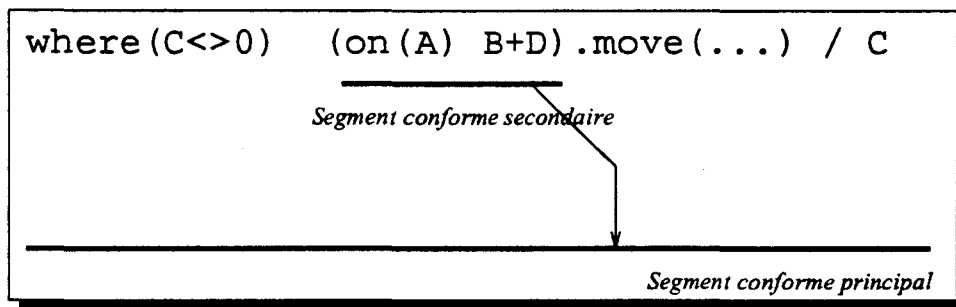


FIG. III.19 - Exemple de découpage hiérarchique

3.3 Opérations macroscopiques

Conformément au modèle HELP exposé dans le chapitre précédent, les opérateurs macroscopiques modélisent les communications sous forme de déplacements géométriques d'objets à l'intérieur d'un même hyper-espace.

Un appel à une suite de primitives macroscopiques crée un nouveau segment conforme pour une expression C-HELP. La règle de conformité s'applique donc sur l'expression qui va faire l'objet de la migration, indépendamment de la conformité du segment d'utilisation du résultat de cette migration.

Le langage C-HELP propose comme opérations macroscopiques une série de primitives qui peuvent s'appliquer sur des DPO, variables parallèles du programmes ou résultats d'évaluation d'un segment conforme. Pour une migration, ce DPO est appelé DPO *source*. Toutes ces primitives sont utilisées relativement à l'hyper-espace, pour garder le même référentiel tout au long du développement de l'algorithme.

Le résultat produit par une opération macroscopique est un DPO temporaire, qui doit entrer en conformité avec le segment supérieur dans lequel il est consommé (cf. figure III.20).

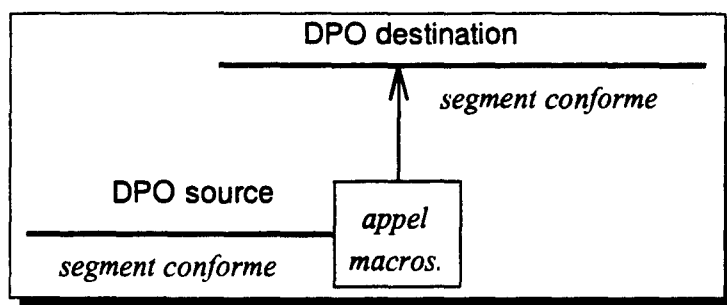


FIG. III.20 - Appel macroscopique et conformité

3.3.1 Fonctions d'association source/cibles

Pour donner aux primitives macroscopiques du langage une sémantique clairement établie, chacune de ces primitives M est caractérisée par la fonction f_M qui à partir des coordonnées (x_1, x_2, \dots, x_n) d'un point P appartenant au domaine source de l'hyper-espace donne les coordonnées $(x'_1, x'_2, \dots, x'_n)$ des points P' appartenant à l'ensemble des points cibles.

$$\forall P = (x_1, \dots, x_n) \in Dom_{source} \quad M(P) = \{P' = (x'_1, \dots, x'_n) \mid (x'_1, \dots, x'_n) \in f_M((x_1, \dots, x_n))\}$$

Cet ensemble peut comporter un nombre variable d'éléments pour un point source donné. On classe les différentes primitives macroscopiques du langage suivant la cardinalité de ces ensembles :

Primitives bijectives L'ensemble cible est un singleton pour tout point source; et tous les ensembles cibles sont disjoints deux à deux ;

Primitives sélectives Certains des ensembles de destination sont vides. Les autres sont des singletons, la fonction d'association pour ces points est l'identité ;

Primitives répliquatives Tous les ensembles cibles comportent la même cardinalité, supérieure ou égal à 1; et ils sont tous disjoints.

Réducteurs associatifs Les ensembles cibles sont des singletons pour chaque point source; et ne sont pas disjoints. Un opérateur associatif est déclenché à chaque réception sur des points cibles.

Les autres configurations possibles de ces ensembles ne sont pas représentées par des primitives du langage C-HELP. Il existe toutefois une primitive non-géométrique qui ne peut être caractérisée dans cette classification (primitive **scatter** cf. 3.3.5.5).

L'évaluation de la primitive macroscopique effectue une copie de la valeur du DPO sur un point source vers l'ensemble des points destinations (sauf dans le cas des primitives de réduction où un opérateur est appliqué au moment de la réception, cf. 3.3.5.4). Il y a ainsi création d'un DPO temporaire sur le nouveau domaine d'allocation.

3.3.2 Application d'opérateurs macroscopiques

L'application d'un opérateur macroscopique, ou d'une succession d'opérateurs macroscopiques, se fait par l'intermédiaire du constructeur point ('.'). Le programmeur a la possibilité de contrôler les communications sur chaque point du domaine source par l'expression d'un test évalué avant le déclenchement de la communication. Cette expression apparaît entre parenthèses, on l'appelle le *domaine de contrôle macroscopique*. La forme générale d'appels macroscopiques est donc :

```
dpo_expr . (dcm) macro_1() . . . . macro_n() : dpo_else
```

où `dpo_expr` et `dcm` font partie du même segment conforme d'une part et le résultat de la migration (DPO temporaire produit) et `dpo_else` font partie du segment conforme englobant d'autre part.

3.3.3 Sémantique de l'appel macroscopique

Durant un appel de primitives macroscopiques, l'expression source est évaluée indépendamment du contexte englobant : l'appel macroscopique crée un segment conforme de niveau inférieur dans l'arbre de l'expression. Le compilateur génère ensuite le code nécessaire à l'évaluation du domaine de contrôle macroscopique `dcm`. Ce domaine peut éventuellement être lui aussi issu de l'évaluation d'une expression macroscopique, mais doit toujours être conforme avec le segment principal de `dpo_expr` produisant le DPO source.


Le DPO source fait ensuite l'objet de l'application de la suite d'opérateurs macroscopiques. Chaque point du domaine de conformité source qui possède une valeur non nulle pour le domaine de contrôle macroscopique va émettre la valeur de l'expression source à destination d'un ou plusieurs points, suivant les primitives appelées (cf. 3.3.5). L'éventuel domaine d'activité du segment principal de `dpo_expr` n'est pas considéré pour l'appel macroscopique. Seul le domaine de contrôle macroscopique détermine quels sont les points du domaine de conformité qui vont effectivement envoyer une valeur. Lorsqu'il existe des points sur lesquels l'évaluation de l'expression source `dpo_expr` n'a pas été opérée (des points faisant partie du

domaine de conformité mais n'appartenant pas au domaine d'activité), le programmeur assure que ces points ne font pas partie du domaine de contrôle macroscopique `dcm`. Dans le cas contraire, une valeur non-déterminée est transmise.

Le langage offre tout de même la possibilité pour le programmeur de considérer le domaine d'activité de l'évaluation du segment conforme principal de l'expression source comme domaine de contrôle macroscopique par l'omission de l'expression `dcm` (tout en gardant les parenthèses vides). Dans ce cas, seuls les points actifs lors de l'évaluation du DPO source sont émetteurs. Un tel appel se fait suivant la syntaxe :

```
dpo_expr . ( ) macro_1() . . . . macro_n() : dpo_else
```

Quand plusieurs primitives macroscopiques sont successivement appliquées, les coordonnées des éléments de l'ensemble de points cibles sont calculées par la composition des fonctions d'association.

Un DPO temporaire résultant est ainsi produit. Son domaine d'allocation est calculé suivant les primitives utilisées, il est conforme au DPO qui aurait été produit par les mêmes appels aux primitives macroscopiques si le domaine de contrôle macroscopique avait été complet (vrai en tout point du domaine d'allocation du DPO source). Les éléments du DPO résultant qui n'appartiennent pas à l'ensemble des points cibles se trouvent affectés par le résultat de l'évaluation de l'expression par défaut.  Le programmeur doit assurer que le DPO résultant de l'évaluation de l'expression par défaut respecte le domaine de conformité du segment qui l'utilise. L'expression par défaut peut aussi être scalaire. La possibilité ainsi proposée au programmeur de contrôler l'application d'une suite d'appels macroscopiques permet de limiter les nombres de données qui transitent par le réseau virtuel de l'hyper-espace, et par conséquent, sur le réseau de données de la machine cible.

L'explicitation du domaine de contrôle macroscopique est optionnelle. Lors de l'absence de cette information, le domaine de contrôle macroscopique est considéré comme complet. Il n'y a, dans ce cas, pas d'expression par défaut. L'appel se fait donc dans ce cas suivant la syntaxe :

```
dpo_expr . macro_1() . . . . macro_n()
```

3.3.4 Composition des appels macroscopiques

Il est utile de remarquer que la composition des appels macroscopiques donne à l'opérateur `.'` deux rôles sémantiques différents, suivant leur placement dans l'expression. Ainsi, la

première occurrence de ‘.’ s’applique sur un DPO en partie gauche et peut se voir appliquer la construction de contrôle macroscopique. Les occurrences suivantes modélisent la composition des appels macroscopiques, \triangleleft elles ne peuvent pas comporter d’expressions de contrôle macroscopique.

Dès lors, le programmeur doit remarquer que l’expression `(dpo.macro_1).macro_2` n’est pas équivalente à `dpo.macro_1.macro_2`. Dans le souci de garder une syntaxe simple, le langage n’utilise néanmoins qu’un opérateur unique pour ces deux sémantiques.

3.3.5 Primitives macroscopiques

Certaines opérations macroscopiques ont un usage limité pour les DPO alloués sur des dimensions secondaires, ou pour des appels faisant intervenir de telles dimensions. Dans ce paragraphe, nous ne considérons que le cas de DPO alloués sur des dimensions primaires et des appels aux primitives avec des paramètres dimensions primaires. Le paragraphe 3.3.6.2 fixera les limitations dues aux dimensions secondaires.

Notations On note :

$x_{i_{Orig}}$: coordonnée de l’origine du DPO sur la i^e dimension de l’hyper-espace.

$x_{i_{Extrem}}$: coordonnée de l’extrémité du DPO sur la i^e dimension de l’hyper-espace.

$lg_{i_{DPO}}$: longueur de l’intervalle d’allocation du DPO sur la i^e dimension de l’hyper-espace.

3.3.5.1 Primitives bijectives Les primitives bijectives associent à chaque point source un ensemble de points cibles réduit à un singleton. Tous les ensembles de destination sont disjoints et l’union de ces ensembles donne un domaine compact de points, qui constituera le domaine d’allocation du DPO résultant.

3.3.5.1.1 Translations Ces opérations effectuent une migration d’un DPO à l’intérieur de l’hyper-espace d’allocation. La forme du DPO destination et son orientation par rapport aux axes sont identiques à celles du DPO source.

transabs(d, off) translation le long d’une dimension **d** dont le nom est passé en premier paramètre. L’origine du DPO destination est fixée à **off** sur la dimension **d**. Ainsi, la longueur de la translation est calculée à partir à la coordonnée sur **d** de l’origine du DPO source, et elle est égale à la différence entre le second paramètre **off** (dont l’évaluation lors de l’appel donne une valeur entière comprise dans l’intervalle de définition de **d**) et la coordonnée d’origine (cf. figure III.21).

$$f_{\text{transabs}(d,\text{off})}((x_1, \dots, x_d, \dots, x_n)) = \{(x_1, \dots, \text{off} + (x_d - x_{d_{\text{Orig}}}), \dots, x_n)\}$$

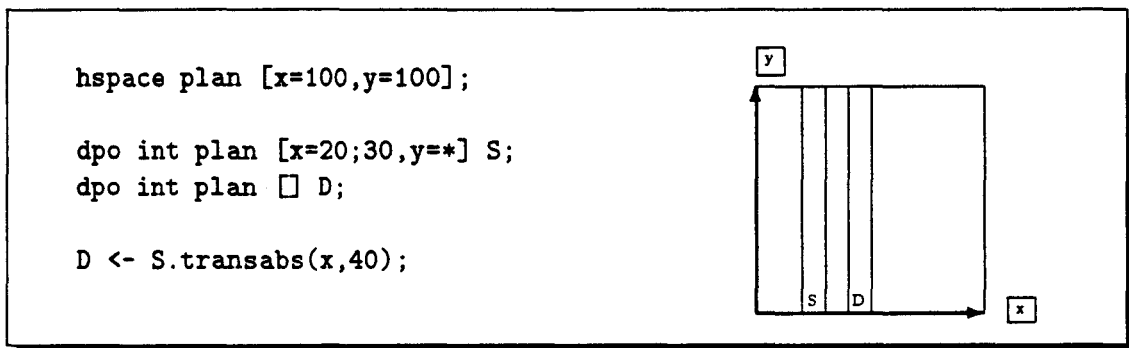


FIG. III.21 - Translation vers une position absolue

transrel(d, off) translation le long d'une dimension *d* dont le nom est passé en paramètre. La longueur de la translation est donnée par le second paramètre (expression entière) (cf. figure III.22).

$$f_{\text{transrel}(d,\text{off})}((x_1, \dots, x_d, \dots, x_n)) = \{(x_1, \dots, x_d + \text{off}, \dots, x_n)\}$$

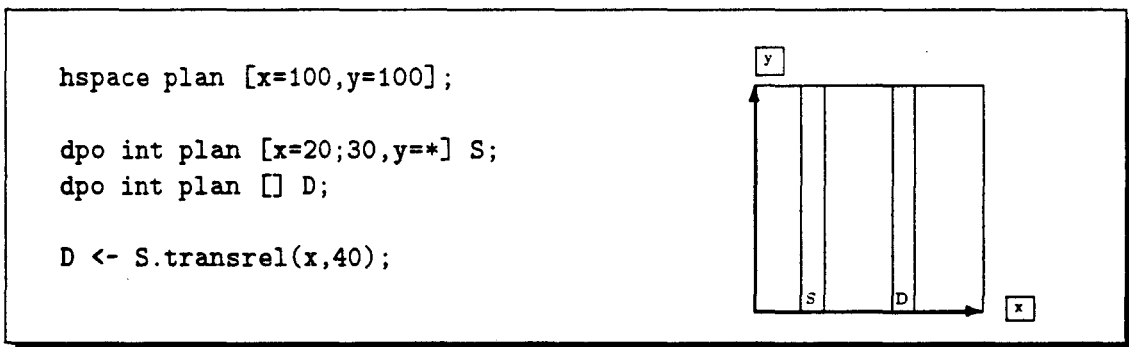


FIG. III.22 - Translation avec déplacement relatif

On peut remarquer que **transrel(d, off)** est équivalent à **transabs(d, $x_{d_{\text{Orig}}} + \text{off}$)**.

moveabs(o₁, ..., o_n) translation vers une position absolue de l'origine dans l'hyper-espace. Les paramètres représentent les nouvelles coordonnées de l'origine dans l'hyper-espace, dans l'ordre de déclaration des dimensions de l'hyper-espace (cf. figure III.23).

$$f_{\text{moveabs}(o_1, \dots, o_n)}((x_1, \dots, x_n)) = \{(o_1 + (x_1 - x_{1_{\text{Orig}}}), \dots, o_n + (x_n - x_{n_{\text{Orig}}}))\}$$

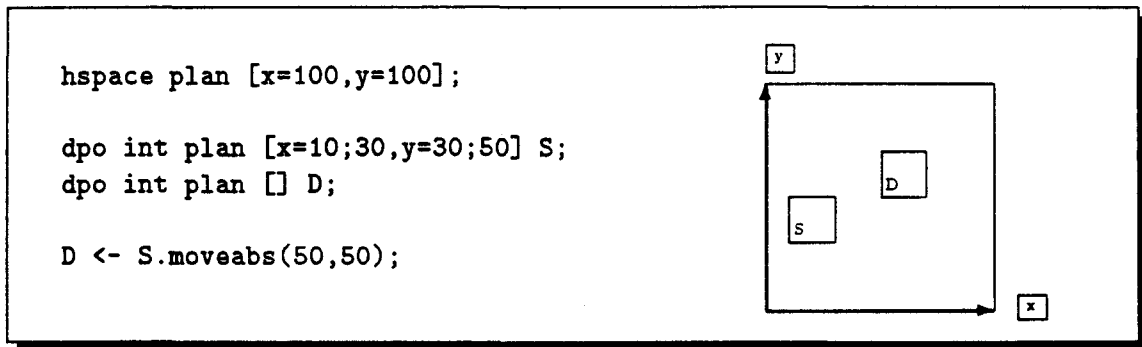


FIG. III.23 - *Changement absolu d'origine*

On peut remarquer que `moveabs(o1, ..., on)` est équivalent à `transabs(d1, o1).transabs(dn, on)`.

`moverel(o1, ..., on)` translation relative de l'origine du DPO source dans l'hyper-espace. Les paramètres représentent les différences des coordonnées de l'origine et de la destination dans l'hyper-espace, dans l'ordre de déclaration des dimensions primaires de l'hyper-espace (cf. figure III.24).

$$f_{\text{moverel}(o_1, \dots, o_n)}((x_1, \dots, x_n)) = \{(x_1 + o_1, \dots, x_n + o_n)\}$$

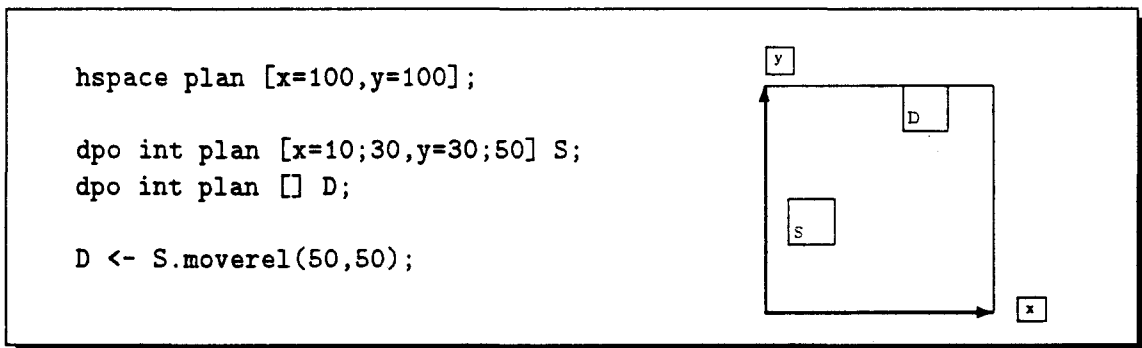


FIG. III.24 - *Changement relatif d'origine*

On peut remarquer que `moverel(o1, ..., on)` est équivalent à `moveabs(o1 + x1Orig, ..., on + xnOrig)`.

3.3.5.1.2 Échanges de dimensions Ces primitives produisent un DPO dont l'orientation et, éventuellement, la position de l'origine sont différentes de celles du DPO source.

`exchabs(d1, d2)` échange des coordonnées absolues dans l'hyper-espace d_1 et d_2 (cf. figure III.25).

$$f_{\text{exchabs}(d_1, d_2)}((x_1, \dots, x_{d_1}, \dots, x_{d_2}, \dots, x_n)) = \{(x_1, \dots, x_{d_2}, \dots, x_{d_1}, \dots, x_n)\}$$

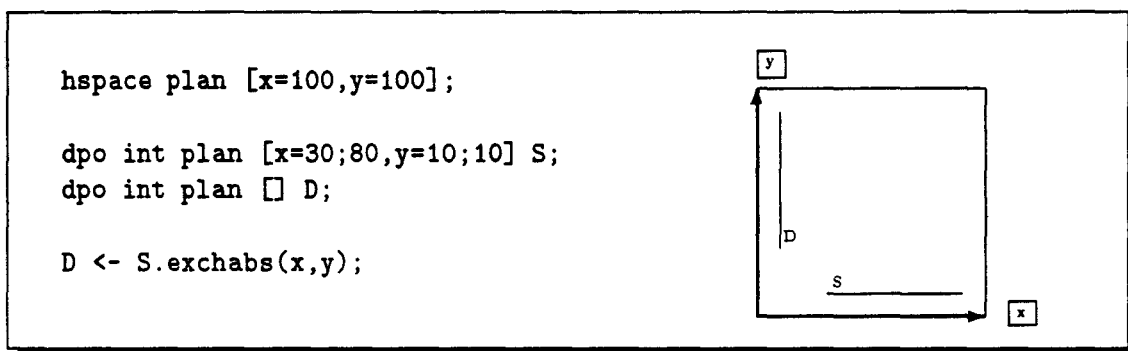


FIG. III.25 - Échange absolu d'orientation

`exchrel(d1, d2)` échange des coordonnées dans l'hyper-espace d_1 et d_2 , relativement à l'origine de l'objet (cf. figure III.26).

$$f_{\text{exchrel}(d_1, d_2)}((x_1, \dots, x_{d_1}, \dots, x_{d_2}, \dots, x_n)) = \{(x_1, \dots, x_{d_2} - x_{d_2_{\text{orig}}} + x_{d_1_{\text{orig}}}, \dots, x_{d_1} - x_{d_1_{\text{orig}}} + x_{d_2_{\text{orig}}}, \dots, x_n)\}$$

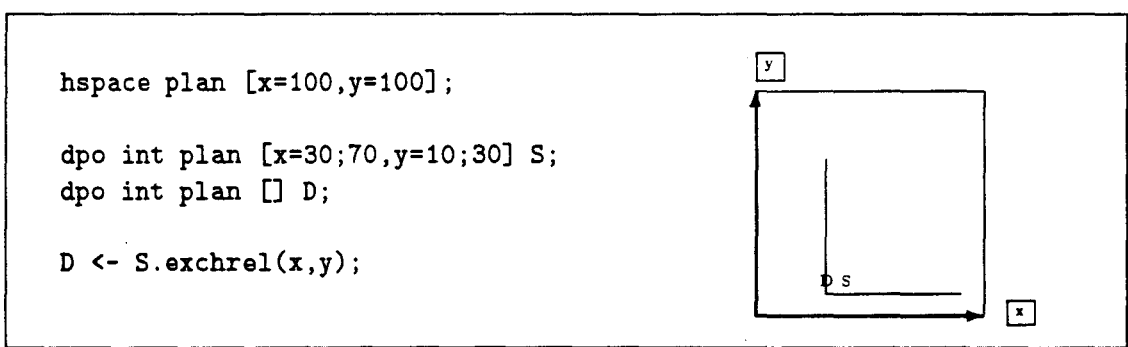


FIG. III.26 - Échange relatif d'orientation

3.3.5.1.3 Autres primitives géométriques bijectives Il existe d'autres primitives bijectives, parfois utiles pour l'expression simple de certains algorithmes.

`shifttor(d, off)` Cette primitive opère un décalage torique le long de la dimension d de longueur `off`, vers les coordonnées croissantes si `off` est positif; vers les coordonnées

décroissantes si *off* est négatif (cf. figure III.27). La coordonnée du point cible sur la dimension *d* est calculée par une opération de modulo avec la taille du domaine d'allocation sur cette dimension du DPO source.

$$f_{\text{shifttor}(d,\text{off})}((x_1, \dots, x_d, \dots, x_n)) = \{(x_1, \dots, x_{d_{\text{orig}}} + (x_d + \text{off}) \bmod l_{g_{d_{DPO}}}, \dots, x_n)\}$$

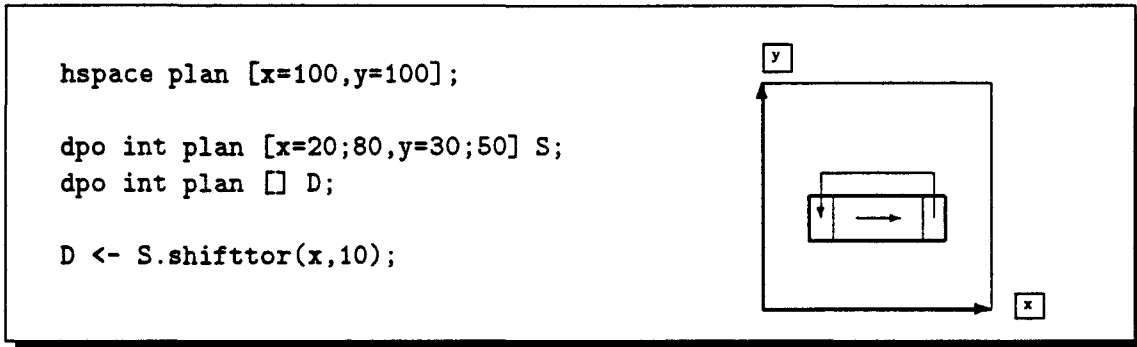


FIG. III.27 - Décalage torique

flip(d) Cette primitive effectue une opération de miroir du DPO source sur son intervalle d'allocation le long de la dimension *d* passée en argument. La coordonnée sur *d* du point destination pour un point source du DPO est donc calculée en fonction des bornes de l'intervalle d'allocation du DPO source sur la dimension concernée (cf. figure III.28).

$$f_{\text{flip}(d)}((x_1, \dots, x_d, \dots, x_n)) = \{(x_1, \dots, x_{d_{\text{orig}}} + x_{d_{\text{extrem}}} - (x_d), \dots, x_n)\}$$

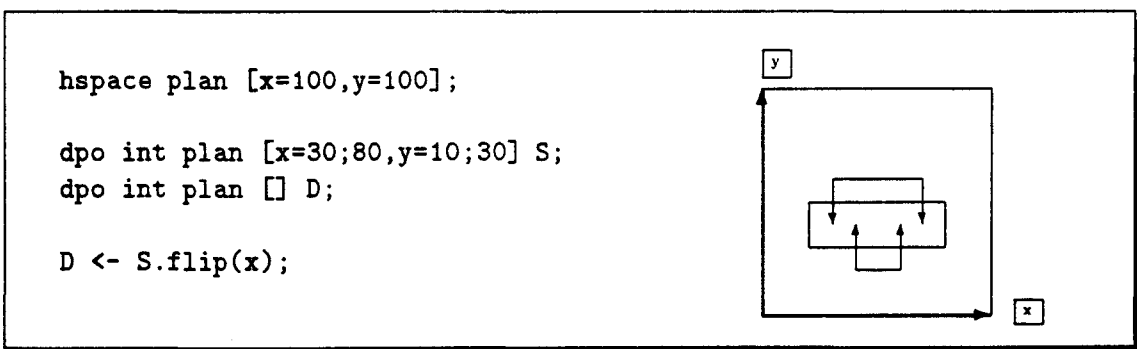


FIG. III.28 - flip

3.3.5.2 Primitives sélectives Les primitives sélectives permettent d'extraire un sous-objet géométrique d'un DPO. Chaque ensemble associé à un point source est soit vide, soit réduit au singleton qui contient le même point que la source (la fonction est l'identité pour ces

points). \triangle Le programmeur doit assurer que les paramètres d'appels sont compatibles avec le DPO source. Une extraction d'un domaine qui ne fait pas partie du domaine d'allocation source est une erreur de programmation.

extrabs(d,off) Extraction d'un sous-DPO de taille 1 sur la dimension concernée, à une coordonnée absolue *off* (cf. figure III.29).

$$f_{\text{extrabs}(d,\text{off})}((x_1, \dots, x_n)) = \begin{cases} \{(x_1, \dots, x_n)\} & \text{si } x_d = \text{off} \\ \emptyset & \text{sinon} \end{cases}$$

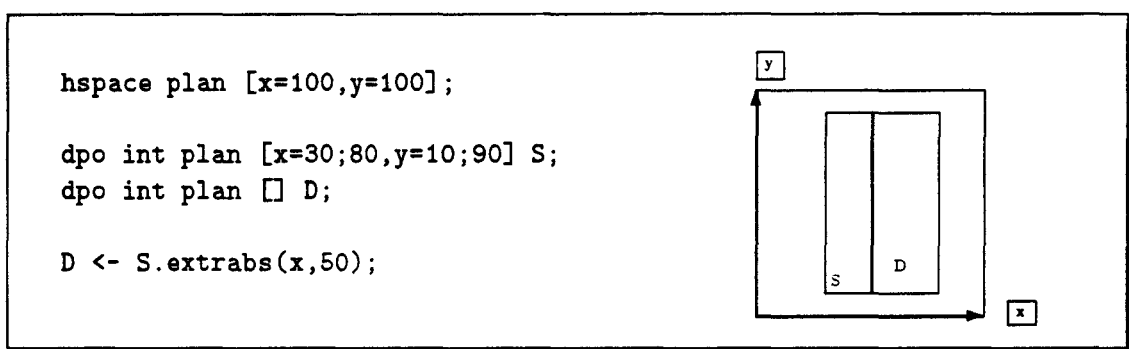


FIG. III.29 - *Extraction d'une coordonnée absolue*

extrabs(d,low..up) Extraction d'un sous-DPO sur un intervalle de coordonnées absolues [low..up] (cf. figure III.30).

$$f_{\text{extrabs}(d,\text{low}..\text{up})}((x_1, \dots, x_n)) = \begin{cases} \{(x_1, \dots, x_n)\} & \text{si } \text{low} \leq x_d \leq \text{up} \\ \emptyset & \text{sinon} \end{cases}$$

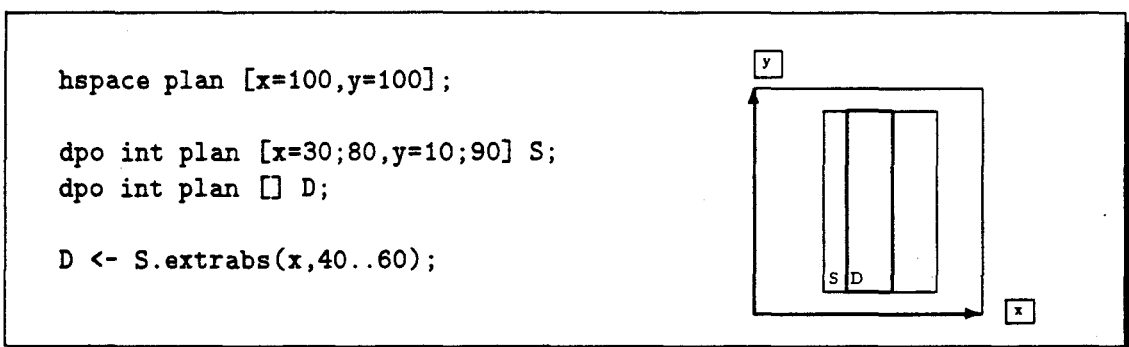


FIG. III.30 - *Extraction d'un intervalle de coordonnées absolues*

On peut remarquer que **extract(d,off..off)** est équivalent à **extract(d,off)**.

extrrel(d,off) Extraction d'un sous-DPO de coordonnée **off** sur la dimension concernée, relativement à l'origine de l'objet source (cf. figure III.31).

$$f_{\text{extrrel}(d,\text{off})}((x_1, \dots, x_n)) = \begin{cases} \{(x_1, \dots, x_n)\} & \text{si } x_d - x_{d_{\text{orig}}} + 1 = \text{off} \\ \emptyset & \text{sinon} \end{cases}$$

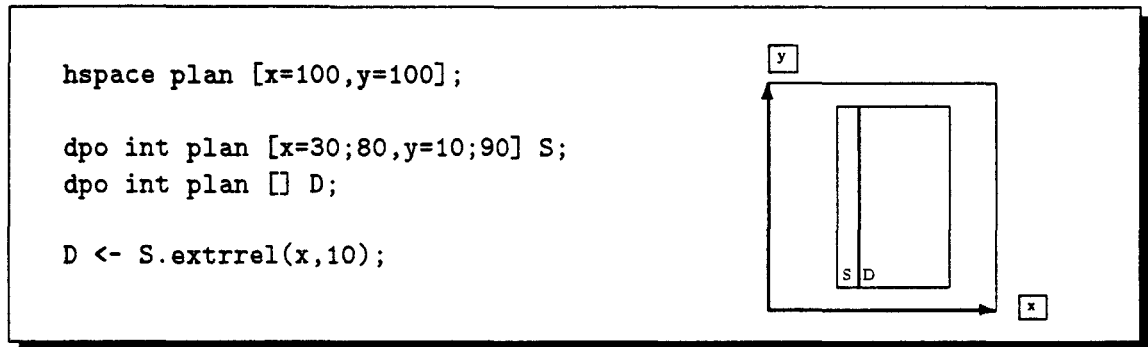


FIG. III.31 - *Extraction d'une coordonnée relative*

extrrel(d,low..up) Extraction d'un sous-DPO sur un intervalle de coordonnées sur **d**, par rapport à l'origine de l'objet (cf. figure III.32).

$$f_{\text{extrrel}(d,\text{low}..\text{up})}((x_1, \dots, x_n)) = \begin{cases} \{(x_1, \dots, x_n)\} & \text{si } \text{low} \leq x_d - x_{d_{\text{orig}}} + 1 \leq \text{up} \\ \emptyset & \text{sinon} \end{cases}$$

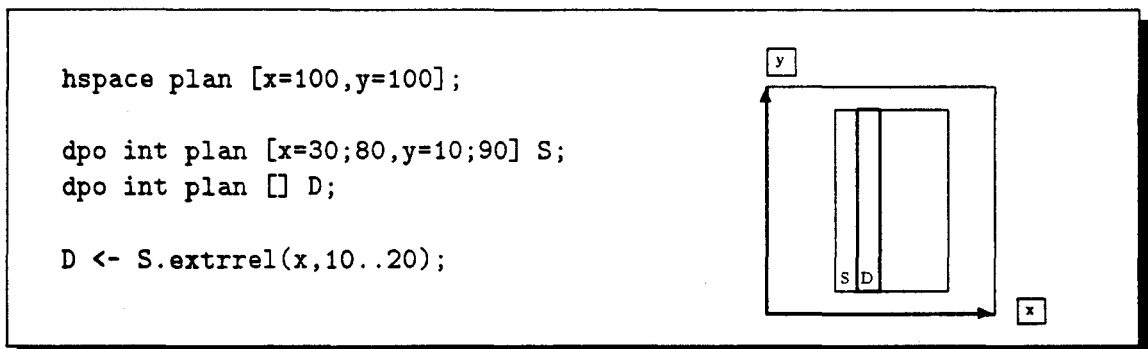


FIG. III.32 - *Extraction d'un intervalle de coordonnées relatives*

3.3.5.3 Primitives répliquatives Les fonctions d'association définies pour les primitives répliquatives donnent, pour tout point source, un ensemble de cardinalité supérieure ou égale

à 1. Tous ces ensembles sont disjoints et leur union définit un pavé, domaine d'allocation du DPO résultant.

expand(d) Réplique le DPO source sur toute la dimension d. Si le DPO source possède une taille de 1 sur cette dimension, alors le DPO produit est alloué sur la dimension complète (cf. figure III.33).

Dans le cas contraire, le nombre de réplifications est choisi maximal, dans le sens des coordonnées négatives et positives, en conservant les mêmes valeurs sur les points sources. Le DPO destination est alors formé de juxtapositions successives de copies du DPO source.

$$f_{\text{expand}(d)}((x_1, \dots, x_d, \dots, x_n)) = \{(x_1, \dots, x_d + k * lg_{d_{DPO}}, \dots, x_n)\}$$

$$k \in]-1 * \lfloor x_{d_{Orig}} / lg_{d_{DPO}} \rfloor, \lfloor (lg_{d_{HS}} - x_{d_{Extrem}}) / lg_{d_{DPO}} \rfloor]$$

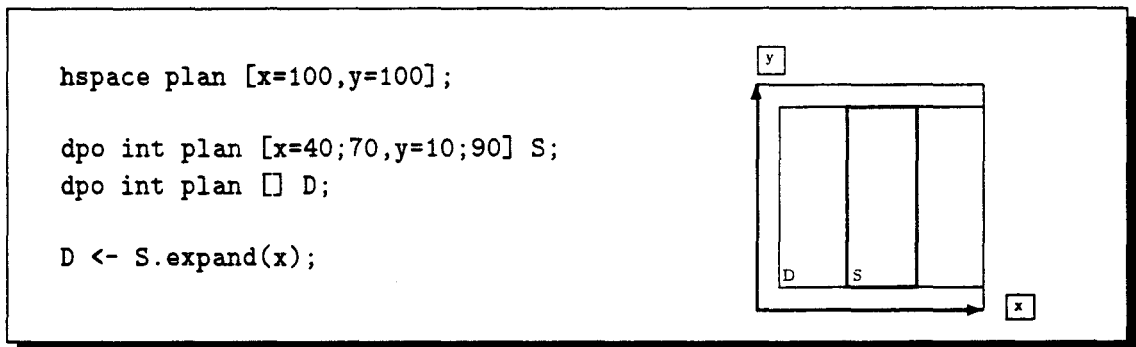


FIG. III.33 - Réplifications sur une dimension complète

expand(d,t) Réplique t fois le DPO source, vers les coordonnées négatives si t est négatif, vers les coordonnées positives si t est positif. Quand t est nul, le DPO résultat est une copie du DPO source (cf. figure III.34).

$$f_{\text{expand}(d,t)}((x_1, \dots, x_d, \dots, x_n)) = \left\{ \begin{array}{l} (x_1, \dots, x_d + k * lg_{d_{DPO}}, \dots, x_n), \\ k \in [t, 0] \text{ si } t < 0 \\ k \in [0, t] \text{ si } t \geq 0 \end{array} \right\}$$

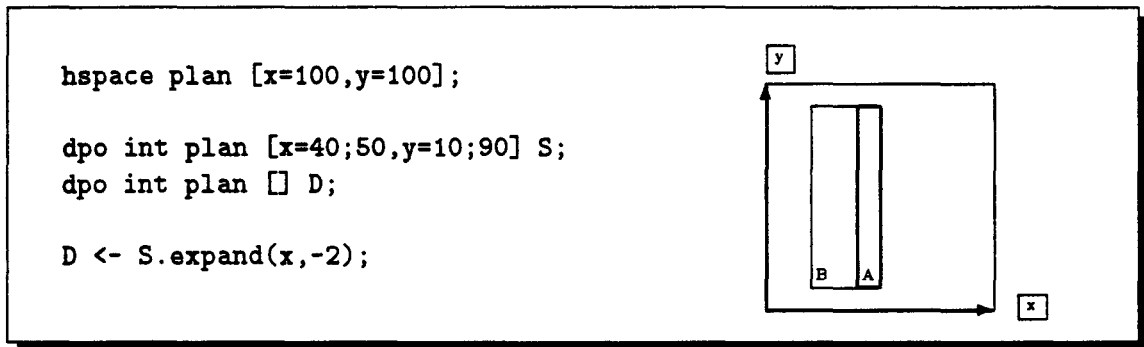


FIG. III.34 - Réplifications n fois vers les coordonnées décroissantes

stretch(d,t) Étend le DPO source sur la dimension passée en paramètre. Le DPO destination possède une largeur t fois supérieure à celle du DPO source. Les t points de coordonnées les plus petites du DPO destination reçoivent la valeur du premier point du DPO source, les t suivants reçoivent la deuxième valeur...

La valeur de t est toujours strictement positive. L'extension est faite vers les coordonnées croissantes (cf. figure III.35). \triangle Le programmeur a pour charge de s'assurer que la réplification ne dépasse pas l'intervalle de coordonnées de l'hyper-espace⁷.

$$f_{\text{stretch}(d,t)}((x_1, \dots, x_d, \dots, x_n)) = \left\{ \begin{array}{l} (x_1, \dots, x_{d_{\text{Orig}}} + k + \text{lg}_{DPO} * (x_d - x_{d_{\text{Orig}}}), \dots, x_n), \\ 0 \leq k < t \end{array} \right\}$$

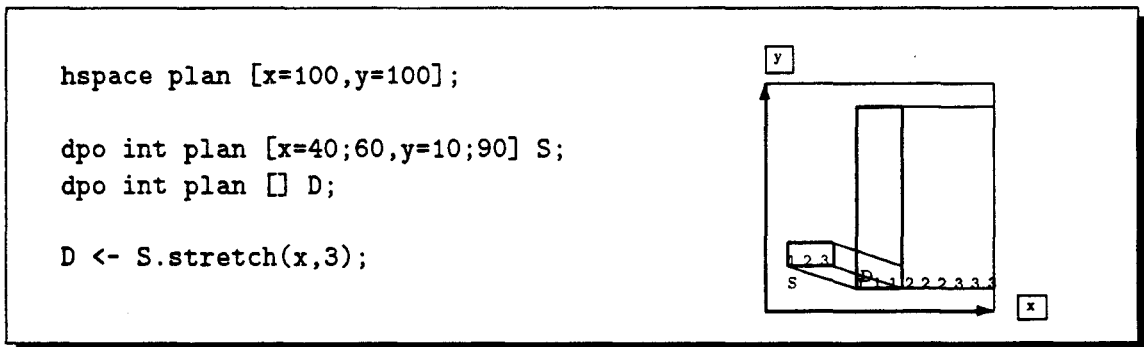


FIG. III.35 - Réplication par étirement

L'appel à **stretch** sur un DPO source de largeur 1 sur la dimension concernée est équivalent à la réplification par **expand**.

3.3.5.4 Réducteurs associatifs Les réducteurs associatifs permettent d'opérer des réductions le long d'une dimension. La réduction donne un DPO de largeur 1 sur la dimension concernée, de coordonnée égale à celle de l'origine du DPO source sur cette même dimension.

7. Cette vérification pourrait être générée par le compilateur.

La description de ces opérations de réduction suit le modèle précédent (cf. infra). À chaque point source, correspond un point cible. La particularité de ces opérations est que chaque point cible reçoit (en général) plusieurs valeurs. L'opérateur associé à la réduction permet d'obtenir la valeur résultante par application de cet opérateur sur l'ensemble des valeurs reçues. L'ordre des réceptions (et donc des calculs) n'est pas spécifié.

`reduceadd(d)`

`reducemul(d)`

`reduceor(d)`

`reduceand(d)`

`reducemax(d)`

`reducemin(d)` opèrent la somme, la multiplication, le ou logique, le et logique, le calcul de la valeur maximale ou de la valeur minimale des valeurs des points sources.

$$f_{\text{reduce...}(d)}((x_1, \dots, x_d, \dots, x_n)) = \{(x_1, \dots, x_{d_{orig}}, \dots, x_n)\}$$

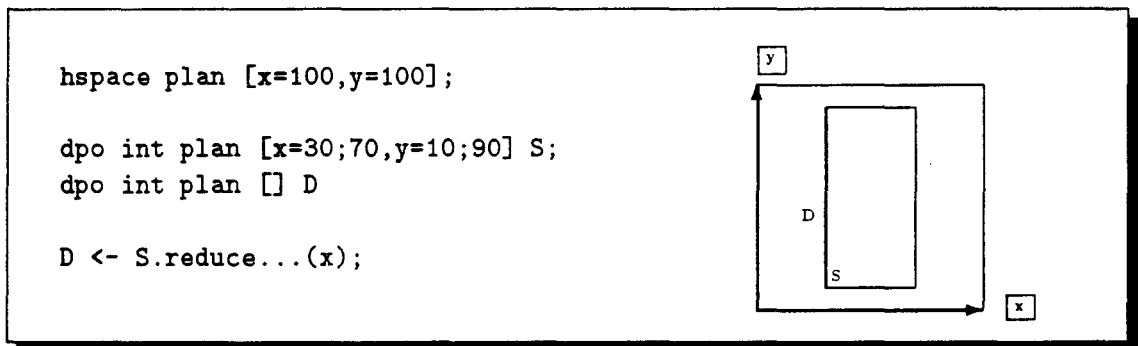


FIG. III.36 - Réductions associatives

3.3.5.5 Primitive macroscopique non géométrique Pour certaines opérations particulières, il est nécessaire de contrôler un réarrangement du placement des données par des valeurs parallèles précédemment calculées. La primitive que nous allons présenter ne peut pas entrer dans le modèle géométrique du fait de la totale irrégularité potentielle des communications qu'elle peut décrire.

`scatter(DPO1, ..., DPON, DPOframe)` opère un réarrangement du DPO source suivant les valeurs des N premiers DPO passés en argument, à destination d'un DPO temporaire dont la

géométrie est une copie de la géométrie d'un DPO passé en dernier paramètre. N est le nombre de dimensions primaires de l'hyper-espace.

Chacun des N premiers arguments représente la coordonnée du point cible sur la dimension correspondante. Il est donc du type entier non signé et doit englober le DPO source. Pour un point source, l'ensemble des points destination est limité au singleton formé du point de coordonnées égales aux valeurs des DPO arguments sur ce même point.

$$f_{\text{scatter}}(c_1, \dots, c_n, \text{frame})((x_1, \dots, x_n)) = \{(c_1|_{(x_1, \dots, x_n)}, \dots, c_n|_{(x_1, \dots, x_n)})\}$$

⚠ Le programmeur doit assurer que les coordonnées de tout point cible réfèrent un point faisant partie de la géométrie du dernier argument. Un point cible qui ne fait pas partie de ce domaine entraîne un comportement indéfini du programme.

L'utilisation de cette primitive doit être soumise à la précaution du programmeur : lorsque plusieurs points sources ont le même point cible, la valeur résultante sur ce point est une des valeurs reçues, sélectionnée aléatoirement lors de l'exécution. Il n'y a pas de vérification de cette condition.

3.3.6 Validité des opérations macroscopiques

3.3.6.1 Dépassements d'hyper-espace **⚠** Les opérateurs macroscopiques sont définis comme des fonctions qui associent à chaque point du DPO source un ensemble de points destinations. Il n'y a pas de contrôle de validité des coordonnées des points destinations. Le programmeur a la charge d'assurer que l'application de ces primitives n'entraîne pas de dépassement de l'espace de validité géométrique de l'hyper-espace pour l'ensemble des points cibles d'un appel macroscopique. Néanmoins, quand une suite de plusieurs appels macroscopiques est appliquée, il peut y avoir des coordonnées intermédiaires (entre deux primitives macroscopiques) non valides pour l'hyper-espace. La condition de validité est uniquement requise pour les coordonnées des points de destinations finales.

Une option du compilateur permet de générer le code qui vérifie dynamiquement le non-dépassement du cadre de l'hyper-espace.

3.3.6.2 Primitives macroscopiques et dimensions secondaires Un DPO peut être alloué sur une dimension secondaire. Il ne possède alors qu'une indication de taille sur cette dimension et qu'une coordonnée pour l'origine sur les dimensions qui entrent dans la composition de la dimension secondaire.

Dans tous les cas, un DPO doit préserver à tout instant de l'exécution cette condition de validité : il ne peut en aucun cas comporter une largeur supérieure à 1 dans une dimension

primaire et dans une dimension secondaire construite sur cette dimension primaire. C'est sur cette condition que se limitent les appels macroscopiques faisant intervenir des dimensions secondaires.

Certaines restrictions sont par conséquent introduites dans les conditions d'appels des primitives macroscopiques dans le cas de dimensions secondaires. Le programmeur a pour charge de s'assurer que ses paramètres d'appels et la géométrie du DPO source sont compatibles. Le DPO temporaire résultant de la suite totale des appels aux primitives macroscopiques possède dans ce cas la même limitation pour son allocation sur les dimensions secondaires.

La sémantique de l'application de primitives macroscopiques sur les DPO alloués sur une dimension secondaire ne change pas par rapport au cas général. En particulier, pour tout point source, les coordonnées du (des) point(s) cible(s) sont calculées par les mêmes fonctions d'association source/cible.

Dans l'exemple de la figure III.37, l'appel à la primitive de réplication entraîne la définition d'un domaine d'allocation du DPO résultant non valide. Il y a erreur de programmation⁸.

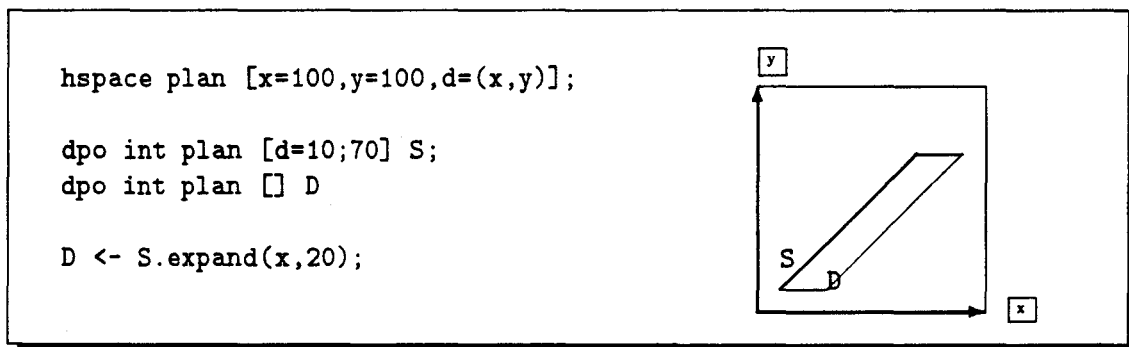


FIG. III.37 - Erreur de programmation : DPO non valide

4 Fonctions et Bibliothèques

ON PEUT distinguer plusieurs types de fonctions, suivant leur positionnement vis-à-vis du modèle géométrique et de ses conséquences.

fonctions intrinsèques Cette première famille de fonctions permet l'accès à la géométrie des objets, pour faciliter le développement de fonctions lorsque la géométrie des objets ne peut être connue à l'écriture du programme (par exemple, lors d'un passage de DPO en paramètre).

fonctions microscopiques Le programmeur peut aussi écrire ses propres fonctions mi-

⁸. Le compilateur pourrait en générer la vérification dynamique.

croscopiques suivant la vue microscopique induite par la séparation communications/calculs. Ces fonctions s'exécuteront localement aux points actifs du domaine de conformité, offrant au programmeur la possibilité d'entrer dans une partie de code asynchrone.

fonctions générales Nous nous pencherons ensuite sur les problèmes de passage de DPO en paramètre lors de l'écriture de fonctions faisant intervenir un ou plusieurs hyper-espaces. Ces fonctions étendent la notion de fonction classique du C au modèle HELP.

fonction extra-hyper-espace Pour transférer les valeurs d'un DPO entre deux hyper-espaces, le langage C-HELP offre la possibilité d'utiliser la primitive de transfert extra-hyper-espace. Nous montrerons que le fait d'avoir réservé exclusivement le transfert par cette primitive permet l'abord de la programmation hétérogène (cf. 5).

fonctions à géométrie générique Le langage permet l'écriture de bibliothèques, par le passage d'hyper-espaces en paramètres. On parle alors de fonctions à géométrie générique.

fonctions d'entrées/sorties Nous évoquerons le problème des entrées/sorties par la définition de primitives permettant le transfert des valeurs parallèles vers le monde scalaire.

4.1 Fonctions intrinsèques

Les primitives intrinsèques permettent de manipuler les valeurs relatives à la géométrie des DPO. Certaines d'entre elles produisent un DPO conforme au DPO sur lequel elles sont appliquées ; les autres produisent un résultat scalaire. Leur application se fait suivant la syntaxe `DPO.f()`.

Ces fonctions sont considérées comme faisant partie de la vue microscopique. Leur appel ne crée pas de nouveau segment conforme, mais l'argument DPO peut, lorsqu'il est composé d'une expression, être évalué avec création de segments conformes inférieurs.

4.1.1 Fonctions produisant un résultat DPO

cabs(d) donne un DPO conforme au DPO sur lequel s'applique la primitive et qui contient pour chaque point, la valeur de la coordonnée de ce point sur la dimension **d** relativement à l'origine de l'hyper-espace.

crel(d) donne un DPO conforme au DPO sur lequel s'applique la primitive et qui contient pour chaque point, la valeur de la coordonnée de ce point sur la dimension **d** relativement à l'origine du DPO argument (sur l'origine, **crel** donne 1 pour toute dimension).

4.1.2 Fonctions produisant un résultat scalaire

size(d) donne un scalaire de valeur égale à la taille de l'intervalle d'allocation du DPO source sur la dimension passée en argument.

low(d) donne un scalaire de valeur égale à la borne inférieure de l'intervalle d'allocation du DPO source sur la dimension passée en argument.

up(d) donne un scalaire de valeur égale à la borne supérieure de l'intervalle d'allocation du DPO source sur la dimension passée en argument.

4.2 Fonctions microscopiques

Une fonction microscopique permet de définir un traitement local à un point quelconque de l'hyper-espace. Les données présentes dans l'espace mémoire de ce point sont alors considérées comme scalaires et le point est lui-même considéré comme un processeur indépendant du reste de la machine cible.

Ces fonctions supportent donc toutes les constructions du langage C, à la condition d'être évaluables sur le point actif. **⚠** En particulier, l'usage de fonctions d'entrées/sorties y est interdit. Par contre, la récursivité, l'appel d'autres fonctions microscopiques, l'usage de structures de contrôle telle que le **if** ou le **while** sont autorisés et conservent les mêmes sémantiques, transférées au niveau du point actif.

Là encore, ces fonctions ne déterminent pas de nouveaux segments conformes. Seules les évaluations des paramètres d'appel peuvent en créer, pour le premier appel d'une fonction microscopique (l'appel d'une fonction microscopique à l'intérieur d'une autre fonction microscopique ne crée pas de nouveau segment).

4.2.1 Un modèle d'exécution asynchrone

Le modèle d'exécution de telles fonctions est proche du modèle asynchrone. En effet, si le programme est compilé pour une machine asynchrone, il n'est pas nécessaire de synchroniser les processeurs virtuels pendant l'évaluation de la fonction.

Pour mettre en œuvre ces fonctions microscopiques, le programmeur devra par conséquent écrire un code identique à un code développé pour une machine scalaire. La déclaration et la définition d'une fonction microscopique est précédée du mot-clef **micro**. Elle est écrite en C scalaire. Tous les paramètres seront considérés comme éléments de DPO lors de l'appel.

4.2.2 Appel d'une fonction microscopique

Les appels aux fonctions microscopiques sont effectués sur les points du domaine d'activité du segment conforme en cours d'évaluation. Les paramètres scalaires sont étendus au domaine d'appel de la fonction. De même, le résultat de la fonction est toujours considéré comme un DPO temporaire, conforme à ce domaine.

Comme pour le langage C, par convention, les paramètres (DPO ou scalaires, cf. infra) d'une fonction microscopique sont passés par valeur [HJ90]. Le passage explicite par adresse d'une variable suit le modèle présenté pour les pointeurs (1.4) : les adresses d'objets parallèles sont considérés au niveau de la mémoire du point. Les variables scalaires du programme sont visibles à l'intérieur de ces fonctions, Δ mais leur affectation y est interdite. De même, le passage par adresse d'une de ces variables n'a guère de signification : une adresse scalaire n'est pas valide dans la mémoire d'un point.

Si tous les paramètres d'appels sont scalaires, ils sont étendus au domaine de conformité englobant. Si celui-ci n'existe pas (quand la fonction microscopique est appelée sans paramètre DPO et en dehors d'un bloc déterminant explicitement le domaine de conformité), il y a erreur de programmation.

4.2.3 Exemple de fonction microscopique

```

micro unsigned int next(unsigned int Un)
{
  if (Un%2 == 0)
    return 3 * Un + 1;
  else
    return Un >> 1;
}
...

where(A!=1)
  A=next(A);

```

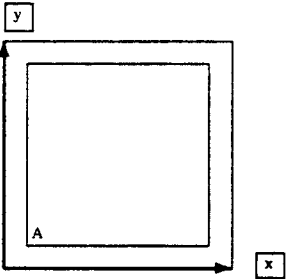


FIG. III.38 - Exemple de fonction microscopique

La fonction `next` du programme écrit en figure III.38 est une fonction écrite en code scalaire qui est appliquée sur les points du domaine d'activité courant (i.e. qui comporte une valeur différente de 1 pour le DPO `A`). Sur chacun de ces points, la valeur de `A` est passée en

paramètre effectif et le déclenchement de l'évaluation de la fonction est opéré indépendamment sur chaque point actif.

4.2.4 Bibliothèques microscopiques

Comme en HyperC [Hyp93], certaines bibliothèques du langage C peuvent être étendues à la vue microscopique. Par exemple, le programmeur a la possibilité d'appeler la bibliothèque mathématique à l'intérieur de l'évaluation d'un segment conforme. Pour ce faire, le langage offre la possibilité d'inclure une bibliothèque en spécifiant son extension au parallélisme microscopique. Chaque fonction de la bibliothèque se voit étendue sur la vue microscopique.

4.3 Passage de DPO en paramètre et retour de DPO

Les fonctions classiques du langage C sont généralisées pour C-HELP. Les DPO peuvent ainsi être passés en paramètre, ou faire l'objet d'un retour de fonction. Nous ne considérons dans ce paragraphe que les fonctions traitant des DPO alloués à l'intérieur d'hyper-espaces visibles, par opposition aux fonctions à géométrie générique que nous exposerons par la suite.

Ainsi, nous appellerons *fonction générale*, une fonction qui utilise les spécificités de C-HELP, à l'exclusion des autres familles clairement caractérisées.

Une fonction générale manipule exclusivement des paramètres DPO ou scalaires tous alloués dans un ou des hyper-espaces visibles lors de la compilation de la fonction. Le DPO retour est obligatoirement déclaré dans un hyper-espace visible.

4.3.1 Appel de fonctions générales

L'appel d'une fonction générale détermine un nouveau segment conforme. Par conséquent, les constructeurs `on` ou `where` englobant un tel appel n'ont ni d'influence sur l'évaluation des paramètres, ni à l'intérieur de la fonction appelée.

4.3.2 Passage par valeur de la référence

La déclaration des paramètres formels DPO comporte le type des éléments, et le nom de l'hyper-espace. Ils ne comportent pas de déclaration de géométrie d'allocation⁹.

9. Lors de la prise en compte des DPO statiques (*steady*), cette convention pourrait être remise en cause.

Le passage de paramètre se fait, dans le cas général, par valeur de la référence. Il est alors possible de modifier les valeurs des éléments du DPO passé en paramètre, mais la modification de son domaine d'allocation serait perdue au retour de la fonction. Par conséquent, le langage C-HELP interdit cette modification : Δ l'association d'un DPO reçu en paramètre par valeur de la référence est interdite.

4.3.3 Passage par adresse de la référence

Le passage par adresse d'un DPO à une fonction générale est possible. Dans ce cas, le programmeur doit compléter le paramètre formel en insérant l'étoile après le mot-clef `dpo` se rapportant au paramètre (par exemple : `dpo * int plan M`)¹⁰.

Lors de l'appel de la fonction, l'adresse de la référence est transmise à la fonction appelée. Les modifications du domaine d'allocation sont alors possibles, et récupérées en sortie de la fonction.

4.3.4 Retour de DPO

De façon similaire au passage de paramètres, le retour de DPO est autorisé pour une fonction générale. La fonction qui retourne un tel DPO doit être précédée de la spécification du type de retour :

```
dpo <type_éléments> <nom_hyper-espace> F(...)
```

La géométrie du DPO de retour ne peut pas être spécifiée lors de l'écriture d'une telle fonction, de par la dynamicité introduite dans le modèle.

L'instruction `return DPO_expr` renvoie le DPO résultant de l'évaluation de l'expression. Le domaine de conformité du segment conforme principal de l'expression devient le domaine d'allocation du DPO retourné.

Dans le cas d'un domaine d'activité non complet, les valeurs contenues dans le DPO de retour sur les points inactifs sont indéterminées.

Quand l'expression est limitée à un nom de DPO déclaré localement à la fonction, le compilateur ne libère pas le domaine d'allocation de ce DPO et une copie de la référence à ce

10. L'utilisation de l'étoile a été choisie pour rappeler le passage par adresse, mais cette déclaration ne correspond pas à un type « pointeur de données ».

domaine est renvoyée. Le retour d'un DPO déclaré localement à la fonction est ainsi permis dans le langage C-HELP (contrairement à certains tableaux en C).

4.3.5 Exemple

Dans un hyper-espace à trois dimensions, on écrit une fonction capable de transposer une matrice de dimension 2. Le paramètre de cette fonction est passé par valeur de la référence. Une autre fonction positionne le DPO passé en paramètre à l'origine de l'hyper-espace ; son paramètre est évidemment passé par adresse de la référence (cf. figure III.39).

```

hspace plan [x=100,y=100,z=100];

dpo int plan transpose2d(dpo int plan M)
{
  if (M.size(x)==1)
    return M.exchrel(y,z);
  if (M.size(y)==1)
    return M.exchrel(x,z);
  if (M.size(z)==1)
    return M.exchrel(x,y);
}

void moveorig(dpo * int plan M)
{
  M <- M.moveabs(1,1,1);
}

main()
{
  dpo int plan [x=20:80,y=10:20] A;
  dpo int plan [] B;

  B <- transpose2d(A);
  moveorig(A);
}

```

FIG. III.39 - *Le passage de paramètres par valeur ou adresse de la référence*

4.4 Fonction de transfert extra-hyper-espace

Il est interdit de faire interagir des données allouées sur deux hyper-espaces différents. Pour opérer un tel traitement, il est nécessaire, au préalable, de transférer un des deux DPO dans l'hyper-espace de l'autre DPO. On utilise pour cela la primitive de téléportation¹¹ :

```
void teleporter(source_DPO_name, dest_DPO_name);
```

Cette procédure accepte un DPO source en paramètre et le transfère dans un objet d'un autre hyper-espace, dont le nom est passé en second paramètre. Les domaines d'allocation de chacun des deux DPO doivent reposer exclusivement sur des dimensions primaires des hyper-espaces respectifs.

Les valeurs du DPO source sont lues en parcourant, dans l'ordre inverse de déclaration, les dimensions primaires de l'hyper-espace. Chaque valeur est envoyée sur le point du DPO destination, qui est aussi parcouru suivant le même principe.

▲ Le programmeur doit assurer que le nombre de point du domaine d'allocation de l'objet destinataire est supérieur ou égal au nombre de points du domaine d'allocation de l'objet source. Dans le cas contraire, le comportement du programme n'est pas défini. Par contre, si l'objet destination est plus grand (en nombre de points) que l'objet source, seuls les premières valeurs (dans le sens du parcours) seront affectées. Il n'y a pas de vérification systématique, laissée à la charge du programmeur.

Le DPO destination doit posséder une géométrie, et par conséquent une allocation mémoire. Il n'est pas autorisé d'appeler la primitive `teleporter` quand un des deux objets ne possède pas d'allocation parallèle (DPO de type `void` ou DPO déclaré `[]` et pas encore alloué).

4.4.1 Exemple

Un objet A d'un hyper-espace à n dimensions est transféré sur un objet B d'un autre hyper-espace à m dimensions. L'appel à la primitive de téléportation contient les deux DPO en paramètres, suivant l'ordre émetteur/destinataire (cf. figure III.40).

¹¹ **téléportation**: action de soustraire un objet du monde dans lequel il évolue pour le reconstruire dans un autre monde, à un coût généralement assez élevé.

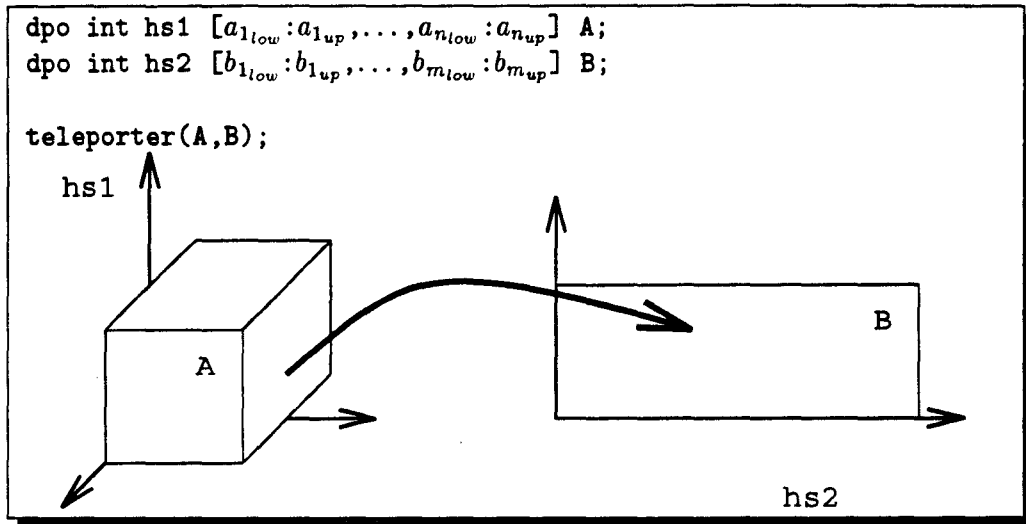


FIG. III.40 - Une téléportation

Nous reviendrons en section 5 sur l'usage de la téléportation dans l'écriture de programmes hétérogènes.

4.5 Fonctions à géométrie générique

Pour l'écriture de certaines fonctions, les objets ne peuvent pas être considérés comme faisant partie d'un hyper-espace déclaré globalement. C'est le cas des bibliothèques. Par exemple, une fonction de multiplication de matrices doit pouvoir être écrite en C-HELP, et appelée pour effectuer de tels traitements, sur n'importe quelles matrices de n'importe quel hyper-espace.

On introduit donc les fonctions à géométrie générique, par la possibilité donnée au programmeur d'écrire une fonction dans un référentiel abstrait, puis d'instancier une telle fonction générique en précisant l'hyper-espace appelant.

4.5.1 Écriture d'une fonction à géométrie générique

La fonction à géométrie générique est précédée du mot-clef **inside** et de la déclaration d'un hyper-espace et de ses dimensions primaires qui constitueront le cadre formel du code de la fonction.

L'hyper-espace ainsi déclaré en préambule de la fonction va permettre l'écriture du corps de la fonction générique. En utilisant cet hyper-espace formel, le programmeur est capable

d'écrire l'algorithme mis en place dans la fonction en respectant les règles habituelles du modèle HELP.

Bien évidemment, il n'y a pas d'information de taille, de distribution, ni d'arbre de priorité pour les dimensions de l'hyper-espace formel. Seuls le nom de l'hyper-espace et les noms de dimensions primaires sont précisés pour permettre l'écriture de l'algorithme mis en place à l'intérieur de la fonction. Il n'est pas possible de définir des dimensions secondaires pour ces hyper-espaces¹².

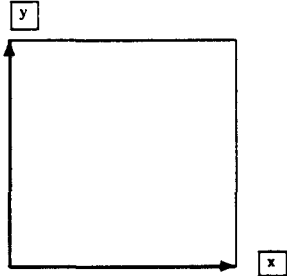
Les DPO paramètres peuvent être déclarés comme étant alloués dans cet hyper-espace, de même que l'éventuel DPO de retour. Les DPO locaux peuvent aussi être alloués dans cet hyper-espace. Les corps des fonctions à géométrie générique ne peuvent considérer que l'hyper-espace formel et les objets qui y sont déclarés. **⚠** Toute autre référence à des DPO d'autres hyper-espaces est interdite. Les variables scalaires globales sont toujours visibles à l'intérieur de la fonction, mais leur affectation y est interdite. Les manipulations de scalaires locaux ne comportent pas de limitations par rapport au langage C.

La figure III.41 montre une fonction d'inversion de matrices dont le géométrie est générique. L'hyper-espace formel permet l'écriture de l'algorithme qui s'exécutera à l'intérieur d'un plan dont les dimensions sont nommées *x* et *y*. Le DPO de retour est alloué virtuellement dans ce plan, suivant la même allocation que le DPO passé en paramètre.

```

inside plan[x,y] dpo float plan InvMat(dpo float plan M)
{
  dpo float plan idem(M) local_mat;
  ...
  /* manipulations dans plan[x,y] */
  ...
  return local_mat;
}

```



The diagram shows a square representing a 2D plane. The vertical axis is labeled 'y' and the horizontal axis is labeled 'x'. Both labels are enclosed in small boxes. The square is drawn with solid lines and is positioned to the right of the code block.

FIG. III.41 - Fonction à géométrie générique

Lors de l'écriture du code d'une fonction à géométrie générique, les conditions de validité des DPO doivent toujours être assurées par le programmeur, à l'intérieur de l'hyper-espace formel; en particulier, la règle de conformité est toujours en vigueur.

12. dans l'état actuel des spécifications du langage.

4.5.2 Appel d'une fonction à géométrie générique

L'appel d'une fonction à géométrie générique est effectué indépendamment de tout domaine contraint ou masqué englobant. Il crée un nouveau segment conforme pour chaque paramètre.

Pour l'appel, le programmeur instancie l'hyper-espace formel dans l'hyper-espace appelant par l'association géométrique des dimensions :

Le nom de l'hyper-espace appelant est précisé avant le nom de la fonction à appeler. Pour toute dimension de l'hyper-espace appelant, une information doit ensuite être présente dans la déclaration : après la donnée du nom de chacune de ces dimensions, le programmeur précise si la dimension est fixée à une certaine coordonnée, ou associée à une dimension de l'hyper-espace à géométrie générique ;

- Si une dimension est fixée, une information entière détermine la coordonnée sur cette dimension du sous-espace de l'espace appelant dans lequel va s'exécuter la fonction appelée;
- Si une dimension est associée, elle se retrouve entre crochets lors de l'appel. Le programmeur précise alors l'intervalle de coordonnées dans lequel la fonction sera appelée. Il a la possibilité de préciser la borne inférieure et la borne supérieure (':') ou la borne inférieure et la taille (';') de l'intervalle (comme pour la déclaration de DPO).

Il est bien évident que le nombre de dimensions associées dans l'appel de la fonction doit correspondre au nombre de dimensions déclarées de l'hyper-espace formel.

Ainsi, l'hyper-espace à géométrie générique est positionné précisément dans l'hyper-espace appelant et la fonction s'exécute à l'intérieur de ce sous-hyper-espace. Les DPO passés en paramètres (et le DPO de retour) doivent être alloués dans ce sous-hyper-espace (cf. exemple en figure III.42).

⚠ L'appel d'une telle fonction avec un scalaire en paramètre effectif pour un paramètre formel DPO attendu est interdit : on ne peut déduire la géométrie du paramètre effectif en entrée de la fonction.

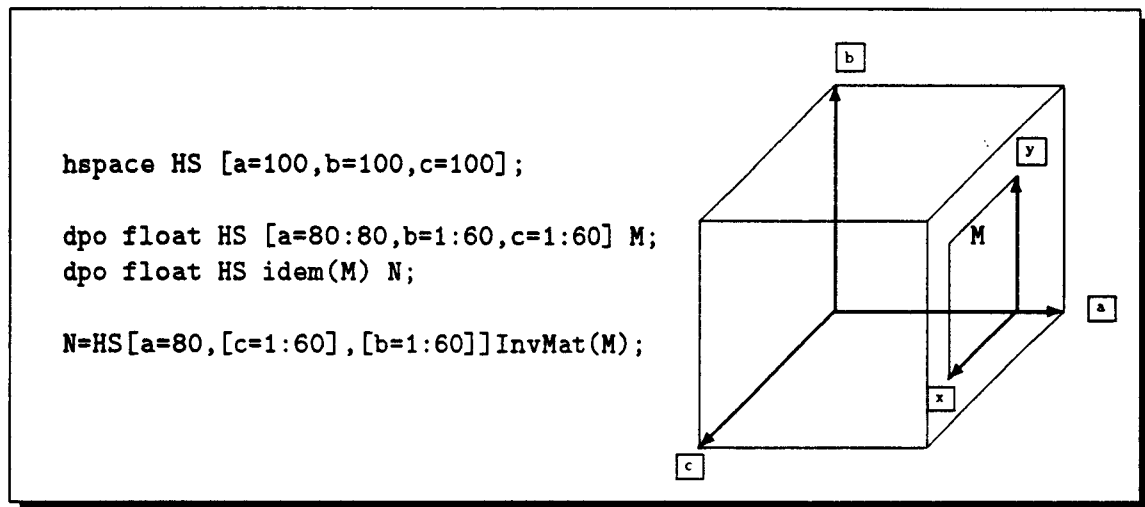


FIG. III.42 - Appel de fonction à géométrie générique

4.6 Les entrées/sorties

Dans la version actuelle du langage, toute entrée/sortie se fait à partir du monde scalaire, à l'aide des primitives du langage C standard. L'intégration du parallélisme dans les entrées/sorties d'un langage est toujours un problème difficile, sur lequel nous ne nous sommes pas encore penchés.

Trois primitives permettent de réaliser les échanges de données entre le monde scalaire et l'hyper-espace : la première est dédiée au transfert d'une valeur unique, les deux autres permettent le transfert entre tableaux scalaires monodimensionnels et DPO.

4.6.1 Primitive scalar

La primitive `scalar` s'applique comme une primitive macroscopique sans domaine de contrôle macroscopique. Elle peut s'appliquer après une suite d'opérateurs macroscopiques.

Les arguments correspondent aux coordonnées d'un point sur les dimensions primaires de l'hyper-espace. La valeur en ce point du DPO sur lequel s'applique la primitive est convertie en scalaire et devient visible sur tout autre point, y compris les points d'autres hyper-espaces. (Voire un exemple de l'utilisation de cette primitive dans l'algorithme de Gauss-Jordan, au chapitre 6).

Si le DPO source n'existe pas au point considéré par les arguments, il y a comportement indéterminé.

4.6.2 Primitive d'importation

La primitive d'importation est destinée à transférer un ensemble de valeurs scalaires vers un DPO alloué sur un hyper-espace.

```
hspace hs [x=100,y=100];  
  
float TabSca[10000];  
dpo float hs [x=*,y=*] M;  
  
import(TabSca,M);
```

FIG. III.43 - import

La sémantique associée à cette instruction (cf. figure III.43) est comparable à celle définie pour la téléportation (cf. 4.4). Les points du DPO cible sont parcourus sur chaque dimension primaire de l'hyper-espace, dans l'ordre inverse de déclaration de ces dimensions primaires.

Pour chaque point, une valeur est lue dans le tampon scalaire et importée vers le DPO destination. **▲** Le programmeur doit assurer que le tampon contient suffisamment de valeurs pour compléter le DPO. L'importation sur un DPO qui ne possède pas de géométrie ('□') est interdite. Il n'y a pas de vérification automatique.

4.6.3 Primitive d'exportation

La primitive d'exportation est destinée à transférer les valeurs d'un DPO alloué sur un hyper-espace vers le monde scalaire (cf. figure III.44).

```
hspace hs [x=100,y=100];  
  
float TabSca[10000];  
dpo float hs [x=*,y=*] M;  
  
export(M,TabSca);
```

FIG. III.44 - export

La sémantique associée à cette primitive est voisine de celle de la primitive d'importation.

⚠ Les mêmes restrictions doivent être respectées par le programmeur : l'allocation mémoire du tampon scalaire doit être suffisante pour accueillir toutes les données du DPO à exporter. Il n'y a pas non plus de vérification automatique.

5 C-HELP et l'hétérogénéité

LA PROGRAMMATION de machines hétérogènes connaît un engouement depuis peu. L'idée de base est relativement simple : on veut exploiter la puissance de calcul de machines réparties sur un réseau local ou plus étendu, de modèles d'exécution divers. On espère ainsi partager l'algorithme global sur des capacités de calculs coopérantes.

Une application hétérogène est par extension un programme qui s'exécute sur plusieurs machines différentes.

Les problèmes posés par ce type de parallélisme couvrent un spectre important :

- quel doit être le grain de parallélisme entre les tâches de la même application ?
- quels mécanismes mettre en œuvre pour gérer les communications et synchronisations ?
- doit-on cacher au programmeur ces communications ? ou même le fait que son application soit exécutée sur plusieurs machines ?
- peut-on décider automatiquement la scission d'une application ? du placement des données ?
- quel langage pour développer une application hétérogène ?

Nous ne prétendons pas répondre à toutes ces questions. Par contre, nous allons montrer que le modèle HELP, et le langage C-HELP se prêtent bien à la programmation hétérogène.

Nous venons de montrer ce que propose C-HELP en terme de séparation des communications macroscopiques et des calculs microscopiques. Toute communication interne à un hyper-espace est clairement identifiée. Il en est de même pour les communications entre deux hyper-espaces différents. Deux hyper-espaces intervenant dans un même programme sont clairement séparés et leurs interactions sont limités à des communications explicites modélisées par des téléportations. Le parallélisme potentiel qui existe entre les parties de code respectifs des hyper-espaces est, de toute évidence, à gros grain.

Pour cette raison, nous pouvons considérer que la programmation d'une application faisant intervenir plusieurs hyper-espaces peut être vue sous l'angle de l'hétérogénéité. Chaque hyper-espace de l'algorithme peut être supporté par une machine (ou une tâche) qui lui est propre.

Les communications entre deux hyper-espaces, par l'appel de la primitive de téléportation, modélisent les communications sur le réseau qui relie les deux machines.

La primitive de téléportation utilise des couches basses du système et permet de transférer des objets. Pour l'hyper-espace émetteur, cette opération est non-bloquante, mais devient bloquante pour la réception de valeurs¹³.

Prenons un exemple élémentaire pour illustrer la possibilité d'écrire un code source C-HELP pour l'exécution hétérogène. Nous voulons trier les éléments d'un vecteur de 128 000 nombres, alloué dans un hyper-espace de départ. Pour exploiter en parallèle la puissance de deux machines, un algorithme consiste à trier des tranches du vecteur sur l'une des machines, et à effectuer la fusion des tranches triées sur une autre machine. Pour cela, nous allons découper le vecteur en tranches de 128 nombres et utiliser un second hyper-espace 128×128 qui effectuera le tri de chaque tranche en une complexité de $O(1)$.

Puis, une fois la tranche triée, elle sera renvoyé à un hyper-espace d'arrivée qui effectuera la fusion avec les tranches déjà traitées. Pour cette étape de l'algorithme, nous déclarons un hyper-espace 128000×128 dans lequel nous développerons l'algorithme de fusion en une complexité de $O(1)$. Le vecteur trié est finalement renvoyé dans le premier hyper-espace, pour mettre à jour le vecteur de départ.

Durant le déroulement de cet algorithme hétérogène, le tri d'une tranche peut s'effectuer en parallèle avec l'insertion de la précédente. Voir figure III.45.

Sur cet exemple, le découpage du programme selon les hyper-espaces utilisés est flagrant, et le parallélisme à gros grain peut être mis en valeur (du fait de sa prise en compte lors de l'écriture de la fonction) : C-HELP peut donc fournir un moyen d'expression d'algorithmes hétérogènes. Il nous restera à définir la façon de déterminer l'association des hyper-espaces d'un programme aux machines du réseau hétérogène. Cette répartition ne peut se faire, dans l'état actuel du domaine, qu'à partir de directives exprimées par le programmeur. D'après le découpage que nous venons d'exposer, ces informations seront attachées à la notion d'hyper-espace.

13. La géométrie du DPO receveur permet de connaître le nombre de valeurs à lire en entrée.

```

hspace depa [l=128000];          /* la machine de départ */
hspace tri  [a=128,b=128];      /* la machine de tri */
hspace arri [x=128000,y=128];   /* la machine d'arrivée */

tri() {
  int i;
  dpo float depa [x=*,y=1] V;    /* le vecteur à trier */
  dpo float tri  [a=*,b=1] deso; /* une tranche désordonnée */
  dpo float arri [y=1:128] ordo; /* la tranche à fusionner */
  dpo float arri [x=128000] resu; /* le vecteur trié */
  dpo char  arri [x=128,y=*] tmp; /* temporaire pour fusion */
  dpo void  arri [x=128]  deja; /* cadre des déjà triés */
  dpo void  arri [x=256]  fait; /* cadre fin d'itération */

  for (i=1;i<1000;i++) {
    /* envoi de la tranche suivante du vecteur */
    teleporter(V.extrabs(1,i*128:(i+1)*128-1),deso);

    /* l'algorithme de tri est ensuite effectué dans le plan */
    deso = deso.scatter(
      (deso.exchabs(a,b).expand(a)>deso.expand(b)?1:0)
      .reduceadd(b)+1 , 1 , deso );

    teleporter(deso,ordo); /* envoi de la tranche pour fusion */

    /* insertion de la tranche ordonnée dans le vecteur resu */
    if (i==1) resu <- ordo.exchabs(x,y);
    else{
      /* les comparaisons pour calculer les décalages */
      tmp <- (on(deja)resu).expand(y) >
        ordo.expand(x,(i-1)*1000) ? 1:0;

      /* calcul et exécution des décalages des déjà triés */
      resu = (on(deja)resu).scatter(
        cabs(x) + tmp.(tmp==1)reduceadd(x).exchabs(x,y),
        cabs(y),fait);

      /* calcul et exécution des décalages des nouveaux élém */
      resu = ordo.scatter(
        cabs(y) + (on(tmp)1).(tmp==0)reduceadd(y),
        cabs(x),fait);

      /* agrandissement pour le prochain scatter */
      if (i<>999) fait <- resu.extrabs(x,(i+1)*1000);
    }
  }
  teleporter(resu,V);
}

```

FIG. III.45 - C-HELP et l'hétérogénéité

Pour la phase de compilation, nous savons déterminer dans le code source C-HELP les opérations spécifiques à un hyper-espace et celles qui le sont pour un autre hyper-espace. Nous savons donc séparer les tâches à produire. Les interactions se font uniquement par téléportation, ce qui regroupe, dans une unique fonction à mettre en œuvre, toute possibilité de communication de données entre les tâches.

De plus, pour obtenir un recouvrement entre les temps de communication et les temps de calcul, il suffit de rendre l'émission de valeur par téléportation non-bloquante. Un hyper-espace peut ainsi envoyer ses valeurs et continuer l'exécution de sa partie de code. L'utilisation d'une couche de communication entre tâches s'exécutant sur réseau hétérogènes (comme PVM) permet de laisser à la charge du système la gestion de tampons qui permettent de libérer la tâche émettrice.

En reprenant l'exemple précédent, nous pouvons séparer les trois tâches correspondant aux trois hyper-espaces du programme. Les figures III.46, III.47 et III.48 montrent les trois pseudo-codes (les fonctionnalités utilisées doivent être affinées...), la figure III.49 présente l'algorithme de fusion utilisé.

```

hspace line [x=128000];          /* première machine      */

tri() {
  int i;
  dpo float line [x=*]      D;   /* le vecteur à trier    */

  for (i=0;i<1000;i++) {
    /* envoi d'une tranche du vecteur */
    send(D.extrabs(x,i*128:(i+1)*128-1), ... );
  }

  /* attente du vecteur trié */
  receive(D,...);
}

```

FIG. III.46 - *La tâche de départ...*

```

hspace tri [a=128,b=128];          /* la machine de tri      */

tri() {
  int i;
  dpo float tri [a=*,b=1] deso; /* une tranche désordonnée */

  /* boucle traitant une à une les tranches      */
  for (i=1;i<1000;i++) {
    /* réception de la tranche suivante du vecteur      */
    receive(deso, ... );

    /* l'algorithme de tri est ensuite effectué dans le plan */
    deso = deso.scatter(
      (deso.exchabs(a,b).expand(a)>deso.expand(b)?1:0)
      .reduceadd(b)+1 , 1 , deso
    );

    /* envoi de la tranche triée pour fusion      */
    send(deso, ... );
  }
}

```

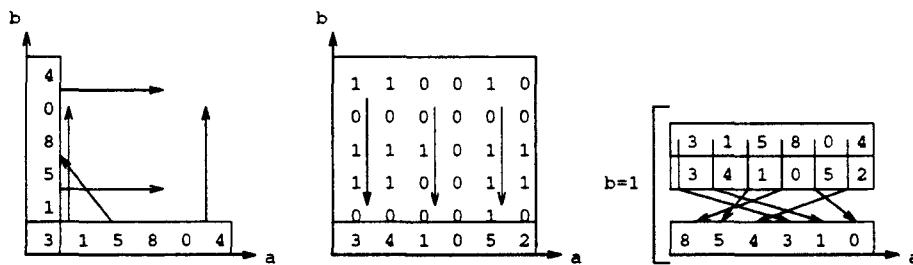


FIG. III.47 - ...la tâche de tri...

```

hspace arri [x=128000,y=128];      /* la machine d'arrivée */

tri() {
  int i;
  dpo float arri [y=1:128] ordo; /* la tranche à fusionner */
  dpo float arri [x=128000] resu; /* le vecteur trié */
  dpo char arri [x=128,y=*] tmp; /* temporaire pour fusion */
  dpo void arri [x=128] deja; /* cadre des déjà triés */
  dpo void arri [x=256] fait; /* cadre fin d'itération */

  /* boucle traitant une à une les tranches */
  for (i=1;i<1000;i++) {
    /* réception de la tranche triée pour fusion */
    receive(ordo, ... );

    /* insertion de la tranche ordonnée dans le vecteur resu */
    if (i==1) resu <- ordo.exchabs(x,y);
    else{
      /* les comparaisons pour calculer les décalages */
      tmp <- (on(deja)resu).expand(y) >
              ordo.expand(x,(i-1)*1000) ? 1:0;

      /* calcul et exécution des décalages des déjà triés */
      resu = (on(deja)resu).scatter(
              cabs(x) + tmp.(tmp==1)reduceadd(x).exchabs(x,y),
              cabs(y),fait );

      /* calcul et exécution des décalages des nouveaux élém */
      resu = ordo.scatter(
              cabs(y) + (on(tmp)1).(tmp==0)reduceadd(y),
              cabs(x),fait);

      /* agrandissement pour le prochain scatter */
      if (i<>999)
        fait <- resu.extrabs(x,(i+1)*1000);
    }
  }

  send(resu, ... );
}

```

FIG. III.48 - ...et la tâche de fusion

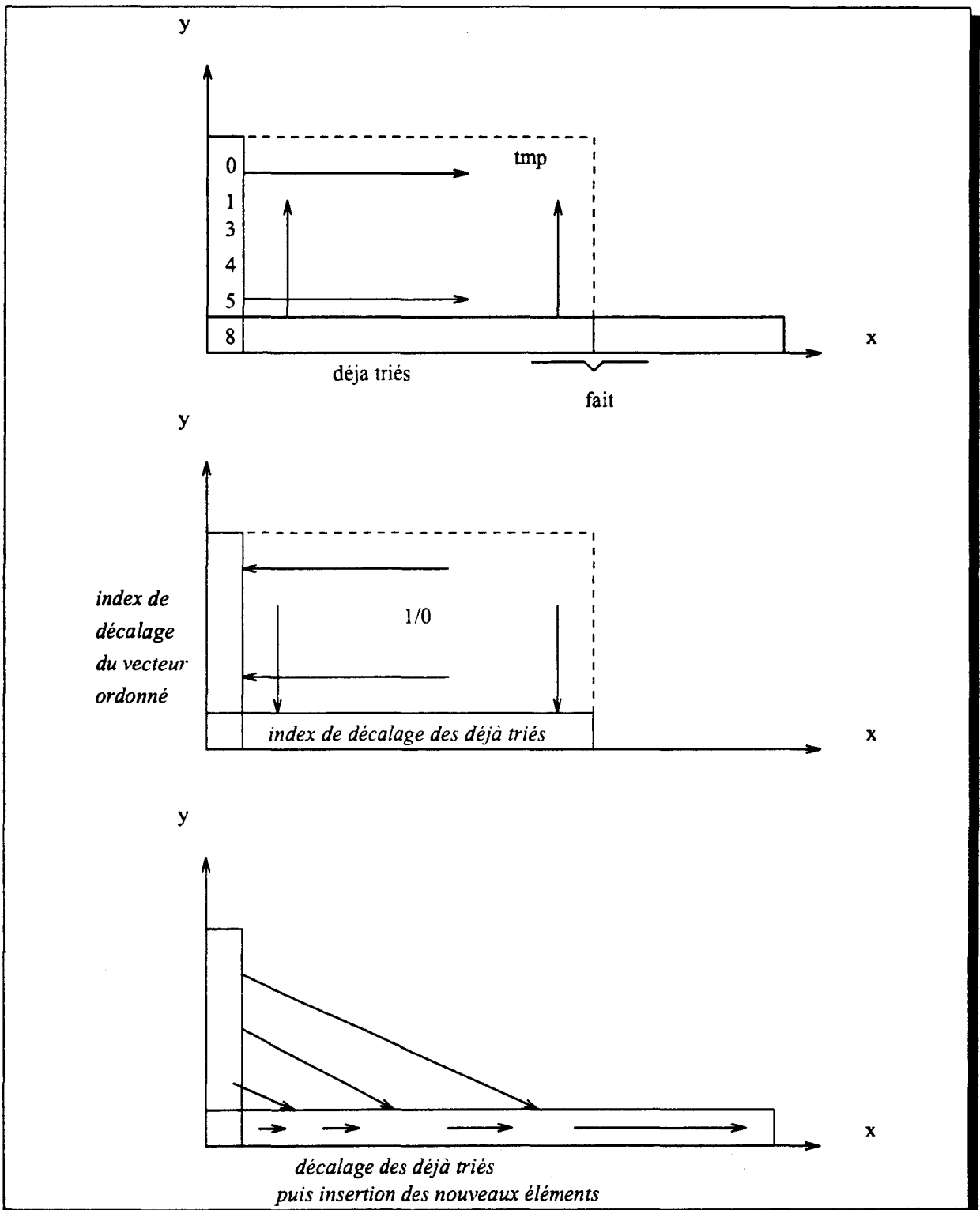


FIG. III.49 - Algorithme de fusion

Il n'y a donc aucune synchronisation artificielle à introduire dans le code source. Il suffit de séparer les parties de code concernant des hyper-espaces différents ; à un « détail » près : celui de la gestion des scalaires. De nombreux travaux ont sur ce point proposé des solutions efficaces que le compilateur C-HELP pourrait intégrer.

6 Conclusion

C E CHAPITRE nous a permis de mettre en œuvre le modèle de programmation HELP dans le langage C-HELP qui incite le programmeur à reprendre la notion d'hyper-espace dans toutes les étapes du développement de son algorithme.

Après avoir montré les objets de base du langage, nous avons proposé une hiérarchisation des expressions qui a permis de formaliser les deux vues existantes dans le langage. La vue microscopique modélise l'asynchronisme de l'évaluation point à point, tandis que la vue macroscopique engendre les communications par l'usage de primitives géométriques.

La caractérisation de plusieurs types de fonctions, suivant leur influence quant aux objets de base du langage, permet de proposer au programmeur plusieurs outils pouvant lui permettre de développer ses applications avec un langage plus proche de l'algorithme mis en place.

Chapitre IV

Compilation du langage C-Help

APRÈS avoir défini le modèle HELP puis le langage C-HELP, nous nous penchons dans ce chapitre sur les techniques de compilation qui peuvent découler des points particuliers du modèle de programmation géométrique. Dans un premier temps, nous présenterons le point de départ de notre travail de développement du compilateur à savoir la machine cible et la justification du langage intermédiaire choisi. Après une vue globale de l'atelier de développement du compilateur, nous étudierons l'allocation mémoire adoptée en fonction du modèle de programmation basé sur le référentiel que constitue l'hyper-espace. Nous détaillerons l'algorithme parallèle d'allocation mémoire introduit dans notre compilateur. Nous montrerons ensuite que l'introduction d'irrégularité dans la boucle de virtualisation a permis de réduire le nombre d'itérations nécessaires au balayage induit par la phase de virtualisation. Pour finir, nous interpréterons des mesures de temps d'exécutions de programmes tests mettant en valeur les bénéfices tirés des techniques de génération de code particulières introduites dans notre compilateur.

1 L'atelier de compilation

LE PREMIER compilateur C-HELP est développé pour générer du code exécutable sur une machine massivement parallèle synchrone : la MasPar MP-1. Le choix a été fait de générer un langage intermédiaire spécifique à cette machine : le *MasPar Programming Language* (MPL) [Mas91c]. Le choix s'est porté sur ce langage issu de C pour éviter la lourdeur de mise en œuvre d'une génération de code de très bas niveau comme l'assembleur. Le fait que ce langage soit non-virtuel et explicite permet de gérer précisément l'exécution par l'utilisation des constructeurs de base qui reflètent directement l'architecture de la machine.

1.1 Machine cible

La MasPar MP-1 est une machine massivement parallèle synchrone à mémoire distribuée. Dans sa version maximale, elle comporte 16 384 processeurs élémentaires chacun muni de

64 Koctets de mémoire vive. La machine est pilotée par un processeur scalaire qui déroule le code exécutable et diffuse le micro-code des instructions parallèles aux PE. Ce processeur scalaire est appelé ACU¹. Cette machine comporte la particularité de proposer deux réseaux de communication entre les PE.

Le premier de ces réseaux est une grille 2D torique qui permet des communications régulières suivant les directions Nord, Est, Sud, Ouest et les composées diagonales (NE, NW, SW, SE). Le second réseau permet des communications irrégulières par l'intermédiaire de trois niveaux de cross-bar dispensant une capacité de 1024 liaisons simultanées d'un processeur à un autre². Voir figure IV.1³.

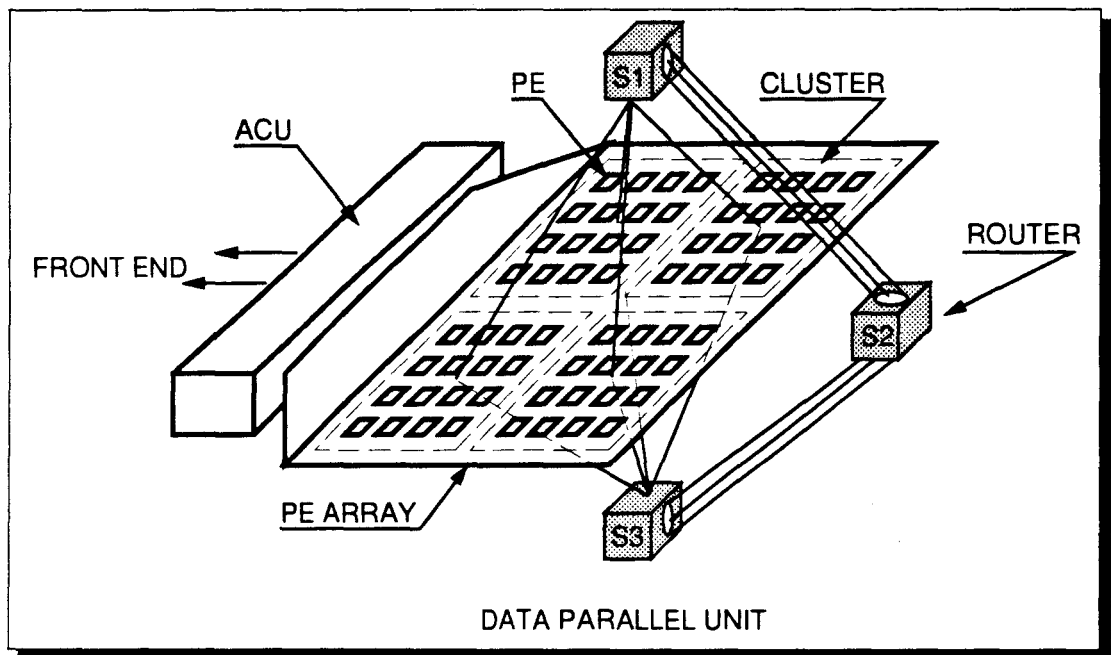


FIG. IV.1 - L'architecture de la MasPar

Suivant le modèle d'exécution SIMD que l'on a présenté au premier chapitre, le parallélisme d'exécution est obtenu grâce à l'allocation des variables parallèles, dont chaque processeur physique prend en charge un élément. Une instruction parallèle est ensuite exécutée simultanément par tous les processeurs actifs. Le taux maximal de parallélisme est donc égal au nombre de processeurs de la machine.

La machine est relié au monde extérieur par l'intermédiaire d'une station de travail UNIX,

1. Array Control Unit.
2. Pour ce faire, le compilateur MPL partage l'accès au routeur par la génération de boucles d'accès pour les processeurs d'un même cluster (groupe de 16 processeurs).
3. Merci à Boris Kokozsko pour ce schéma.

appelée machine hôte⁴.

1.2 Le langage intermédiaire

Le choix d'adopter MPL comme langage intermédiaire est motivé par la facilité de mise en œuvre. En effet, comme C-HELP, le langage MPL est lui aussi proche du C, ce qui va nous permettre de reporter sur le compilateur MPL une bonne partie de la génération de code à partir de C-HELP sans avoir à le modifier. En particulier, la gestion des variables scalaires et les entrées/sorties vers le frontal sont directement « déchargées » sur le compilateur MPL.

MPL permet la programmation de bas niveau de la MasPar. On peut considérer que c'est un langage :

- à données parallèles explicites ;
- non-virtuel ;
- à communications explicites.

Le modèle de programmation MPL coïncide parfaitement avec l'architecture de la machine. Les variables scalaires d'un programme sont déclarées comme des variables ordinaires du C et se trouvent allouées sur l'ACU. Les variables parallèles sont déclarées par le mot-clef **plural** et sont alors allouées avec une instance sur chaque PE : la taille des variables parallèles est égale à la taille de la grille des processeurs.

Le mécanisme d'inhibition de processeurs est exprimé par l'extension des structures de contrôle au flot parallèle de données : il existe les constructeurs **if**, **else**, **for**, **while**... pour les expressions parallèles. Le compilateur se charge alors de gérer l'activité sur chaque processeur.

Les communications sont directement explicitées par le programmeur qui peut, suivant le schéma à mettre en place, utiliser la grille torique (instructions **xnet**) ou utiliser le cross-bar (instruction **router**).

Les communications entre le monde scalaire et le monde parallèle se font par diffusion (un scalaire est visible pour tous les PE) ou par réduction : le constructeur **proc** permet le rapatriement d'une valeur d'un PE spécifique vers une variable scalaire, tandis que des bibliothèques MPL mettent à disposition du programmeur des fonctions associatives exécutables sur l'ensemble des PE pour les calculs intervenant sur un ensemble de valeurs réparties (par exemple, opérer un **or** logique sur des conditions distribuées)⁵.

4. Front End.

5. Au niveau matériel, un arbre binaire permet de telles opérations en un temps proportionnel au logarithme du nombre de processeurs.

1.3 Outils de développement

L'esprit principal du développement du compilateur est dans un premier temps de prouver que le langage est raisonnablement compilable, et de montrer que le modèle HELP permet des optimisations dans les techniques de compilation. C'est sur ces points particuliers que nous focaliserons notre étude, sachant qu'il sera probablement possible d'en faire profiter d'autres compilateurs orientés vers d'autres types d'architectures, en particulier vers les machines asynchrones.

Le compilateur C-HELP a été développé à l'aide de l'ouvrage de Allen Holub *Compiler Design in C* [Hol90]. La compilation est effectuée suivant le schéma de la figure IV.2.

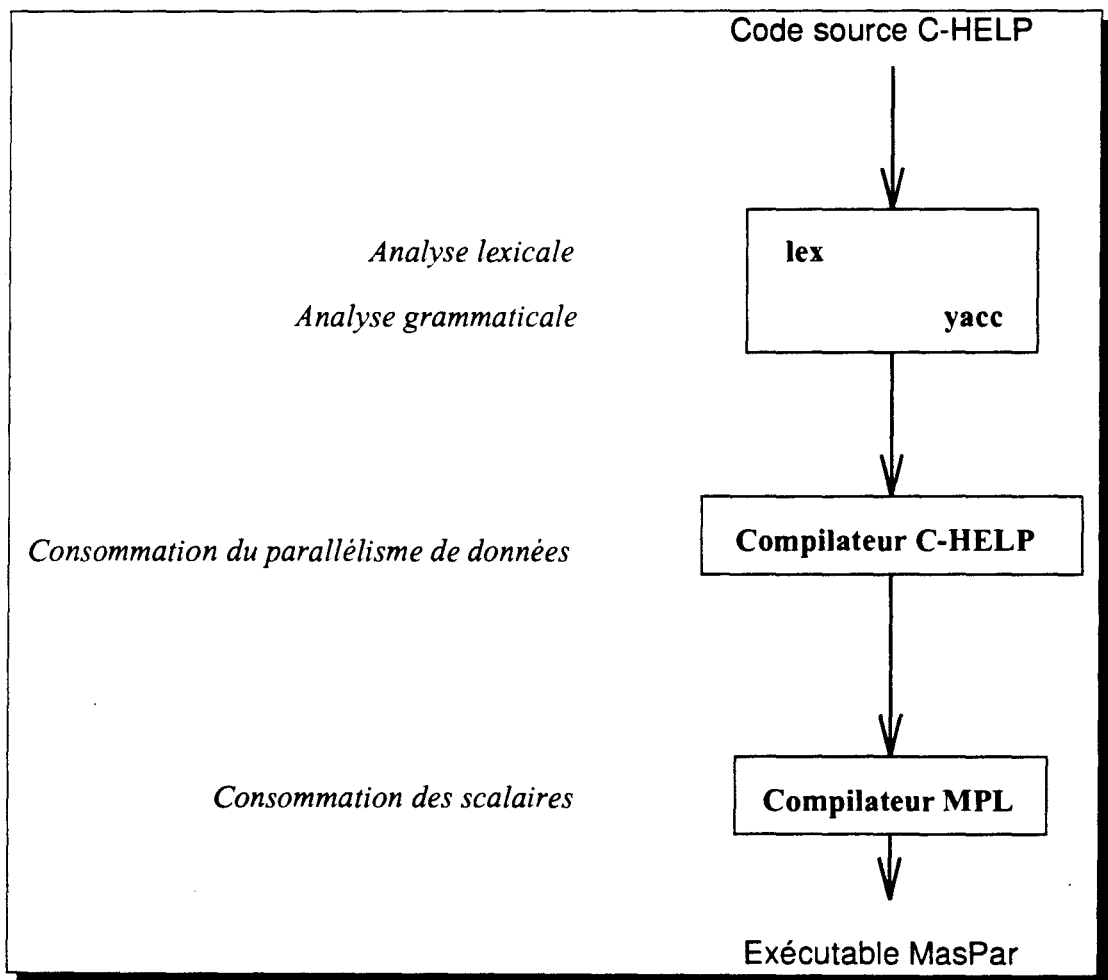


FIG. IV.2 - Atelier de développement du compilateur

L'utilisation des outils Unix classiques (Lex et Yacc), ainsi que de la grammaire ANSI-C

courante permet d'obtenir rapidement un noyau de compilateur sur lequel peuvent venir se greffer les techniques proposées.

2 Représentation des données à l'exécution

AVANT de proposer des techniques dédiées à la compilation du modèle HELP, il est nécessaire de présenter les structures de base pour la gestion des objets qui interviennent dans un programme. Dans cette partie, nous montrerons de quelle façon ces entités collaborent pour exécuter les constructions HELP.

2.1 DPO

Les DPO du langage possèdent une géométrie relative à l'hyper-espace dans lequel ils sont alloués. Par conséquent, pendant la phase d'exécution, chaque DPO visible possède un entête comportant ces informations de géométrie.

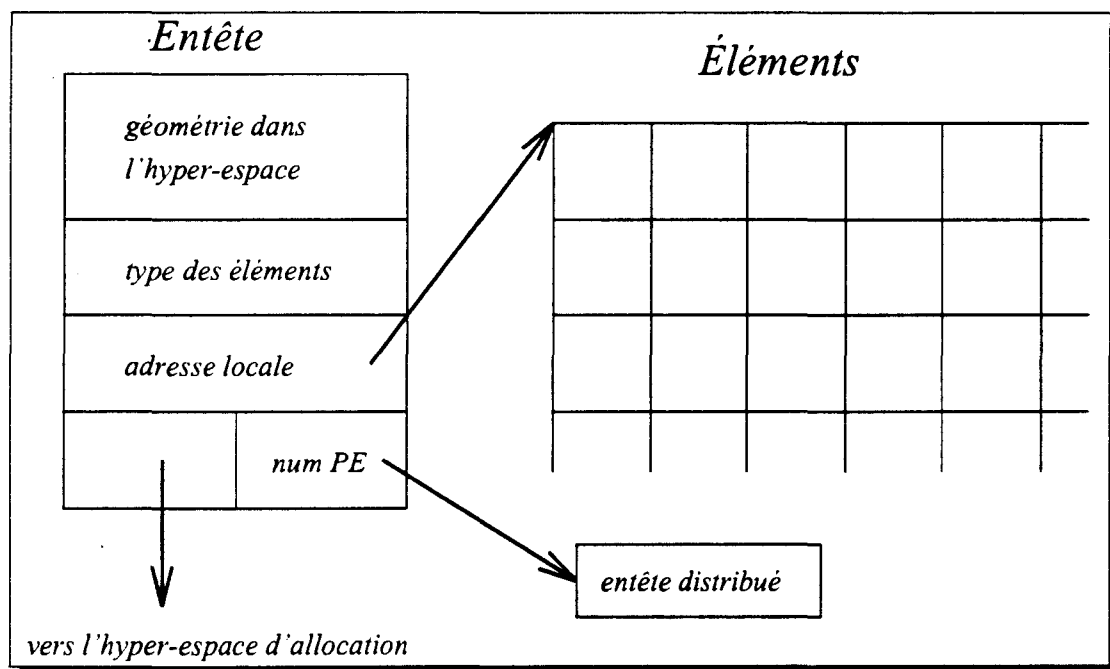


FIG. IV.3 - Les DPO à l'exécution

La figure IV.3 montre l'allocation des informations présentes à l'exécution pour un DPO. On y trouve deux parties distinctes, l'une est scalaire, l'autre est parallèle.

La partie scalaire, appelée *entête* et allouée sur l'ACU, comprend les informations de géométrie du DPO sous forme des bornes de l'intervalle de coordonnées pour toute dimension

de l'hyper-espace, quelle que soit la forme du DPO. On est ainsi capable de gérer dans la même structure tous les changements d'allocation du DPO dans l'hyper-espace. En outre, l'entête contient les informations suivantes (que nous détaillerons par la suite) :

- type des éléments du DPO (taille d'occupation mémoire pour un élément) permettant l'allocation dynamique⁶ ;
- un pointeur vers la structure associée à l'hyper-espace d'allocation ;
- l'adresse locale des éléments (voir 3.1) ;
- un numéro de processeur pour l'allocation dynamique (voir 3.1.3).

La partie parallèle du DPO contient les valeurs de ses éléments. Nous détaillerons par la suite le système d'allocation dynamique de cette partie parallèle du DPO.

2.2 Hyper-espaces

Le modèle HELP est basé sur la géométrie qui s'exprime par des opérations diverses, toutes en rapport avec l'hyper-espace. Pour compiler le langage associé, nous avons inclus dans les données du programme une structure pour accéder dynamiquement aux informations relatives à l'hyper-espace.

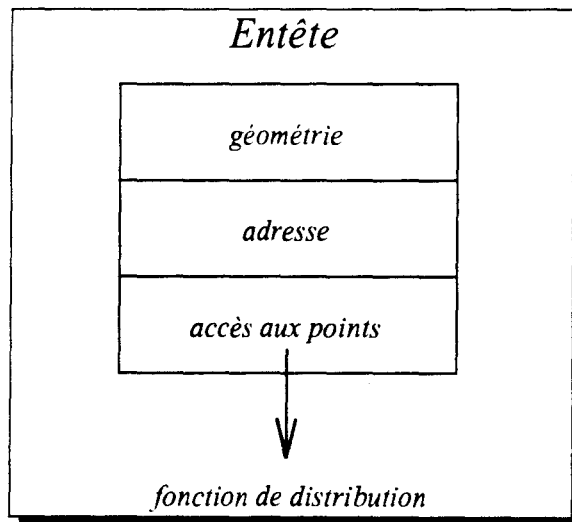


FIG. IV.4 - L'hyper-espace à l'exécution

6. Cette information n'est pas absolument nécessaire car évaluable à la compilation.

Un hyper-espace est alloué au cours de l'exécution sous forme d'un unique entête (cf. figure IV.4). La structure de donnée allouée sur l'ACU comprend :

- les informations de géométrie: nombre et taille des dimensions ;
- l'adresse de base d'allocation des points (voir 3.1);
- un pointeur vers la fonction de conversion de coordonnées.

La distribution des points étant associée à l'hyper-espace, c'est dans son entête que l'on retrouve un pointeur vers le code permettant l'accès aux variables parallèles. Tous les accès aux données parallèles seront réalisés par conversion de coordonnées d'un point de l'hyper-espace en adresse de la mémoire parallèle (numéro de processeur + adresse locale). Cette fonction consomme donc la virtualisation. Elle est « simplement » constituée d'un calcul arithmétique pour chaque dimension de l'hyper-espace (selon la taille des blocs de communication, la taille de chaque dimension de l'hyper-espace, l'arbre de priorité entre les dimensions). On peut définir cette fonction de distribution formellement :

$$\begin{array}{lcl} conv_{hs} : & \mathcal{D}_{hs} & \rightarrow (\mathbf{N} \times \mathbf{N}) \\ & (x_1, x_2, \dots, x_n) & \rightarrow (\#PE, Adr) \end{array}$$

3 Conséquences du modèle HELP

TOUTES les techniques proposées pour le compilateur C-HELP sont des conséquences directes du modèle de programmation HELP. Le fait d'avoir considéré une sémantique basée sur le référentiel nous permet de proposer une allocation mémoire associée à ce référentiel.

Nous présenterons d'abord ce système d'allocation, nous en déduirons une caractéristique intéressante pour les calculs d'adresse d'objets intervenant dans une expression. Nous détaillerons l'algorithme d'allocation dynamique. Puis, nous exposerons le problème de la boucle de virtualisation et la solution choisie en conséquence des concepts introduits par le modèle.

3.1 Allocation mémoire

L'allocation mémoire est un point crucial dans l'obtention de performances. Dans le modèle à parallélisme de données, tous les langages proposent des concepts qui permettent de diriger le code généré vers une solution minimisant le nombre de communications entre deux processeurs. Le bon placement des données est directement satisfait par une allocation mémoire cohérente avec le modèle de programmation. Ainsi, quand le modèle permet l'utilisation

d'objets parallèles réguliers, la compilation ne pourra être efficace que si l'allocation mémoire propose une organisation des données proche de la topologie logique de ces données.

3.1.1 Allocation par les points

Le modèle HELP introduit la notion de conformité et, par conséquent, induit la séparation entre la vision microscopique qui donne à une expression une portée locale au point, et la vision macroscopique dans laquelle les communications sont effectuées.

Nous avons montré que deux éléments qui interagissent pendant l'évaluation d'une expression microscopique se trouvent sur le même point de l'hyper-espace. Dès lors, en adoptant le principe de projeter toutes les données d'un point sur le même processeur physique, deux éléments qui sont alloués sur le même point sont aussi physiquement alloués sur le même processeur. On n'a donc aucune communication pendant les calculs microscopiques, la conformité est retrouvée physiquement sur la machine cible.

La solution sémantique adoptée a pour conséquence d'attribuer à chaque point de l'hyper-espace un espace mémoire qui lui est propre. On peut faire l'analogie entre un point de l'hyper-espace et un processeur virtuel qui dispose de sa mémoire privée ; et par conséquent entre un hyper-espace et une machine régulière à mémoire distribuée.

Dans la version actuelle du compilateur, la mémoire de chaque processeur est divisée statiquement pour être répartie sur les points projetés sur chaque processeur. Ce partage est équitable entre les points de l'hyper-espace projeté. L'allocation globale des points est donc opérée statiquement. L'étude de l'introduction de dynamisme de l'allocation des points doit être envisagée.

La figure IV.5 représente le processus d'allocation des DPO. Un DPO possède une adresse fixe par rapport à la base des points de son domaine d'allocation⁷. Chaque point est ensuite projeté sur la machine physique, suivant le résultat de l'évaluation des directives de distribution.

7. Le partage d'une adresse par deux DPO disjoints fera l'objet d'une étude lors de l'exposé du système d'allocation dynamique.

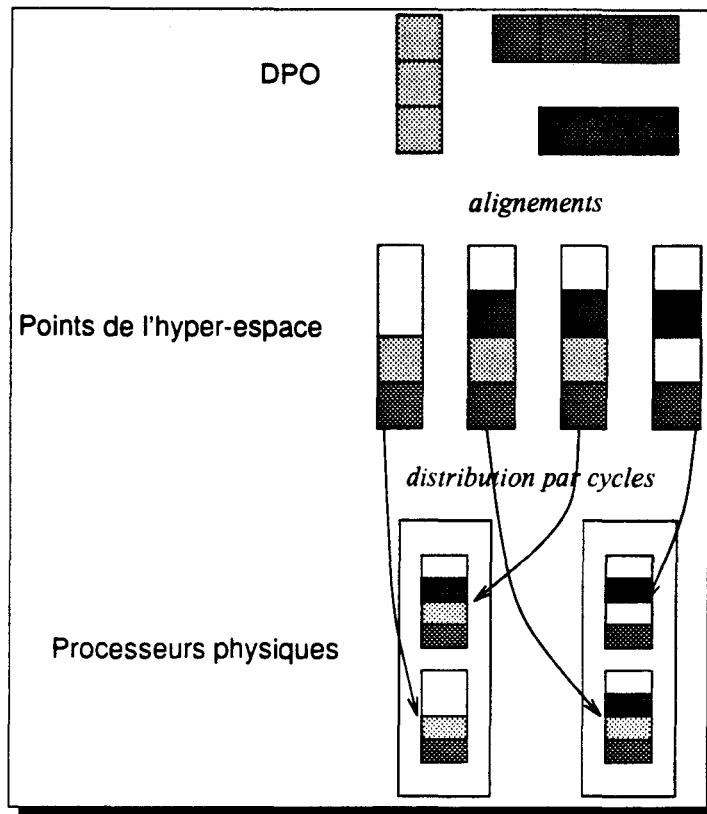


FIG. IV.5 - Allocation par les points

3.1.2 Calculs d'adresse

Une bonne propriété découle directement de l'allocation que l'on vient de décrire. Les calculs d'adresse permettant l'évaluation de toute expression (microscopique ou macroscopique) sont simplifiés et les accès mémoire vont s'en trouver rendus plus efficaces. En effet, une expression fait intervenir des DPO conformes et la phase préliminaire de calcul consiste à lire en mémoire le contenu des éléments des points actifs.

Or, dans une architecture à mémoire distribuée, l'adresse d'un élément est toujours composée d'un numéro de processeur et d'une adresse locale qui représente l'emplacement de l'élément dans la mémoire de ce processeur. Par conséquent, une référence à un élément induit un calcul de ces deux informations.

Lors de l'évaluation d'une expression faisant intervenir plusieurs objets, si le modèle de programmation ne permet pas au compilateur de considérer globalement l'expression, les accès aux données devront se faire indépendamment les uns des autres. Il est dans ce cas indispensable de répéter les calculs d'adresse autant de fois qu'il y a de références à des objets

différents dans l'expression, comme c'est le cas dans les langages existants, en particulier pour tous les langages basés sur Fortran [HKT92].

Avec l'allocation par points qui découle du modèle HELP, il est maintenant possible de factoriser une partie de ces calculs. Un élément est adressé par un déplacement constant⁸ à partir de la base du point dans lequel il est alloué. Tous les calculs d'adresse d'éléments de DPO sur un point se font par calcul de l'adresse de base de ce point dans la mémoire du processeur physique sur lequel il est projeté. Il ne reste plus, ensuite, qu'à décaler l'accès mémoire pour chaque DPO, en fonction de leur adresse locale au point.

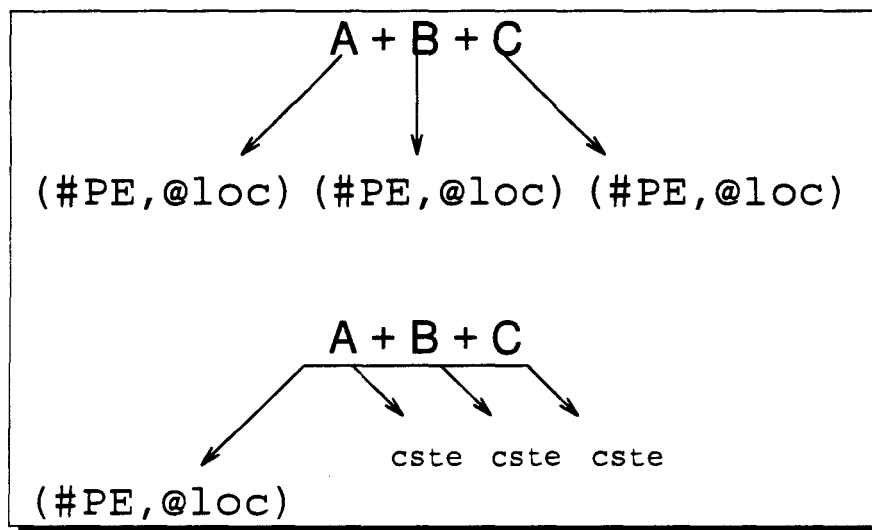


FIG. IV.6 - La factorisation du calcul d'adresse

La fonction de distribution n'est ainsi évaluée qu'une seule fois, quel que soit le nombre d'opérandes d'une expression microscopique (cf. figure IV.6). Nous montrerons l'intérêt de cette factorisation sur les mesures de temps d'exécution.

3.1.3 Allocation dynamique

L'allocation effective des éléments de DPO sur les points de l'hyper-espace engendre une gestion dynamique de la mémoire de ces points. Le fait d'avoir associé à un DPO une adresse unique pour tous les éléments ne doit pas entraîner l'allocation d'un DPO à réserver un emplacement d'adresse constante pour tous les points de l'hyper-espace. Cette solution reviendrait à allouer potentiellement un DPO sur tout l'hyper-espace, quelle que soit sa géométrie. On

8. « constant » dans le sens uniforme d'un point à l'autre, mais ce déplacement n'est pas évaluable à la compilation du fait de la dynamique des objets.

imagine bien que le taux de remplissage mémoire⁹ pour un problème donné serait inévitablement très bas dans la plupart des cas, ce qui empêcherait une bonne exploitation des capacités de la machine cible.

Nous devons donc gérer l'allocation mémoire de façon à ce que deux DPO qui ont une intersection géométrique vide puissent se partager la même adresse locale. Pour ce faire, nous avons écrit un algorithme d'allocation dynamique qui a été intégré dans le compilateur.

L'algorithme d'allocation mémoire dynamique Au moment de satisfaire un requête de création de DPO, le code doit évaluer le recouvrement du domaine de requête avec tous les domaines des DPO présents en mémoire. Or, le recouvrement éventuel de deux domaines ne peut être calculé qu'à partir de leur géométrie dans l'hyper-espace. Ce calcul étant identique pour tous les DPO alloués à l'instant de la requête, nous avons naturellement pensé qu'un algorithme à parallélisme de données répondrait parfaitement à ce calcul.

En entrée de l'algorithme, en fonction des DPO présents en mémoire, nous devons satisfaire une requête d'allocation dynamique composée d'un domaine et d'une taille des éléments du DPO à allouer. Le calcul consiste à trouver une adresse locale aux points, libre pour le domaine de requête, et suffisamment « large » pour accueillir le DPO à allouer. L'algorithme que nous avons défini consiste à opérer en parallèle le test de recouvrement des DPO par rapport au domaine de requête, puis à partir des résultats répartis calculer l'adresse libre pour le domaine de requête. Détaillons les étapes successives de cet algorithme (le lecteur peut de reporter à l'exemple qui suit ces explications) :

Structure de données Un processeur physique est associé à chaque DPO présents dans l'hyper-espace à l'instant de la requête d'allocation. Sur celui-ci, nous avons alloué la partie distribuée de l'entête. On y trouve la géométrie du DPO (intervalles pour chaque dimension du domaine d'allocation du DPO), l'emplacement mémoire (adresse constante à partir de la base des points) et la taille d'un élément¹⁰.

Calcul de recouvrement En parallèle sur les processeurs concernés, le calcul de recouvrement est effectué. Il consiste à comparer les bornes de l'intervalle d'allocation des DPO présents avec les bornes de l'intervalle de requête. La visibilité des DPO dans le programme source entraîne une gestion en pile de l'activité des processeurs pour cette phase. L'allocation d'un nouveau DPO augmente le numéro du dernier PE actif, tandis que la libération décrémente

9. rapport entre la taille mémoire utile et la taille mémoire réservée.

10. La taille des éléments pourrait ne pas apparaître dans l'entête, c'est une information statique.

ce numéro. Par conséquent, tous les processeurs de numéro inférieur à cette limite sont activés pour cette phase de calculs. À la fin de cette phase, nous savons distinguer les processeurs qui sont associés à des DPO qui ont un domaine d'allocation non-disjoint du domaine de requête.

Cette étape de l'algorithme est la seule étape de complexité dépendante des données du programme : elle est fonction du nombre de dimensions de l'hyper-espace car l'inclusion d'un point dans un domaine se fait par comparaison, pour chaque dimension de l'hyper-espace, des coordonnées du point et de l'intervalle du domaine.

Projection des recouvrants Nous voulons créer une image de la mémoire libre des points du domaine de requête. Là aussi, nous allons utiliser le réseau de processeurs physiques pour supporter la distribution de cette image. Chaque PE va représenter une partie de la mémoire des points du domaine de requête : une valeur négative indiquera que l'adresse est occupée, une valeur positive ou nulle indiquera que l'adresse est libre. Par défaut, nous considérons ces points comme totalement libres (l'image mémoire est initialisée à 1). Les deux étapes qui viennent ont pour but d'empêcher l'allocation sur une adresse déjà occupée. Pour ce faire, chaque processeur dont le calcul précédent a donné la valeur *vrai* (i.e. chaque DPO qui recouvre totalement ou partiellement le domaine de requête) envoie un message vers le processeur de numéro égal à l'adresse locale de son DPO associé. Ce message contient la taille du DPO associé à l'émetteur, multipliée par -1 (dans l'exemple, le dpo g est recouvrant ; son entête est présente sur le processeur 5 et contient l'adresse locale : 8 . Le processeur 5 envoie donc un message au processeur 8).

Cette étape est exécuté en un temps indépendant des données. Elle est uniquement fonction de la machine cible. Sur la MasPar, cette étape est opérée par une opération de routeur global. Son temps d'exécution effectif est inférieur à celui de la première étape.

Création de l'image mémoire Une opération de *parallel prefix* est déclenchée sur la machine physique, en considérant des segments d'activité dont les frontières sont les processeurs qui ont reçu une valeur à l'étape précédente. On obtient alors l'image mémoire représentant l'occupation des adresses mémoire pour le domaine de requête. Toute adresse mémoire libre pour le domaine de requête considéré apparaît dans cette image sous la forme d'un résultat positif sur le PE de même numéro (le nombre calculé représente la largeur d'allocation possible « sur la gauche » de chaque PE).

Là aussi, cette étape est de complexité indépendante des données. Elle s'exécute en un temps proportionnel au logarithme du nombre de processeurs, inférieur aussi à la première étape.

Sélection de l'emplacement Il reste enfin à sélectionner l'adresse mémoire dans l'image calculée. Pour ce faire, on sélectionne un des segments de l'image suffisamment large pour recevoir les éléments du DPO de requête. Ce segment est choisi en sélectionnant la valeur la moins élevée du segment (« le plus à droite ») qui est immédiatement supérieure à la taille des éléments du DPO à allouer moins 1¹¹. La valeur finale est égale au numéro de ce processeur auquel on soustrait la valeur de son élément de l'image mémoire. L'adresse de base du DPO de requête est ainsi obtenue.

Là encore, la complexité de cette étape est moins importante que la première étape : elle s'exécute aussi en un temps logarithmique par rapport au nombre de processeurs.

En terme de complexité globale, on peut remarquer que la première étape est seule dépendante du programme : elle est linéaire par rapport au nombre de dimensions de l'hyper-espace.

Prenons un exemple de l'exécution de l'algorithme : un objet, dont les éléments ont une taille de deux octets, doit être alloué (par sa déclaration ou par une migration dynamique) sur les points numérotés de 2 à 4 d'un hyper-espace linéaire qui contient déjà 7 objets (voir figure IV.7).

Le calcul de recouvrement donne un résultat positif pour les objets **a**, **d**, **f** et **g**. Les processeurs qui sont associés à ces objets communiquent avec les processeurs dont le numéro est égal à l'adresse d'allocation de ces DPO, en leur envoyant la taille des éléments déjà en mémoire. L'opération de *parallel prefix* est déclenchée sur les 4 segments et la carte mémoire est inspectée pour trouver le plus petit intervalle libre. Enfin, l'algorithme effectue la soustraction de la taille au numéro du processeur sélectionné pour donner l'adresse finale, ici 6.

11. suivant l'algorithme - *Best-Fit* -.

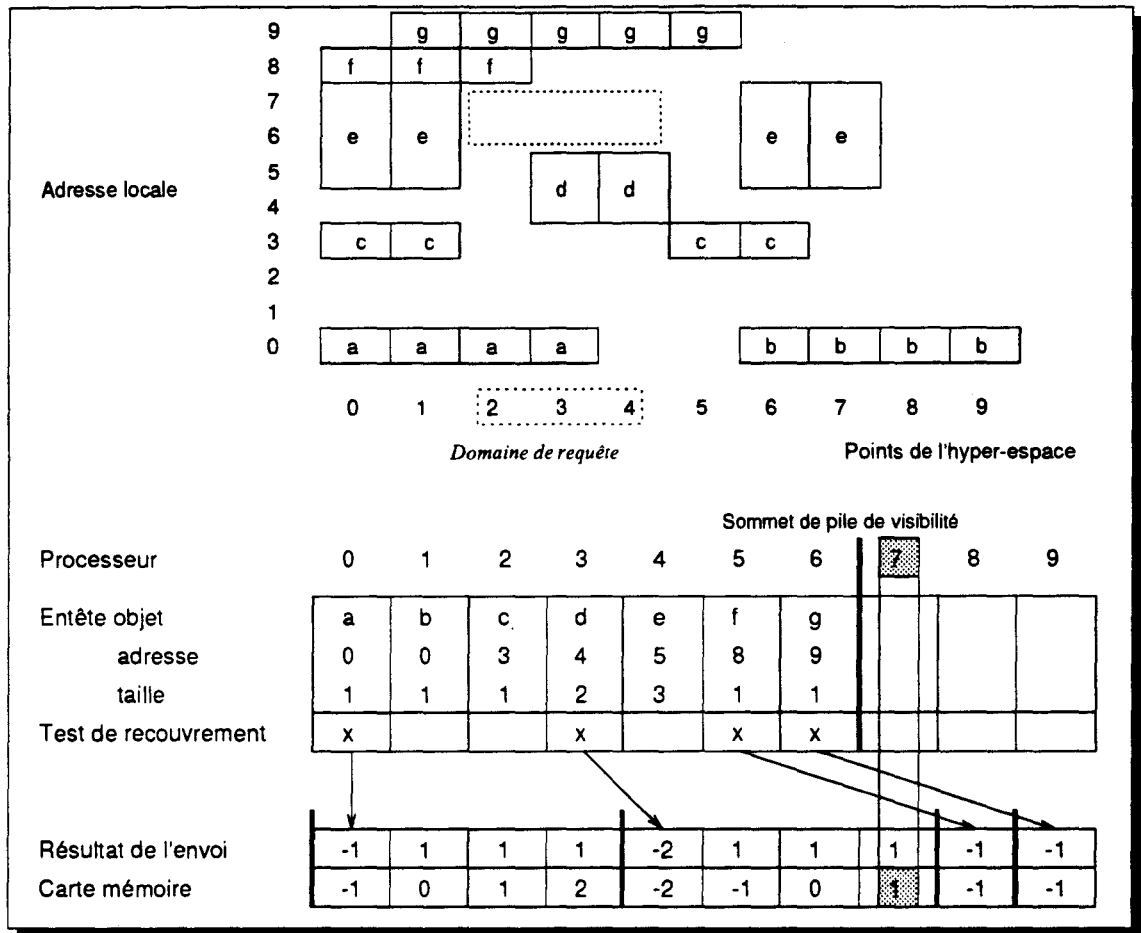


FIG. IV.7 - L'allocation avec recouvrement

Remarquons que la machine MasPar est tout-à-fait bien équilibrée pour cet algorithme :

- l'allocation d'un entête géométrique de DPO par processeur élémentaire permet la gestion de plus de 16000 objets simultanément alloués. Une virtualisation n'est donc (*a priori* !) pas nécessaire ;
- la mémoire maximale étant de 64 Koctets par processeur, il est possible de créer directement une image mémoire dont la granularité est de 4 octets, soit l'alignement courant. En effet, avec 16 k PE, chaque PE peut recevoir une tranche de l'image mémoire correspondante à 4 octets.

3.1.4 Allocation des points

Les accès mémoire de notre architecture cible ne comportent pas de mécanismes de cache. Les temps d'accès à des adresses contiguës ou non contiguës sont égaux. Par contre, sur d'autres architectures, en particulier sur les machines asynchrones, une exploitation rationnelle de la puissance potentielle de calcul ne peut être envisagée qu'à la condition d'utiliser effectivement les mémoires caches, si elles existent.

On doit donc s'efforcer de proposer une allocation mémoire qui permette des accès contigus. Pour notre solution exposée au paragraphe précédent, deux éléments d'un DPO voisins au niveau de l'hyper-espace ne sont pas voisins en mémoire. Dès lors, la boucle de virtualisation va entraîner un rechargement des caches à chaque itération.

Une solution est néanmoins envisageable pour pallier cet éclatement des données. Elle consiste à garder un système d'allocation dans lequel l'allocation des objets se fait toujours par rapport aux points de leur hyper-espace de déclaration, mais la projection des points sur les processeurs physiques se fait ensuite par intercalage des adresses.

Les éléments d'un même objet se retrouvent ainsi alloués sur des adresses consécutives (cf. figure IV.8). Ce système ne change rien quant à l'algorithme que l'on vient de décrire ni aux autres mécanismes mis en jeu.

Le calcul d'adresse d'un élément s'en trouve modifié suivant la projection choisie (voir la figure IV.8 qui fait suite à la figure précédente):

Avec entrelacement L'adresse d'un élément est obtenue à partir de l'adresse de base des points par l'opération: `adresse_locale * vp_ratio + adresse_du_point` en notant `vp_ratio` le nombre de points sur chaque processeur.

Sans entrelacement Le calcul est réalisé à partir de l'adresse de base du point avec un déplacement constant suivant l'adresse du DPO:
`adresse_locale + adresse_du_point * taille_des_points.`

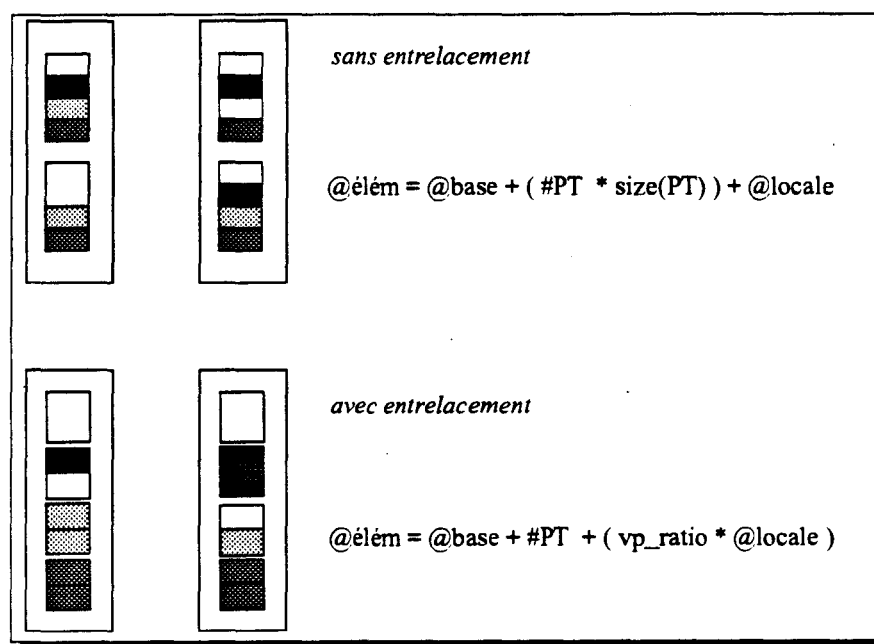


FIG. IV.8 - Deux alternatives pour l'allocation des points

Nous n'avons pas testé la seconde solution pour le compilateur C-HELP du fait de l'absence de différence avec la première solution proposée, pour notre machine. Par contre, sur une architecture différente, qui privilégierait les accès mémoire d'adresses contiguës, se peut-il que la meilleure solution soit trouvée avec l'éclatement des points?

Sur cette question interviennent les caractéristiques propres de la machine et du programme. En effet, si une même expression fait intervenir de nombreux DPO, il est possible que le cache des processeurs ne puisse pas accueillir plusieurs jeux de valeurs¹². Dans ce cas, pendant l'exécution d'une seule itération, le chargement des valeurs du second opérande écraserait les éléments du premier opérande nécessaire à l'évaluation future de l'expression sur les autres points actifs. Pour rendre efficace le cache il est alors plus intéressant de favoriser les accès contigus pour les différents éléments d'un point plutôt que pour les éléments d'un même DPO sur plusieurs points. Par conséquent, il est préférable d'adopter la première méthode d'allocation des points sur les processeurs physiques.

Il y a donc un compromis à effectuer. Comme nous venons de le dire, le nombre d'opérandes intervenant dans une expression est directement influent sur l'exploitation (ou l'impossibilité d'exploitation) du cache. Or, la méthode d'allocation des points sur les processeurs ne peut être que statique : on ne conçoit pas un changement complet d'allocation pour l'évaluation d'une expression puis un rétablissement dans l'autre configuration pour l'expression

12. Un jeu de valeurs est ici l'ensemble des éléments qui interviennent dans l'évaluation de l'expression pour un processeur donné.

suyvante. Le choix impose donc une configuration constante pour le déroulement complet du programme.

Cette question reste sans réponse formelle car fortement dépendante de critères matériels et logiciels (taille du cache, nombre moyen de références dans une expression) que nous ne pouvons pas déterminer pour un cas général. Nous pouvons tout de même penser que la taille des caches couramment utilisés permettent d'effectuer une lecture de plusieurs jeux de données, les expressions couramment écrites ne comportant guère plus de quelques références.

3.2 Boucle de virtualisation

La virtualisation a pour effet d'attribuer plusieurs points à un même processeur physique. La boucle de virtualisation consiste à balayer les points d'un même processeur pour y effectuer successivement les traitements.

3.2.1 Notion géométrique d'activité

Toute activité est déterminée au niveau de l'hyper-espace par les constructeurs du langage. Le domaine d'activité est toujours issu du domaine d'allocation d'un DPO, aussi bien pour l'application d'un traitement macroscopique que microscopique. Du fait de la dynamique des objets, et de l'expression des migrations au niveau du référentiel, le compilateur doit gérer l'activité au niveau même de l'hyper-espace, en balayant successivement tous les points du domaine de conformité.

Dans un contexte séquentiel, la boucle de virtualisation s'exprimerait par l'écriture de boucles imbriquées dont chaque indice correspondrait à un intervalle de coordonnées du référentiel. Dans le cadre du parallélisme de données et plus particulièrement du code généré par notre compilateur, il est nécessaire de répartir ce traitement sur les processeurs physiques qui détiennent les points actifs, suivant la distribution des points effectuée en fonction des informations liées à l'hyper-espace.

C'est ainsi que le compilateur génère le code qui active les processeurs en fonction du domaine actif et de la distribution des points sur ces processeurs. Pour chaque processeur, une boucle parcourt les points détenus, indépendamment des numéros locaux de chacun de ces points sur le processeur.

3.2.2 Balayage irrégulier

Les directives de projection associées à l'hyper-espace introduisent une correspondance entre un n-uplet (coordonnées dans l'hyper-espace) et un couple (numéro de processeur physique, numéro du point dans la mémoire du processeur). Une fonction qui permet l'évaluation de cette correspondance statique est générée lors de la compilation ; et fait partie de l'entête associé à chaque hyper-espace.

Les points ainsi répartis sur les processeurs constituent des couches. On peut décrire l'activité globale d'un traitement par une matrice booléenne rectangulaire A indicée par le numéro de processeur physique et par le numéro de point sur ce processeur physique. Soient \mathcal{PP} l'intervalle des numéros de processeurs physiques et \mathcal{PT} l'intervalle des numéros de points sur les processeurs (si le nombre de points diffère suivant les processeurs, on prend le maximum de ces nombres).

Il existe alors deux solutions pour activer successivement les points de chaque processeur :

- on garde une vision par couches. Chaque couche est successivement accédée et seuls les processeurs dont le point associé à la couche est actif effectuent le traitement. Le nombre d'itérations est alors borné par le nombre de couches qui possèdent au moins un point actif, on notera Reg ce nombre ;

$$Reg = Card(\{pt \in \mathcal{PT} \mid \exists pp \in \mathcal{PP} \text{ tq } A(pp, pt) = 1\})$$

- on considère l'activité au niveau des points de chaque processeur et on instaure un balayage des points particulier pour chacun des processeurs. Le nombre d'itérations est borné par le nombre maximal de points actifs pour un processeur, on notera $Irreg$ ce nombre.

$$Irreg = \max_{pp \in \mathcal{PP}} \left(\sum_{pt \in \mathcal{PT}} A(pp, pt) \right)$$

On démontre par l'absurde que $Irreg$ est toujours inférieur ou égal à Reg . D'après la définition de $Irreg$, on note pp_{max} le processeur physique qui détient le plus de points actifs et n_{max} ce nombre de points actifs :

$$\exists pp_{max} \in \mathcal{PP} \text{ tq } Irreg = \sum_{pt \in \mathcal{PT}} A(pp_{max}, pt) = n_{max}$$

L'hypothèse $Irreg > Reg$ peut se réécrire en $n_{max} > Reg$, or :

$$\exists (pt_1, pt_2, \dots, pt_{pp_{max}}) \in \mathcal{PT}^{n_{max}} \text{ tq } \forall i \in [1..n_{max}] A(pp_{max}, pt_i) = 1$$

On a donc l'inclusion suivante :

$$\{pt \in \mathcal{PT} \mid A(pp_{max}, pt) = 1\} \subset \{pt \in \mathcal{PT} \mid \exists pp \in \mathcal{PP} \text{ tq } A(pp, pt) = 1\}$$

d'où :

$$\begin{aligned} Reg &= Card(\{pt \in \mathcal{PT} \mid \exists pp \in \mathcal{PP} \text{ tq } A(pp, pt) = 1\}) \\ &\geq Card(\{pt \in \mathcal{PT} \mid A(pp_{max}, pt) = 1\}) \\ &\geq pp_{max} \end{aligned}$$

ce qui contredit l'hypothèse de départ. On a donc toujours $Irreg \leq Reg$. L'association de l'activité par points, plutôt que par couches, permet donc de réduire le nombre d'itérations nécessaires. On peut imaginer que cette réduction va abaisser le coût total du balayage. Pour cette raison le compilateur C-HELP intègre cette gestion irrégulière de la boucle de virtualisation. Comme nous l'avons défini, ce système consiste à évaluer localement à un processeur physique l'activité des points. Pour ce faire, la géométrie du domaine de conformité de l'expression courante est évaluée sur le processeur, en fonction de la distribution. Ainsi, le compilateur génère, pour chaque processeur physique un balayage qui n'itère que sur les points appartenants au domaine de conformité courant détenus par ce processeur.

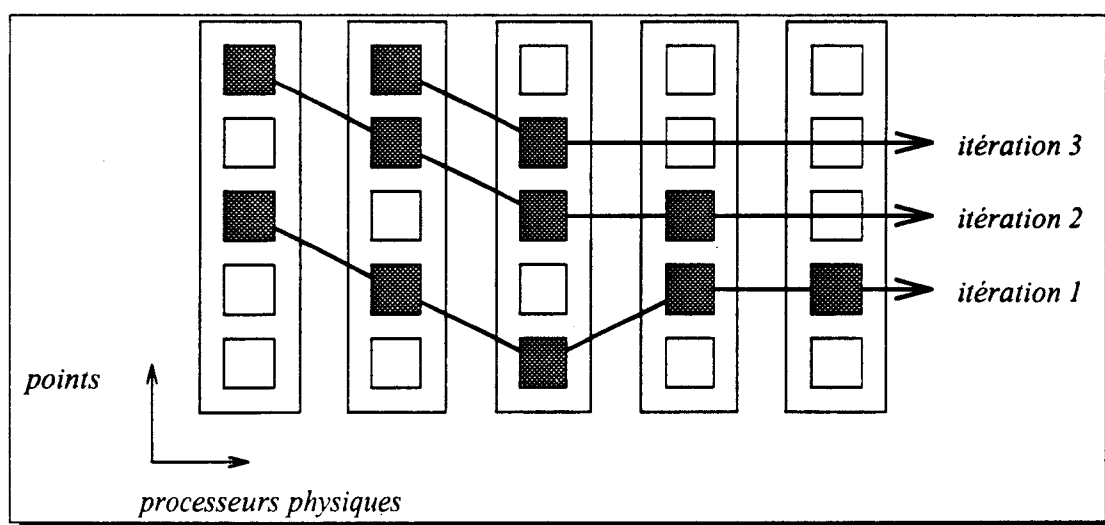


FIG. IV.9 - La boucle irrégulière de virtualisation

Comme nous l'avons prouvé formellement, cette introduction de l'irrégularité dans la boucle de virtualisation a pour effet de réduire le nombre d'itérations nécessaire à un traitement data-parallèle. Cette réduction sera d'autant plus significative que l'activité moyenne d'une couche sera basse [KF93], ce qui permet un regroupement plus efficace des activités de processeurs physiques. L'exemple montré dans la figure IV.9 illustre que cette optimisation permet des gains très sensibles dans certaines configurations (ici, 3 itérations suffisent au lieu de 5).

Pour pouvoir introduire cette optimisation dans le code généré, il est indispensable que l'architecture de la machine permette un adressage irrégulier suivant les processeurs (deux processeurs accèdent simultanément à deux adresses différentes de leur mémoire). C'est le cas de la MasPar, et du langage intermédiaire qui propose les variables de type `plural int * plural` qui permettent une utilisation irrégulière des pointeurs parallèles. Toutefois, ces accès mémoire sont plus coûteux¹³, ce qui handicape cette solution.

Nous montrerons dans la section suivante l'efficacité de cette solution par des mesures de temps d'exécution.

3.3 Compilation des appels macroscopiques

La boucle de virtualisation que nous venons de décrire est implémentée en langage intermédiaire par une boucle sur chaque processeur. Les variables d'indexation sont directement reprises des coordonnées du point à traiter. Ainsi, avant le traitement proprement dit, on connaît les coordonnées dans toutes les dimensions du point qui va subir le traitement.

La modélisation géométrique impose de calculer les coordonnées du point cible par l'évaluation des paramètres d'appel des opérations macroscopiques. Puisque les coordonnées de départ sont connues grâce aux variables d'itération de la boucle de virtualisation, le calcul de ces coordonnées se fait directement à partir de ces variables. Les opérations macroscopiques sont traduites sous forme de fonctions affines évaluées sur les coordonnées sources. La composition d'appels macroscopiques est traduite par la composition de ces fonctions affines. Par conséquent, il est possible d'enchaîner les appels successifs aux primitives macroscopiques, sans effectuer les migrations intermédiaires. À chaque point source, on est capable de donner l'ensemble des points cibles.

À partir des coordonnées obtenues pour chaque point cible, le compilateur appelle la fonction de distribution pour convertir ces coordonnées vers le couple (PE,Point) correspondant.

13. Avant d'accéder à l'élément, chaque processeur doit d'abord lire dans sa propre mémoire l'adresse de cet élément. La MasPar partageant les bancs mémoire pour une grappe de processeurs, cet accès préalable à la mémoire parallèle est pénalisant par rapport à une adresse uniforme qui peut être gérée sur l'ACU. Entre les deux modes, les temps d'accès varient d'un facteur d'environ 3,5.

C'est pour appeler cette fonction que la structure correspondante à l'hyper-espace de migration comporte un pointeur vers cette fonction. On génère ensuite un appel à la bibliothèque de communications MPL¹⁴ qui se charge de la communication et du rangement de la valeur dans la mémoire du processeur distant.

Cet enchaînement des appels successifs est cohérente avec la possibilité accordée au programmeur d'utiliser dans son programme des coordonnées dépassant l'espace du référentiel, entre deux appels. Seul le résultat final de l'évaluation des coordonnées cibles doit vérifier la cohérence géométrique.

4 Étude des performances

CETTE section va nous permettre d'illustrer par des mesures de temps d'exécution de programmes triviaux, la pertinence des propositions que nous venons d'exposer.

Dans un premier temps, nous présenterons le langage MP-FORTRAN qui nous sert de référence pour les performances. Le choix de comparer les performances du compilateur C-HELP sur la base de MP-FORTRAN est motivé par la volonté de se référer à un langage largement diffusé. De plus, la comparaison ne pouvait se faire que sur un langage virtuel, du fait des techniques de compilation à comparer.

Nous évaluerons ensuite les gains obtenus d'une part grâce à la factorisation du calcul d'adresses puis d'autre part, par l'irrégularité de la boucle de virtualisation.

4.1 Avertissement

Les tests que nous allons exposer dans cette section ont pour objet la comparaison entre les performances du code généré à partir de C-HELP par notre compilateur et les performances obtenues avec MP-FORTRAN.

Nous nous devons d'avertir le lecteur que les programmes Fortran étudiés ne sont en aucun cas le reflet de la programmation d'un problème général. Nous avons voulu tester l'efficacité des particularités du compilateur C-HELP, en s'abstrayant des autres concepts. Pour ce faire, les codes Fortran ont été développés de façon à retrouver un fonctionnement proche de celui de notre modèle d'exécution, en particulier vis-à-vis de la dynamique des objets. Alors que l'on aurait pu écrire des codes plus efficaces en MP-FORTRAN, nous avons choisi d'introduire la dynamique par appel d'une fonction. Les performances s'en trouvent grandement dégradées.

Le modèle HELP repose sur la dynamique des objets, en taille et en position dans l'hyper-

14. fonction `sp_rsend` ou `sendwith...` pour les réductions associatives

espace. Cette caractéristique a donc été retranscrite au niveau du langage et du code généré. La prise en compte de données parallèles statiques (mot-clef **steady**) n'a pas été intégrée dans le compilateur, car considérée comme faisant partie des optimisations de code, non prioritaires vis-à-vis de notre volonté de s'attacher aux concepts spécifiques du modèle HELP. Il est bien évident que malgré ce caractère optionnel, cette possibilité d'utiliser des données statiques est primordiale pour l'obtention de bonnes performances. C'est seulement à partir de cet instant que le code générée pour une application réelle pourrait être concurrentiel par rapport à Fortran.

4.2 Un langage de comparaison : MP-FORTRAN

MP-FORTRAN[Mas91a, Mar93b] est issu de Fortran 77 dans la lignée de Fortran 90. En particulier, il propose des constructions spécifiques dédiées au parallélisme de données. Le compilateur génère du code exécutable pour la MasPar.

4.2.1 Un langage data-parallèle

Le modèle de programmation MP-FORTRAN est un exemple du modèle de programmation par les indices, comme nous l'avons présenté pour HPF. L'objet parallèle de base est le tableau multi-dimensionnel classique. Le parallélisme de données peut s'exprimer de différentes façons :

Description par triplets La description d'un tableau à l'aide de triplets, en partie gauche ou partie droite de l'affectation, permet d'appliquer un traitement data-parallèle sur les tableaux, en ne considérant qu'une partie régulière des éléments. La détermination des éléments intervenant dans l'expression est donc effectuée par un traitement sur les indices de ces triplets.

Il existe un triplet pour chaque dimension de l'objet, chacun d'entre eux est composé d'une borne inférieure, d'une borne supérieure et d'un pas. Par défaut, le pas prend la valeur 1, et la notation ':' est un résumé syntaxique permettant de décrire l'intervalle d'allocation d'un tableau sur la dimension donnée.

$$C(i,J) = \text{SUM} (A(1:100,J) * B(I,1:100))$$

$$C(I,J) = \text{SUM} (A(:,J) * B(I,:))$$

Description par un vecteur Cette possibilité offerte par MP-FORTRAN est directement issue du légendaire « *gather/scatter* ». Le programmeur accède aux éléments d'un objet parallèle en fonction des valeurs d'un autre objet parallèle. Ces accès sont ainsi totalement dynamiques et ne peuvent pas être modélisés formellement. En particulier,

l'usage de telles constructions est toujours accompagné de mise en garde à destination du programmeur : on ne peut en générale prédire aucun comportement global et les communications engendrées par de telles descriptions ne peuvent *a priori* être contrôlées finement par le programmeur. D'autres problèmes sont posés par cette construction, en particulier si un élément du vecteur est accédé plusieurs fois dans la même expression.

Instruction forall Comme HPF, MP-FORTRAN propose une instruction **forall** destinée à rénover le classique **do**. Les variables d'itérations forment un polyèdre dont chaque point représente une valeur à calculer. L'utilisation de cette instruction permet au programmeur de spécifier que le traitement qui suit peut (doit) être exécuté en parallèle.

Constructeur where La possibilité d'appliquer un masque sur certaines valeurs est donnée au programmeur par l'intermédiaire de l'instruction **where**. Cette instruction est limitée aux affectations de tableau.

La distribution des données est effectuée relativement à chaque tableau. En conséquence, il n'existe pas d'alignement entre tableau au niveau de la machine physique. Tout tableau est alloué à partir du processeur zéro. Les directives permettent un découpage par blocs ou un découpage cyclique.

4.2.2 Le compilateur

Le compilateur MP-FORTRAN intègre des techniques de compilation propres à la machine cible : outre la génération directe de l'assembleur [Mas90], la gestion des scalaires est reléguée sur la station frontale qui dispense une puissance supérieure à l'ACU. Cette optimisation est un exemple de l'utilisation de bibliothèques développées indépendamment du compilateur (et naturellement fortement optimisées). Nous nous plaçons donc face à un produit commercialisé qui a sûrement nécessité un gros effort de développement.

Il est important de remarquer la différence entre les deux stratégies de génération de code adoptées pour la compilation de l'instruction **forall** et de la description des vecteurs par des triplets. Il existe deux stratégies différentes du fait de l'adoption du modèle basé sur les indices [Ste93].

Description par triplets Pour ces constructions, le compilateur génère le code qui évaluera l'expression selon le modèle « *Owner Compute Rule* » (OCR). Le calcul d'une expression est effectué sur le processeur qui détient l'élément à affecter en sortie d'expression. Les valeurs servant à l'évaluation de la partie droite sont amenées par le compilateur, pour être consommées sur place¹⁵.

¹⁵. pas d'emballage, merci.

Instruction forall Le compilateur abandonne pour cette instruction le modèle OCR, au profit de la distribution d'itérations. Chaque portion de calcul correspondant à un élément de l'objet à calculer est projetée sur un processeur, suivant l'ordre d'énumération de la machine. Les données sont ainsi totalement redistribuées pour le temps de l'évaluation de l'expression. En fin de calcul, chaque résultat est ensuite remis en place par une autre phase de communications. Cette méthode de compilation permet d'obtenir un parallélisme maximal, mais nécessite généralement un grand nombre de communications. Pour devenir efficace, elle doit s'accompagner d'une politique de regroupement de boucles qui lui permette de minimiser le nombre des communications.

4.3 Allocation mémoire

L'algorithme que nous avons décrit ne donne pas l'allocation optimale des DPO à un instant donné. Pour ce faire, il eût fallu résoudre un problème qui est à l'évidence difficile (comparativement au problème du sac à dos, on sait qu'il comporte plus de contraintes¹⁶). Si une solution optimale était implantée, il se pourrait qu'une requête exige à un instant quelconque une complète réorganisation de la mémoire par une nouvelle allocation de chaque DPO alloué à l'instant de la requête, au prix d'une copie de tous ces DPO. On ne peut imaginer instaurer un tel fonctionnement.

Nous avons donné une heuristique qui autorise le partage des adresses locales quand plusieurs DPO ne se recouvrent pas. Notre algorithme possède la bonne propriété de s'exécuter en temps constant et raisonnable : la première étape, plus coûteuse que les suivantes, a une complexité linéaire par rapport au nombre de dimensions de l'hyper-espace ; tandis que les autres ont une complexité dépendante du logarithme du nombre de processeurs de la machine cible. Cet algorithme s'exécute en un temps indépendant de l'occupation courante de la mémoire. On peut tout de même envisager de ne pas appeler cet algorithme pour toute allocation mémoire. En effet, si très peu de DPO sont déjà alloués, la gestion des adresses libres pourrait se faire par un mécanisme centralisé ; un seuil d'occupation mémoire doit déterminer à partir de quel moment l'algorithme d'allocation mémoire dynamique doit être déclenché.

L'évaluation des performances de cette allocation dynamique en terme de taux d'occupation ne peut pas se faire dans un cas général car elles dépendent fortement de la géométrie des données manipulées par l'algorithme. On peut simplement évoquer les avantages de la distribution par intermédiaire de l'hyper-espace. Dans le cas d'une distribution directement liée aux objets et dans laquelle l'origine de chaque tableau est fixée à l'origine de la machine, comme le fait MP-FORTRAN, tout objet est alloué à l'origine de la grille de processeurs. La

16. Le problème du sac à dos est un problème NP-complet. Comparativement à cette base, nous savons que l'emplacement d'une « pièce » n'est pas librement choisie par l'algorithme car elle doit respecter une contrainte supplémentaire due au fait que l'objet doit être alloué sur des points bien déterminés. Cette contrainte ne semble pas contradictoire avec celles du sac à dos.

saturation mémoire est par conséquent atteinte sur le processeur 0 dans tous les cas. Par contre, avec une projection par points, les objets se trouvent alloués physiquement suivant les traitements qui vont être effectués. Il y a donc forcément un meilleur équilibrage de la charge globale.

4.4 Calculs d'adresses

Nous considérons une expression qui fait interagir plusieurs objets parallèles. Le nombre d'accès mémoire est égal au nombre de références aux objets. Pour nous placer dans des cas comparables, le programme Fortran est écrit avec l'intermédiaire d'un appel de fonction qui permet de mesurer le temps d'exécution pour des données allouées dynamiquement. L'utilisation de triplets pour la description permet d'éviter la redistribution dynamique des itérations.

L'expression considérée est composée de plusieurs références à des objets dont la taille est inférieure à la topologie cible, pour éviter de considérer plusieurs itérations de la boucle de virtualisation (cf. section suivante). On mesure le temps d'exécution pour une expression comportant un certains nombres de références à des objets différents. Comme nous l'avons présenté, le nombre d'évaluations de la distribution est forcément égale au nombre de références pour MP-FORTRAN tandis que C-HELP instaure la factorisation qui a pour effet de permettre un unique appel de cette fonction. (Voir les programmes en annexe A).

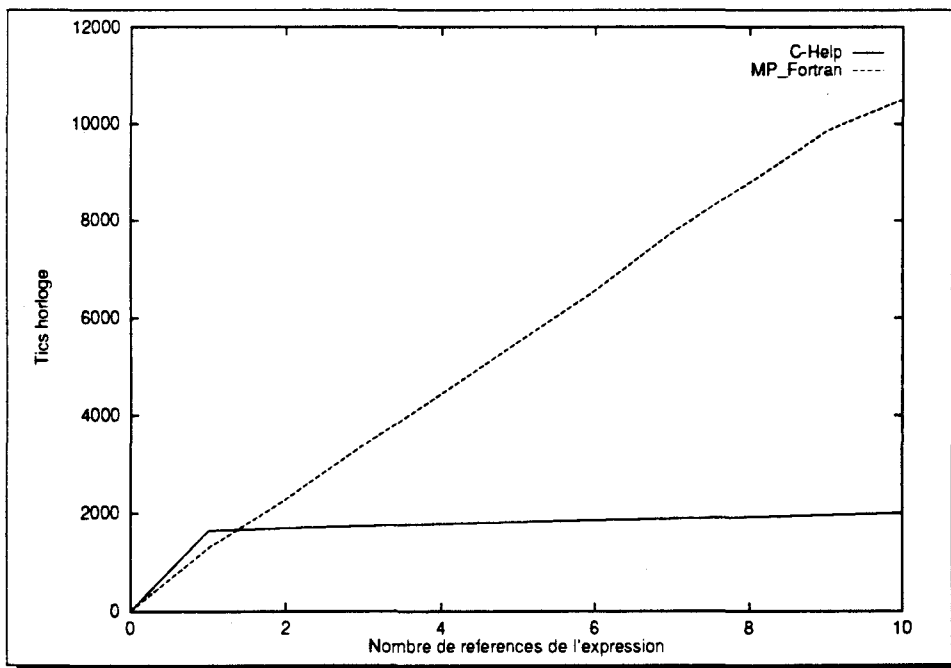


FIG. IV.10 - Efficacité du modèle basé sur le référentiel

La figure IV.10 présente les deux courbes obtenues. Comme nous l'avons exposé, le compilateur C-HELP génère une factorisation du calcul d'adresse qui permet de ne calculer qu'une seule fois l'adresse du point actif. La distribution des données n'est elle-même calculée qu'une fois en entrée de la boucle. Le temps d'exécution s'en trouve largement diminué. Après une première référence, les accès suivants ne nécessitent qu'un seul accès mémoire pour l'obtention de l'adresse de base du DPO, une addition pour le calcul d'adresse effective de l'élément à lire et de la lecture proprement dite. On obtient donc un temps linéairement croissant à partir de la seconde référence.

Pour le programme Fortran, la dynamicité introduite (certes, artificiellement) par l'appel de fonction empêche le compilateur de connaître, lors de la génération de code, la distribution de chaque donnée parallèle. C'est le modèle de programmation qui est ici très pénalisant. Deux tableaux Fortran peuvent être distribués de façon différentes (par blocs ou par cycles). Dès lors, le compilateur doit générer le code d'évaluation de cette distribution pour chaque référence à un objet. Le temps d'exécution de MP-FORTRAN est comparable à celui de C-HELP pour la première référence, mais la factorisation de C-HELP devient rentable dès la deuxième référence ; à partir de la deuxième référence, les deux courbes deviennent linéairement croissantes, mais la pente de C-HELP est nettement moins importante que celle de MP-FORTRAN.

4.5 Boucle de virtualisation

La gestion irrégulière de la boucle de virtualisation permet de réduire le nombre d'itérations. Nous montrons le gain obtenu avec la comparaison entre les temps d'exécution d'un cas trivial écrit en C-HELP et en MP-FORTRAN (cf. annexe A).

On considère un hyper-espace bidimensionnel de taille 300×300 dans lequel on traite un DPO de taille 50×50 , alloué successivement à différents emplacements de l'hyper-espace. La machine cible sur laquelle ont été exécutés les tests est une grille 128×128 . La virtualisation entraîne donc un pliage par couche de 128×128 . Au cours des migrations de l'objet considéré, les points actifs sont amenés à changer de couche. C'est ainsi que sur les frontières, le DPO est « à cheval » sur deux ou quatre couches. C'est dans ces cas que l'irrégularité devient efficace.

Pour se placer dans des conditions équivalentes et afin de ne considérer que la boucle de virtualisation, le programme Fortran est écrit de façon à traiter des objets dynamiques. On a donc procédé à un passage de la géométrie correspondante au domaine de conformité en paramètre. De plus, rappelons que le compilateur MP-FORTRAN redistribue dynamiquement les itérations pour l'instruction `forall`, perdant ainsi le modèle de compilation dit « *Owner-Compute Rule* ». Cette instruction n'a donc pas été utilisée pour éviter les communications qu'elle génère. L'utilisation de triplets pour décrire le sous-objet de taille 50×50 permet de

mesurer le temps d'exécution suivant le modèle identique à celui de C-HELP, mais avec une boucle de virtualisation régulière.

Précisons que les deux programmes opèrent la même distribution par défaut : les deux dimensions virtuelles sont projetées sur les deux dimensions physiques de la machine par intermédiaire d'un découpage en cycles.

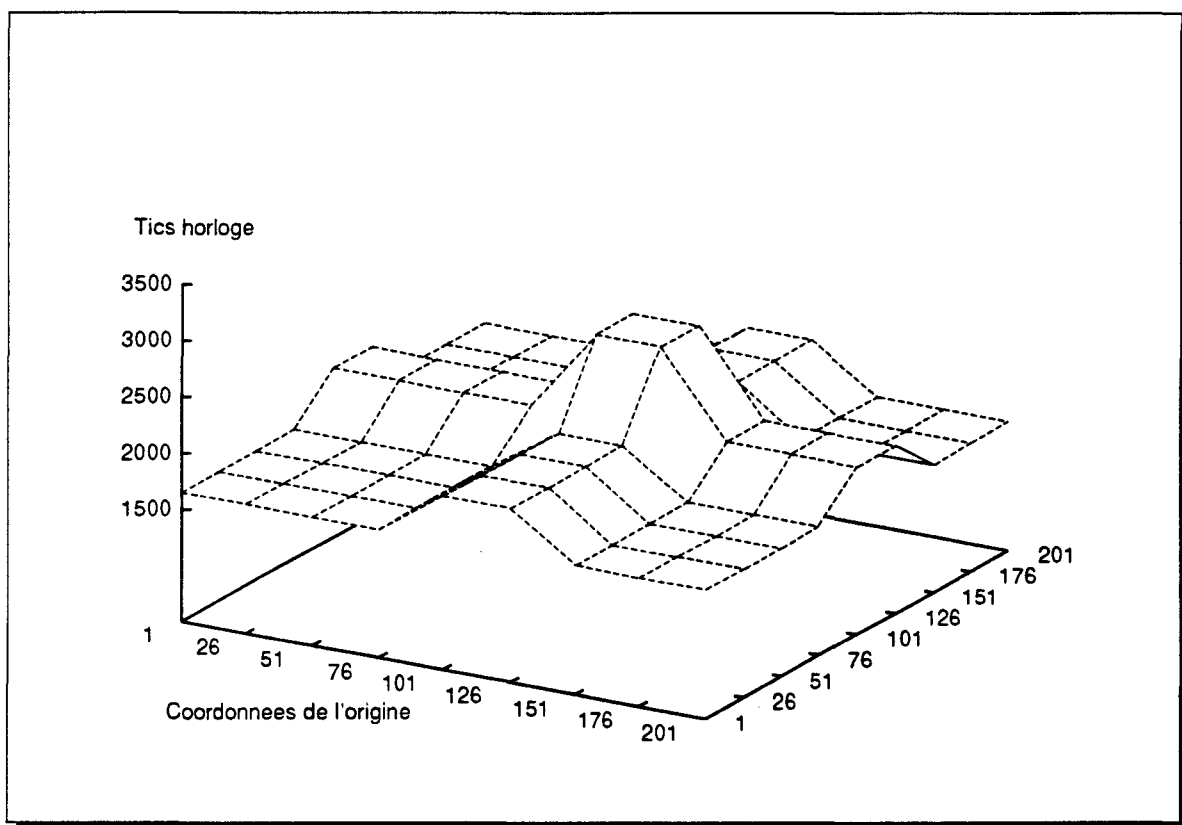


FIG. IV.11 - Performances de la boucle régulière de MP-FORTRAN

Pour Fortran (cf. figure IV.11), les résultats des tests s'interprètent par le fait que le compilateur génère un code de parcours régulier de la boucle de virtualisation. Quand le sous-objet considéré se trouve réparti sur plusieurs couches de l'objet global, plusieurs itérations sont alors nécessaires. Ce qui explique l'apparition sur le graphique de frontières correspondantes avec la topologie cible.

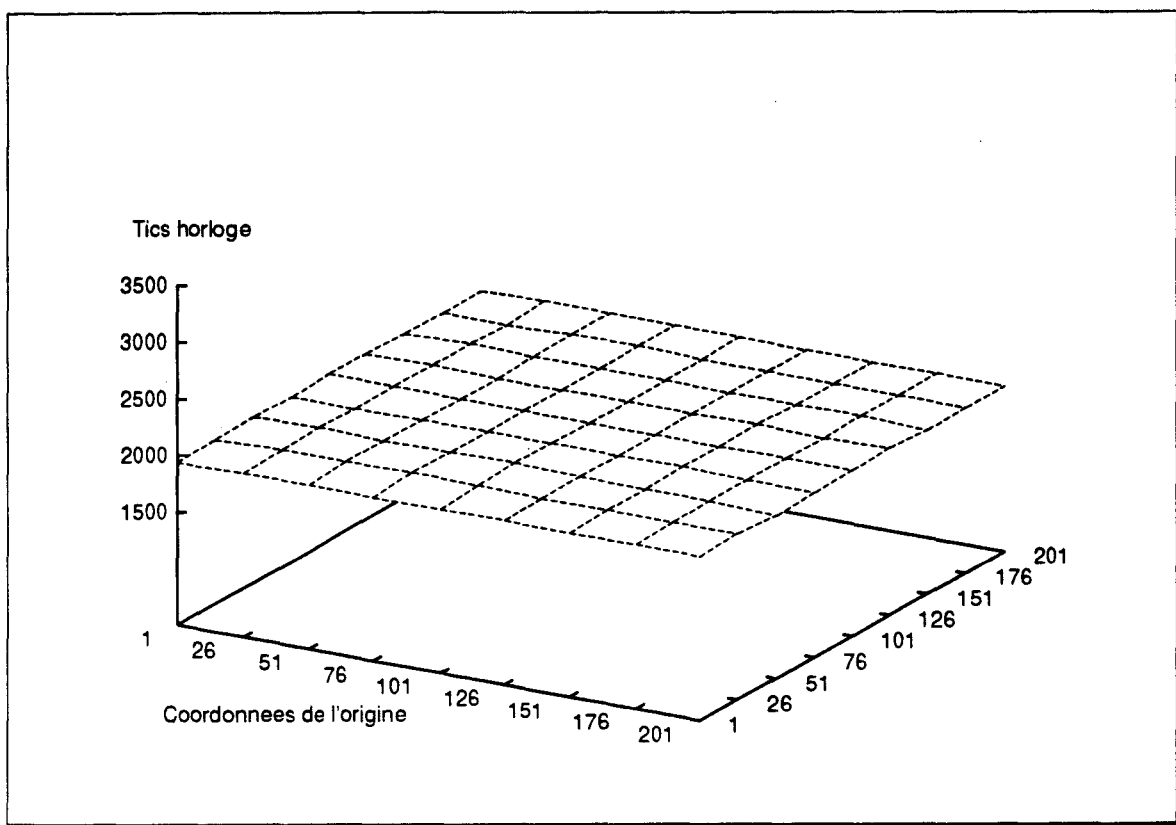


FIG. IV.12 - Performances de la boucle irrégulière de C-HELP

Pour C-HELP (cf. figure IV.12), la courbe montre une constance dans le temps d'exécution. Cela s'explique par la distribution adoptée qui entraîne l'allocation d'un élément par processeur. L'objet étant de taille plus petite que le réseau bidimensionnelle de la machine cible, il n'y a jamais deux points actifs par processeurs. Le point critique de ce programme (quand le DPO se retrouve « à cheval » sur les quatre premières couches) met en valeur les gains potentiels.

Pour étudier les temps d'exécution d'une unique itération de la boucle, nous pouvons comparer les valeurs minimales de ces deux courbes. Pour ces configurations où l'exécution s'opère en une itération car les données se trouvent sur la même couche, C-HELP est 20% plus lent que MP-FORTRAN. Il est important de noter que l'accès mémoire est plus coûteux dans le cas d'une boucle de virtualisation irrégulière. Pour la MasPar, cette solution impose d'effectuer des accès non-uniformes pour tous les processeurs. Des mesures de performances [Gav92] montrent que de tels accès sont 3,5 fois plus lents que les accès uniformes.

Sur machine MIMD, les accès mémoires indirects se font indépendamment sur chaque processeur, avec les mêmes performances lorsqu'ils accèdent tous à la même adresse ou lorsqu'ils adressent tous une adresse différente. On ne retrouvera donc plus ce facteur 3,5 pour cette

phase d'accès mémoire. Cet handicap ne sera plus présent sur des machines asynchrones, ce qui laisse augurer un gain encore plus important.

La figure IV.13 montre l'éclatement des quatre couches concernées durant l'allocation des points de l'hyper-espace sur les processeurs. Durant l'évaluation de l'expression, on peut visualiser l'activité des processeurs physiques par l'environnement de programmation MPPE [Chr90, BB92], et confirmer ainsi qu'une itération est suffisante, au lieu de quatre pour le compilateur Fortran.

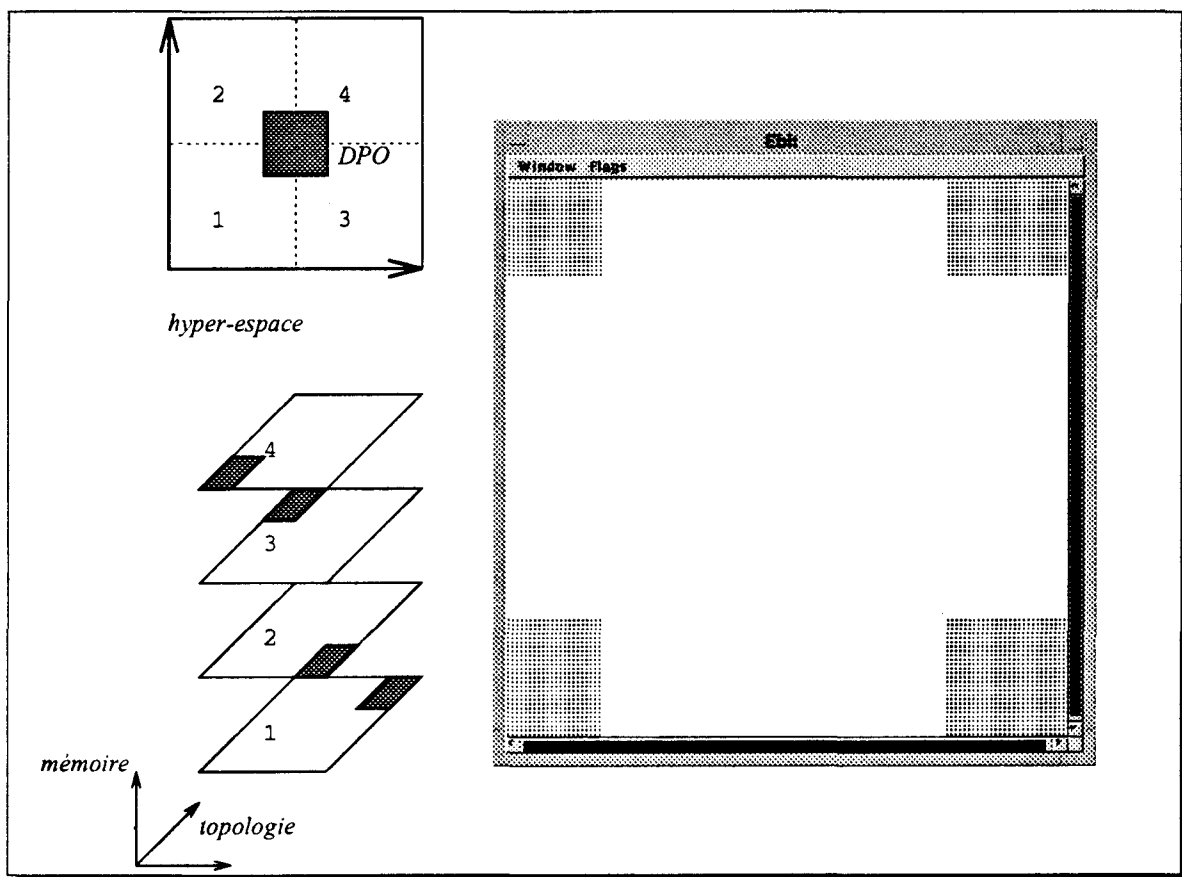


FIG. IV.13 - Une position critique, gain maximal de l'exemple

Ces résultats montrent le gain obtenu grâce à l'irrégularité. C'est avec l'irrégularité du balayage que l'on obtient la régularité des temps d'exécution suivant l'origine de l'objet. Le temps est maintenant proportionnel au nombre de points actifs par processeur physique, quelque soit leur position dans la mémoire des processeurs. Le placement des données est par conséquent rendu plus facile. En effet, un langage dont le compilateur ne possède pas cette propriété amène le programmeur à éviter les positions d'objets « à cheval » sur une frontière

de distribution, ce qui est fortement contraire à l'idée de virtualité qui devrait consister à manipuler les structures de données sans influence de critères matériels.

Remarquons que cette optimisation est d'autant plus importante que la taille des données est proche de la taille de la machine cible. Dans ce cas, pour des données de même taille, la fréquence d'activation de domaines « à cheval » sera plus élevée.

5 Conclusion

NOUS venons de décrire des techniques de compilation qui découlent directement du modèle HELP. La règle de conformité introduite dans le modèle nous permet d'exploiter pleinement le bon placement des données, par la définition d'une allocation mémoire en corrélation avec l'hyper-espace.

Le point, entité de base du référentiel, regroupe les données au niveau virtuel, et on retrouve cette conformité au niveau physique sur les processeurs. L'allocation des DPO est effectuée à l'intérieur de la mémoire de chaque point, suivant une adresse de base constante. La phase de calcul des adresses d'éléments qui interviennent dans une expression s'en trouve largement optimisée par une factorisation permettant l'évaluation unique de la fonction de distribution, quel que soit le nombre de références de l'expression à évaluer.

Le partage d'une même adresse locale pour plusieurs DPO qui ont une intersection vide au niveau de l'hyper-espace a été décrite. L'algorithme d'allocation dynamique est conçu en ce sens, nous avons naturellement utilisé le paradigme à parallélisme de données pour son intégration dans le code généré par le compilateur. Cet algorithme est exécuté en un temps indépendant de l'état de la mémoire au moment de la requête d'allocation mémoire.

L'irrégularité introduite dans la gestion de la boucle de virtualisation a permis de réduire le nombre d'itérations nécessaires au parcours complet du domaine de conformité. Le temps d'exécution n'est pas dépendant de la position du domaine de conformité, mais du nombre de points actifs par processeur physique. On obtient ainsi une plus grande abstraction de la topologie cible.

Tous ces résultats ont été illustrés par des mesures de temps d'exécution issus de notre atelier de compilation.

Chapitre V

Vers les structures irrégulières

IL EXISTE un grand nombre de domaines scientifiques dans lesquels l'algorithmique consiste à traiter des données de très grandes tailles. Issues de mesures d'expériences diverses, ces données sont souvent constituées en grande partie d'éléments nuls. Seul un certain pourcentage (généralement de l'ordre de quelques pour cent) de données sont non nuls. On parle alors de **structures de données creuses**, ou plus couramment de **calculs creux**.

Dans ce chapitre, nous présenterons la problématique du calcul creux et les outils qui existent aujourd'hui pour écrire des algorithmes. De la constatation du manque d'environnement permettant de s'abstraire du problème de la compression, nous déduisons la nécessité de fournir au programmeur un outil qui se charge automatiquement de cette phase de compression, lui laissant la possibilité d'ignorer totalement, ou presque, le fait qu'il manipule des structures creuses.

Pour cela, nous présenterons une façon d'intégrer la compression dans la compilation du langage C-HELP, tout en conservant inchangé le modèle de programmation géométrique.

Nous nous pencherons ensuite sur les conséquences de ce choix vis-à-vis de la compilation. Nous étudierons la faisabilité d'une compilation qui prend en charge la compression des données.

1 Le calcul « creux »

LA PROBLÉMATIQUE posée par ces applications est très simple :
- la mémoire de la machine n'est pas suffisante pour gérer la totalité des données. Les mécanismes d'exécution doivent être limités au traitement des éléments non-nuls des données de départ. Les éléments nuls ne doivent en aucun cas être associés à une quelconque structure de données, la mémoire devant être réservée uniquement aux éléments significatifs.

- le traitement de tous les éléments de la matrice est pénalisant par rapport au traitement des éléments significatifs uniquement. On gagne par conséquent du temps d'exécution en ignorant les éléments nuls. La définition du calcul creux peut découler de cette constatation : *un calcul est creux quand il est exécuté plus rapidement qu'en le considérant dense* [PSWF93].

Une solution consiste à proposer une allocation mémoire la moins grande possible tout en gardant la possibilité d'exploiter le parallélisme de la machine. Les importants problèmes d'algorithmique viennent du fait qu'un stockage « économique » des données rompt les fonctions d'accès aux éléments à partir du référentiel de programmation. La possibilité de calculer l'adresse d'un élément en fonction de sa position par rapport au référentiel n'existe plus, que ce soit dans la sémantique des indices ou basée sur le référentiel. En effet, l'adresse d'un élément dépendra du fait que les autres éléments de la matrice sont significatifs ou nuls. Le stockage est par conséquent globalement calculable, mais localement non calculable. Les mécanismes mis en place ont pour but de recréer, à l'aide de structures diverses, un référentiel d'accès aux éléments significatifs.

Dans cette étude, nous nous limiterons, comme c'est le cas dans les études du domaine, aux matrices bidimensionnelles. Les concepts présentés s'en trouveront plus clairs, mais une généralisation à des données d'ordres supérieurs peut être envisagée par les mêmes mécanismes.

1.1 Compression des données

L'occupation mémoire d'une matrice creuse ne doit pas dépendre de la taille de la matrice, mais doit être fonction du nombre d'éléments non-nuls de la matrice.

Par exemple, on s'interdit d'associer à une matrice creuse une matrice de même taille composée de booléens qui permettraient d'accéder directement à l'information d'existence d'un élément significatif ou nul. En effet, si on considère un faible pourcentage d'éléments non-nuls (par exemple 1%) d'une matrice de réels en double précision (comme c'est souvent le cas), le masque correspondant requerrait une plus grande occupation mémoire que les éléments significatifs de la matrice.

Tous les formats de compression des données que nous allons présenter dans cette section répondent au critère d'une allocation d'encombrement linéaire par rapport au nombre d'éléments non-nuls d'une matrice. Dans ces modèles de structures de données, on tolère une structure de taille correspondant à la racine carrée du nombre total d'éléments de la matrice de départ.

Les formats existants sont multiples, et souvent adoptés par un constructeur (ou une bibliothèque) précis. Nous avons sélectionné six formats qui semblent être les plus répandus [Saa90]. Ils ont tous en commun le fait de proposer une structure regroupant les données significatives, et d'autres structures permettant de calculer les coordonnées de ces éléments dans la matrice de départ. Nous n'allons pas considérer les formats qui extraient de la matrice de départ une structure de donnée directement issus du patron¹ des données. Par exemple, nous ne parlerons pas de formats de stockage de matrices « bande »² ou de matrices composées de blocs denses sur la diagonale. Nous considérons que les présuppositions faites sur ces matrices ne permettent pas de considérer ces matrices comme véritablement faisant partie du calcul creux en général.

1.1.1 Le format COO

Le format à coordonnées est certainement le plus simple de tous les formats de compression. À une matrice creuse de départ, on associe un vecteur qui ne contient que les éléments non-nuls. Ce vecteur obtenu est doublé par deux autres vecteurs de type entier dont chaque élément représente les coordonnées dans la matrice de départ (cf. figure V.1).

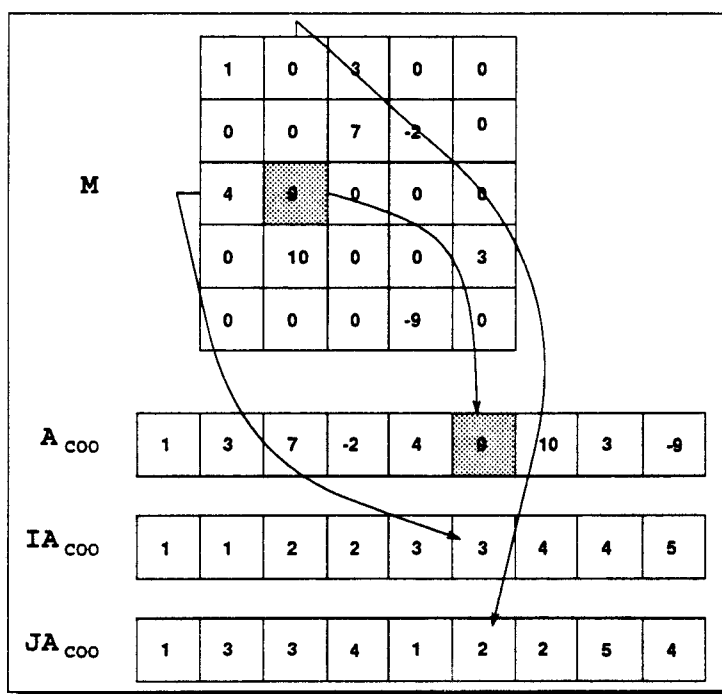


FIG. V.1 - Le format de compression COO

1. Le patron d'une matrice est la géométrie irrégulière décrite par ses éléments non-nuls.
2. matrices diagonales, tri-diagonales, ..., n -diagonales.

Une bonne caractéristique de ce format de compression est de permettre le traitement de matrices déséquilibrées : il n'y a pas *a priori* de nombre maximal d'éléments significatifs pour une ligne (ou une colonne) de la matrice de départ. Le nombre d'éléments stockés et la taille de la structure de donnée principale coïncident parfaitement.

Ce format a le désavantage de consommer beaucoup de mémoire pour le stockage des deux vecteurs de coordonnées. Par contre, le taux de compression des éléments significatifs est maximal.

Pour l'exploitation d'un tel format de compression, le balayage de la matrice pour le traitement élément par élément est aisé : il suffit de balayer la matrice A_{COO} . Par contre, l'accès à un élément précis en fonction de ses coordonnées nécessite une double recherche dans les vecteurs d'indices.

1.1.2 Les formats CSR et CSC

Le format *Compress Sparse Row*³ est apparu avec les calculateurs vectoriels. Le format issu de la compression est un ensemble de trois vecteurs (cf. figure V.2) :

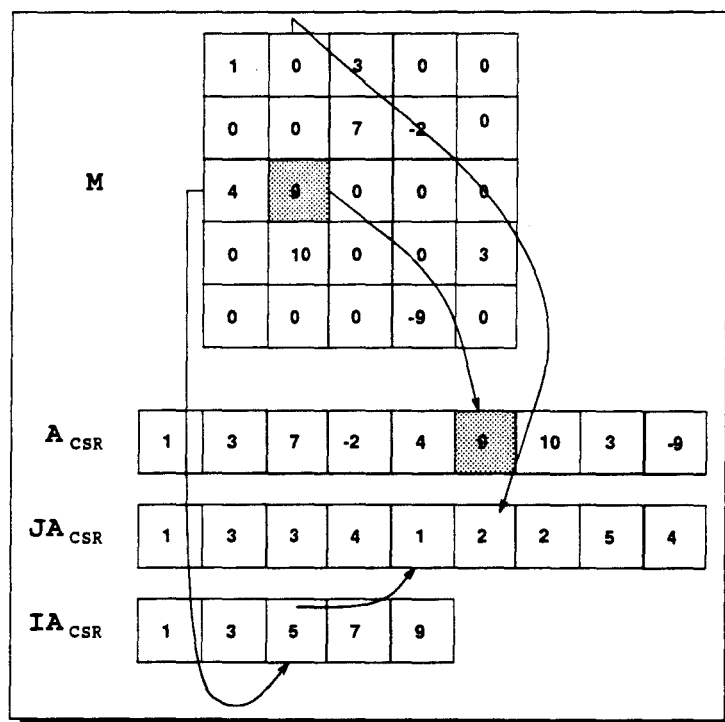


FIG. V.2 - Le format de compression CSR

- $ACSR$ contient les éléments non nuls de (a_{ij}) , lus avec un balayage de la matrice ligne par ligne. La taille de ce vecteur est donc égale au nombre d'éléments significatifs ;
- $JACSR$ contient la coordonnée horizontale des éléments de $ACSR$ dans la matrice de départ ;
- $IACSR$ contient les indices des éléments de $ACSR$ qui correspondent au premier élément non-nul de chaque ligne de la matrice de départ (la taille de ce vecteur est égal à la racine du nombre total d'éléments).

Le format *Compress Sparse Column*⁴ (CSC) est l'équivalent de CSR pour la compression par colonnes. Le stockage est toujours effectué sur trois vecteurs.

Comparativement au format COO, les deux formats CSR et CSC sont plus économiques en occupation mémoire. De plus l'accès à un élément par l'intermédiaire de ses coordonnées est facilité par le second vecteur qui donne directement un accès à la portion de la matrice compressée qui correspond à la ligne de l'élément recherché. La recherche est donc deux fois plus rapide, en moyenne pour une recherche séquentielle.

La difficulté d'utilisation de ces formats pour une architecture massivement parallèle réside dans le fait que le stockage doit être projeté sur une topologie qui n'est pas, en général, linéaire.

1.1.3 Le format ELL

Contrairement aux formats précédents, le format Ellpack-Itpack [Saa90] utilise des tableaux à deux dimensions, qui sont plus efficacement exploités pour une architecture non-linéaire (surtout pour une architecture en grille!).

Ce format comporte deux matrices dont la taille d'un côté est égal au nombre maximum d'éléments significatifs par ligne de la matrice d'origine, et dont la taille de l'autre côté est égal à celle d'un côté de la matrice d'origine. Ce format est donc plus consommateur de mémoire que le précédent, mais permet une exploitation plus facile en deux dimensions.

4. Compression par colonnes

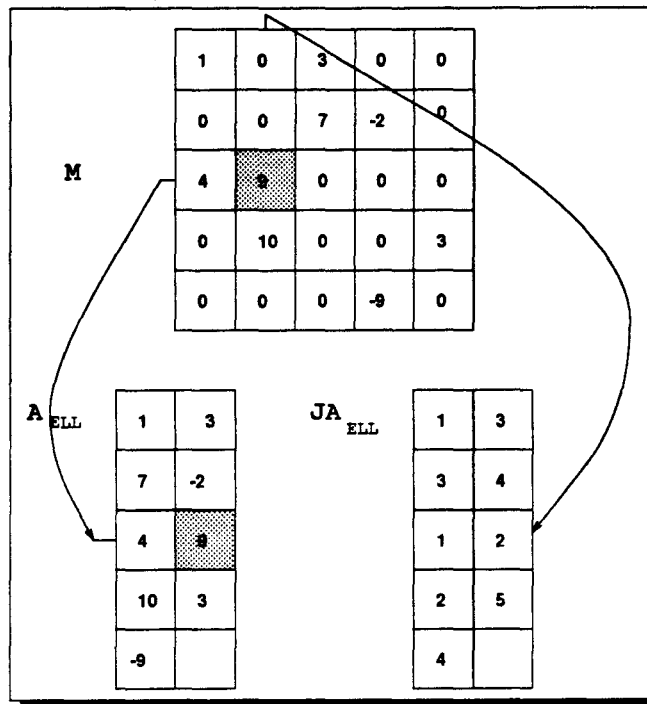


FIG. V.3 - Le format de compression ELL

Les deux matrices de compressions comportent les informations suivantes (cf. figure V.3):

- A_{ELL} contient les compressions par lignes de la matrice d'origine ;
- JA_{ELL} contient la coordonnée horizontale de chaque élément de A_{ELL} , relativement à la matrice de départ.

Un important problème de définition de ce type de compression est la supposition faite qu'une ligne de la matrice ne comporte pas un nombre trop important d'éléments non-nuls. Il faut, pour que l'occupation mémoire soit efficace, que le nombre d'éléments significatifs par ligne de la matrice de départ soit équilibré. Généralement, on associe avec ces matrices un coefficient d'éléments significatifs par ligne, qui détermine la largeur de la matrice A_{ELL} (et par conséquent de la matrice JA_{ELL}). Cette condition est plus contraignante que les formats de stockage CSR et CSC.

1.1.4 Le format SGP

Le format *Sparse General Pattern* [SPNR91] est considéré comme la version « data-parallel » du format précédent [PE94]; c'est aussi un format capable de compresser les lignes

ou les colonnes d'une matrice creuses. Ce format, plus complet que les précédents, permet de trouver plus facilement la correspondance entre la matrice compressée et sa transposée.

La compression donne trois structures (cf. figure V.4):

- A_{SGP} contient les colonnes compressées ;
- IA_{SGP} contient la coordonnée verticale des éléments de A_{SGP} ;
- JC_{SGP} contient le numéro de l'élément correspondant de A_{SGP} dans l'ensemble ordonné des éléments non-nuls de la ligne de l'élément, dans la matrice de départ.

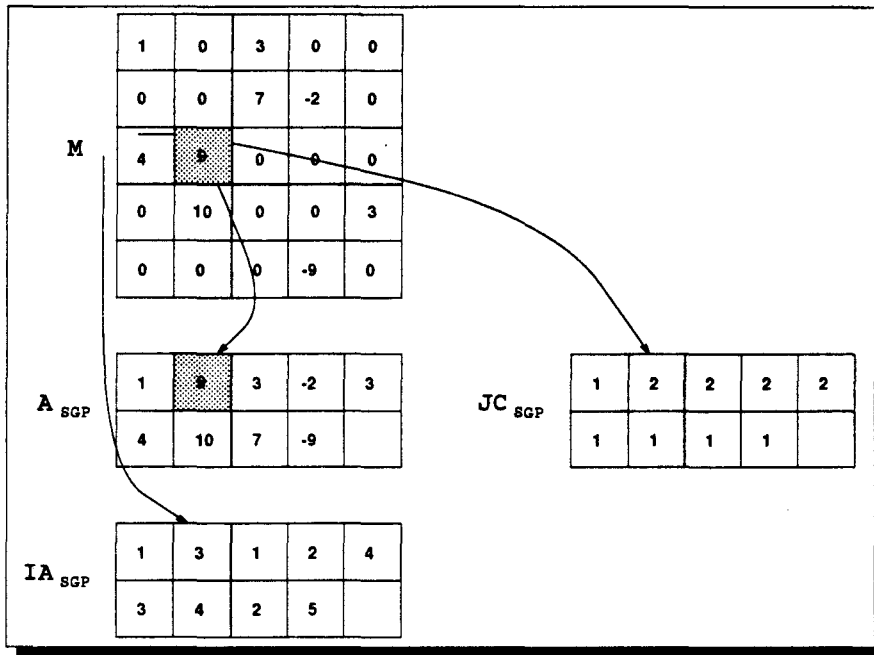


FIG. V.4 - Le format de compression SGP

Ces trois matrices possèdent la même taille, identique à la matrice JA_{ELL} . Ce format présuppose aussi que les éléments non-nuls sont répartis équitablement par colonnes (ou par lignes) dans la matrice de départ, sous peine d'occupation mémoire plus importante que le strict nécessaire.

Grâce à ce format, les opérations faisant intervenir la transposée de matrice sont plus efficacement calculables sur les machines massivement parallèles. Le nombre de communications est minimisé pour le calcul de la coordonnée transversale à la dimension de compression [PE94]. En contrepartie, l'occupation mémoire se trouve augmentée par la définition de cette troisième matrice.

1.1.5 Le format S^2

le format *Symmetrical Scan-class pattern* (S^2) défini dans [PE94] est constitué des compressions par ligne et par colonne suivant le format SGP précédemment exposé. Ce double stockage de la matrice permet des traitements suivant les deux formes compressées. Par exemple, une réduction le long d'une des deux dimensions de la matrice sera effectué à partir de la version compressée suivant cette dimension, réduisant ainsi le nombre de communications sur la topologie cible. Cette facilité et cette efficacité d'exploitation sont très chères en terme d'occupation mémoire: l'encombrement d'une matrice est doublé.

1.1.6 Le format S^3

L'utilisation des formats précédents (ELL, SGP, S^2) est limitée aux cas où le nombre d'éléments non-nuls est équilibré entre les lignes et les colonnes. La généralisation à des matrices qui ne possèdent pas cette propriété a permis la définition de formats voisins, mais qui peuvent néanmoins supporter un plus large spectre de matrices creuses.

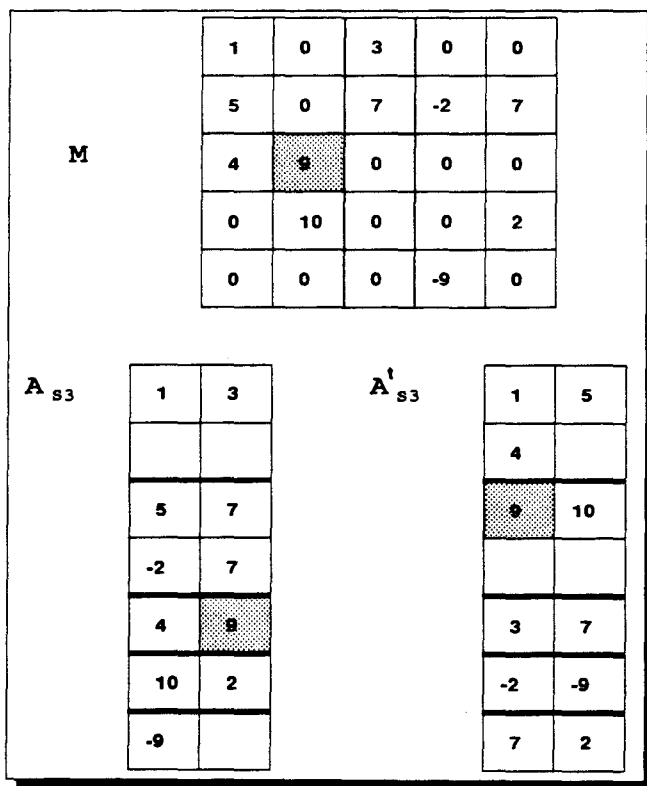


FIG. V.5 - Le format de compression S^3

Le format *Symmetrical Sparse general pattern and Scan class* (S^3) a été ainsi défini en ce sens [PE94]. Le système de compression par ligne est complété pour résoudre le cas où une ligne de la matrice comporte un grand nombre d'éléments non-nuls, par rapport aux autres lignes. Un découpage est alors opéré pour « régulariser » la taille de la matrice compressée.

De plus, un format est issu du calcul du maximum de nombre d'éléments non-nuls d'une ligne et d'une colonne. Les matrices issues de ce système de compressions peuvent alors recevoir le stockage compressé de la matrice de départ ou de sa transposée. Suivant les traitements opérés, on choisira l'une ou l'autre de ces deux compressions (cf. figure V.5).

Les informations « annexes » permettant l'accès aux éléments suivant leur position par rapport à l'origine dans la matrice de départ peuvent être de différentes formes (la plus simple d'entre elles est composée de deux matrices de même taille, associant à chaque élément ses coordonnées dans la matrice d'origine).

1.2 La programmation du creux

Les applications usuelles qui manipulent des structures creuses résident souvent sur un traitement itératif qui permet la convergence vers la matrice solution du problème [WD94]. Une itération est composée de calculs assez simples dans la version dense, mais qui prennent toute leur importance dans le cas du creux, du fait de la manipulation explicite du format compressé [PWD92b].

Sparsekit est la seule véritable boîte à outils de développement d'algorithmes d'algèbre linéaire dans le cas du calcul creux. Proposée par Youcef Saad [Saa90], elle regroupe un grand nombre d'interfaces qui permettent d'utiliser les routines spécifiées sur des matrices de diverses formes de compression, dont celles que nous venons d'énumérer.

En proposant cet ensemble de fonctionnalités, un des buts de Saad était de faciliter les échanges de données et de programmes entre les chercheurs du domaine du calcul creux. Seul « environnement » du creux, *Sparsekit* est alors devenu un passage obligé pour la portabilité des programmes, mais aussi des données. Alors que chacun évoluait dans son propre format de compression, l'ouverture rendue possible par un format « universel⁵ », a permis de reconnaître le calcul creux comme un domaine de recherches à part entière, et pas uniquement un moyen de résoudre ponctuellement des applications particulières.

La manipulation de matrices creuses est aujourd'hui entièrement développée à la main. Le format de compression est la base même de l'algorithmique creuse. Un format est choisi plutôt qu'un autre en fonction de la machine cible et des traitements à opérer. De plus, ces

5. ou tout au moins une collection de formats reconnus.

traitements sont directement exprimés en tenant compte de la compression adoptée.

La programmation du creux consiste par conséquent à rompre le référentiel de la matrice d'origine par l'abandon des coordonnées cartésiennes des éléments non-nuls. Puis, de retrouver les traitements équivalents avec le nouveau système de coordonnées dynamiques (dépendantes du patron de la matrice). En effet, comme nous l'avons déjà évoqué, les coordonnées nouvelles ne sont pas indépendantes des valeurs contenues dans la matrice. Généralement, une des deux coordonnées d'un élément significatif représente son numéro d'ordre sur la ligne parmi les éléments non-nuls (en compressant par lignes, un élément se retrouve en troisième colonne si il y a deux éléments non-nuls à sa gauche sur la même ligne de la matrice d'origine). Il y a donc interférence entre les données et le référentiel, ce qui se traduit en terme d'exécution par un programme qui manipule de nombreuses structures dynamiques, et effectue un grand nombre de balayages de vecteur pour la recherche d'éléments.

Le coût en terme de temps d'exécution introduit par l'utilisation de compression et de traitements indirectes sur des données dynamiques est généralement très élevé. On ne s'étonne pas de trouver des taux de parallélisme très bas sur des machines massivement parallèles. Toutefois, ce coût est à comparer avec le gain que l'on obtient en ne considérant que les éléments significatifs. Il se peut (et c'est souvent le cas) qu'un calcul creux sur machine parallèle n'utilise qu'une faible proportion de la puissance théorique de la machine, mais soit encore nettement plus rapide que le traitement dense.

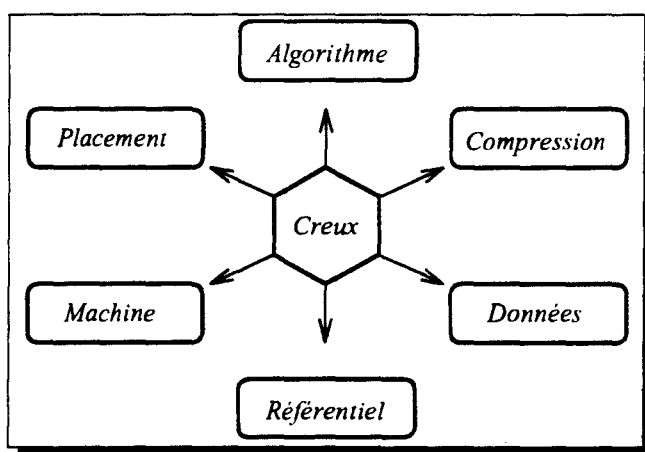


FIG. V.6 - Les interférences du calcul creux

En terme de coût de programmation, le creux nécessite souvent le développement d'un programme par application, voir même par matrice d'origine. En effet, le seul fait de changer les données du programme (changer la répartition des éléments significatifs) peut d'une part faire exploser la capacité mémoire si la compression adoptée n'est plus compatible avec le

patron de la matrice ou d'autre part rendre totalement inefficace le programme. Beaucoup d'algorithmes prennent en compte la moindre particularité de la matrice de départ (si celle-ci est une matrice diagonale ou bande...). Le calcul creux est donc un domaine qu'il faut appréhender en globalité (cf. figure V.6).

2 HELP et le creux

NOTRE ambition dans le domaine que nous venons d'aborder est simple : nous voulons produire un environnement qui propose à l'utilisateur de « programmer du creux comme du dense ». En termes plus précis, nous voulons élaborer un langage qui autorise le programmeur à exprimer ses algorithmes directement au niveau des structures de départ, en s'abstrayant au maximum du fait que ces données constituent une matrice creuse. Avec un tel produit, il n'existe plus de développement explicitement basé sur la compression des données, qui est ramenée à un niveau inférieur, pendant la phase de génération de code.

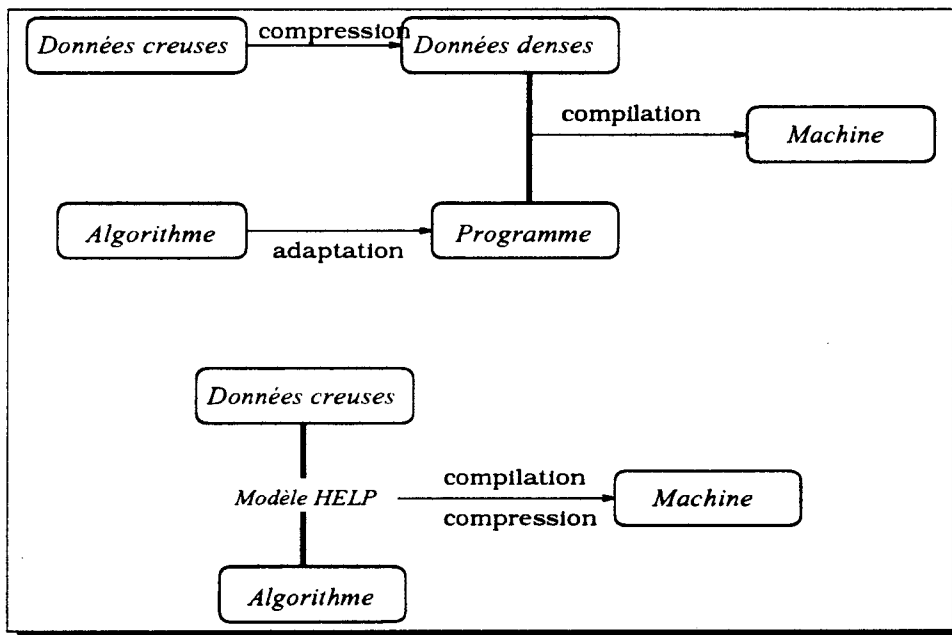


FIG. V.7 - Le creux par la compilation

2.1 Modèle de programmation

Le modèle HELP et le langage C-HELP que nous lui avons associé sont en bonne adéquation avec l'algèbre linéaire (cf. chapitre d'exemples). Le calcul creux est à l'évidence un sous-domaine de ce type d'applications [PWD92a] et doit donc pouvoir utiliser HELP pour

profiter de ses qualités d'expressivité et de clarté.

De plus, la notion primordiale de conformité permet d'optimiser la génération de code pour le cas du dense. Nous avons prouvé que l'évaluation d'une expression microscopique pouvait être rendue plus rapide par la factorisation des calculs d'adresses. En gardant ce principe fort, nous pourrions étendre ce résultat au cas du calcul creux.

C'est donc en ce sens que nous décidons de développer l'environnement dédié au creux. Nous gardons les concepts de base du modèle géométrique, à savoir la notion d'hyper-espace qui propose un référentiel pour toutes les phases du programme, la notion de DPO qui uniformise les objets parallèles et la règle de conformité qui instaure la manipulation explicite des données par une séparation nette entre les phases calculatoires et les phases de communications. Nous ne nous autorisons qu'un minimum de modifications au niveau du langage C-HELP.

2.2 Creux et hyper-espace

Pour rester dans le modèle géométrique et conserver l'hyper-espace comme entité fédératrice de tout traitement, nous allons associer le creux au référentiel plutôt qu'à un objet data-parallèle. C'est en effet la seule façon de conserver l'efficacité du modèle par le respect des règles de programmation, et en particulier de la conformité logique et physique. Nous avons montré dans le chapitre précédent que tous les accès aux données étaient effectués par intermédiaire des points du référentiel. Dès lors, une compression des données différente pour deux objets ne permet plus de retrouver la conformité au niveau intermédiaire des données compressées (entre les objets creux et les données physiquement allouées).

La solution unique nous permettant de garder la conformité – et de fait, la localité des données à l'exécution – est donc d'associer la compression aux points de l'hyper-espace.

Dès maintenant, il nous faut justifier l'acceptabilité de ce choix. En parcourant les applications développées pour le calcul creux [SPNR91, PSWF93, WD94], on s'aperçoit qu'un algorithme ne considère qu'un unique patron à un instant donné du déroulement du programme. Le fait de décrire le patron sur le référentiel n'est donc pas limitatif du point de vue de l'expressibilité de ces algorithmes⁶.

Avec ce choix, nous conservons intégralement le modèle de programmation HELP. Le seul problème de gestion du creux est donc relatif à la compilation du langage C-HELP. Nous allons montrer les quelques extensions de ce langage pour permettre une prise en compte du creux.

6. De plus, il est toujours possible d'évoluer sur plusieurs hyper-espaces.

2.3 Creux et C-HELP

Puisque nous attachons le patron à l'hyper-espace, la déclaration de celui-ci doit comporter la spécification de compression. Nous donnons donc un supplément syntaxique pour déterminer quelle sera la dimension de l'hyper-espace bidimensionnel qui sera compressée.

2.3.1 Déclaration d'un hyper-espace

La déclaration d'une dimension peut maintenant comporter le mot-clef **sparse** à la place du facteur de bloc de cette dimension. Le compilateur va générer un mécanisme de compression le long de cette dimension. Une seule dimension peut être spécifiée creuse. Cette restriction est cohérente avec la problématique du creux : une matrice (ou un hyper-espace) compressée dans une dimension devient dense et la compression dans une seconde dimension n'a pas de signification.

La dimension ainsi déclarée comporte une information de bloc particulière, elle ne peut donc pas être porteuse d'un facteur de bloc. De plus, elle ne peut plus apparaître dans l'arbre de priorité décrivant l'ordre de parallélisme entre les diverses dimensions de l'hyper-espace, on la considère comme ayant une priorité infiniment basse. On a ainsi décidé de projeter en mémoire cette dimension. Comme nous le verrons en décrivant l'allocation mémoire, cette convention nous permet d'effectuer l'accès associatif en un temps raisonnable (logarithmique) et de supprimer les communications dues à la compression.

Le patron de l'hyper-espace va déterminer les points supports de données ou les points non-significatifs. En entrée de l'application⁷, le programmeur doit faire appel à la procédure d'entrée des valeurs constituant la matrice creuse de référence, qui va donner le patron de l'hyper-espace. Pour cela, il utilise la fonction **PatternCOO** décrite en figure V.8. Cette fonction prend en argument le nom de la matrice de référence qui va donner le patron de l'hyper-espace, et un descripteur de fichier. Ce descripteur doit pointer un fichier ouvert, contenant la compression COO de la matrice⁸ : c'est-à-dire un flot constitué des valeurs non-nulles munies de leurs indices dans chaque dimension de l'hyper-espace, suivant l'ordre de déclaration de ces dimensions.

7. et une seule fois pour le programme.

8. Nous choisissons COO pour sa simplicité de mise en œuvre. On peut imaginer d'autres fonctions de lecture qui acceptent d'autres formats en entrée.


```

hspace plan [x=100,y=1000(sparse)];
dpo double plan [x=*,y=*] M;

main() {
    int fd;
    ...
    PatternCOO(M,fd);
    ...
}

```

FIG. V.8 - La création du patron de l'hyper-espace

L'allocation dynamique des points est réalisée à cet instant et l'affectation du DPO creux est réalisée sur chaque point alloué.

2.3.2 Déclaration d'objets

La programmation dans un univers creux peut renfermer des manipulations de structures correspondantes au patron, mais aussi à des objets denses. La distinction ne doit pas être explicitée lors de la déclaration. Un DPO devient creux lorsque son allocation sur l'intervalle de son domaine d'allocation sur la dimension creuse est différent de [1..1] (cf. figure V.9). Cette distinction permet d'allouer dans le même hyper-espace les objets creux et denses, par la conservation de l'allocation mémoire d'un sous-espace complet.

```

hspace plan [x=100,y=1000(sparse)];

dpo double plan [x=*,y=*] M; /* dpo creux */
dpo double plan [x=*,y=1] V; /* dpo dense */

V.shifttor(x,10)           /* dpo dense */
V.translate(y,1)           /* dpo creux */
V.exchabs(x,y)             /* dpo creux */

```

FIG. V.9 - Les objets denses et creux

Pour les DPO non-creux, la déclaration doit donc impérativement être effectuée sur l'origine de la dimension creuse.

2.3.3 Hyper-espace et trous noirs

Le compilateur projette en mémoire la dimension creuse. Comme dans toutes les solutions existantes, une compression est effectuée sur cette dimension et seuls les points comportant un élément non-nuls seront alloués. On conservera aussi l'information que l'on perd lors de cette compression à savoir la coordonnée du point suivant l'axe creux. La figure V.10 représente la compression mise en place : un point faisant partie du patron conserve sa coordonnée dans la dimension non-compressée, tandis qu'on lui associe la coordonnée dans la dimension faisant l'objet de la compression. Les points de coordonnée 1 sur la dimension compressée qui ne font pas partie du patron sont aussi alloués, mais sont associés à l'information 0 permettant de les distinguer des autres points. Cette distinction est nécessaire pour réussir à donner une sémantique uniforme pour les opérations effectuées sur un domaine creux.

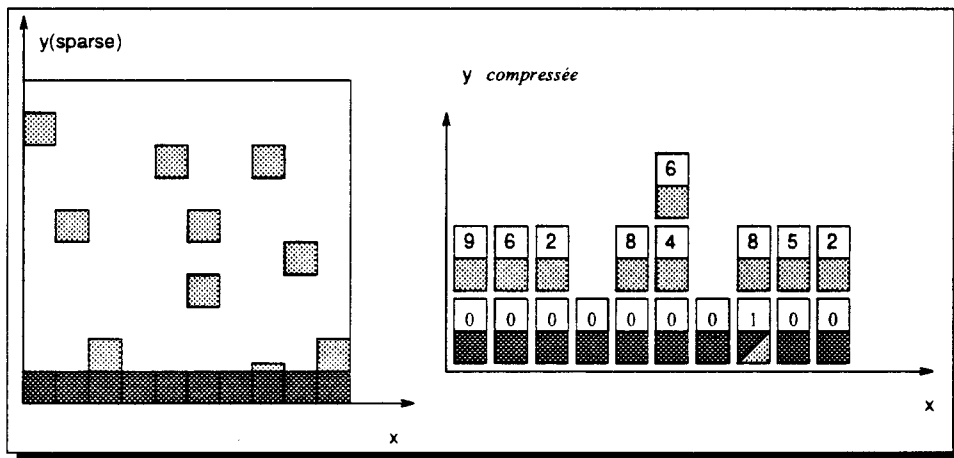


FIG. V.10 - La compression des points

Les points qui ne sont pas alloués lors de la compression sont appelés *trous noirs*, à l'image de l'astronomie. **Toute valeur arrivant sur un trou noir est définitivement perdue.** C'est le complément au modèle de programmation HELP pour le creux.

Ainsi, dans la figure V.9, les opérations macroscopiques appliquées au vecteur dense V peuvent amener à perdre certaines valeurs du vecteur de départ : toutes les valeurs tombant dans un trou noir sont irrémédiablement détruites. La géométrie de l'objet est néanmoins conservée, même si la valeur sur une des bornes du domaine d'allocation est absorbée par un trou noir.

2.3.4 Domaine de conformité creux

Tout segment conforme dont l'évaluation est déclenchée sur un domaine de conformité qui comporte au moins un point de coordonnée différente de 1 sur la dimension compressée, sera évalué sur le domaine de conformité creux. Ce domaine est issu de l'intersection du domaine de conformité géométrique (tel que nous l'avons défini au chapitre exposant le langage) et du patron de l'hyper-espace.

Lors d'une telle évaluation, le nombre d'itérations nécessaires pour l'activation des points alloués est donc égal, pour chaque colonne (sous-espace de coordonnées constantes pour la(es) dimension(s) dense(s)) au nombre de points alloués et faisant partie du domaine géométrique.

2.3.5 Appels macroscopiques et creux

Lors de l'application d'une suite de primitives macroscopiques, seuls les points alloués sont considérés comme potentiellement source. Le domaine source creux est donc défini comme l'intersection du domaine géométrique source (comme nous l'avons défini dans le cas du dense) et du patron de l'hyper-espace.

L'attention du programmeur doit être portée vers les conséquences de l'adoption de cette règle. D'une part, on ne retrouve plus l'« associativité » des applications d'opérateurs macroscopiques (que l'on avait d'ailleurs que dans le cas où les conditions sur les tailles des dimensions de l'hyper-espace n'étaient pas violées). Voir figure V.11.

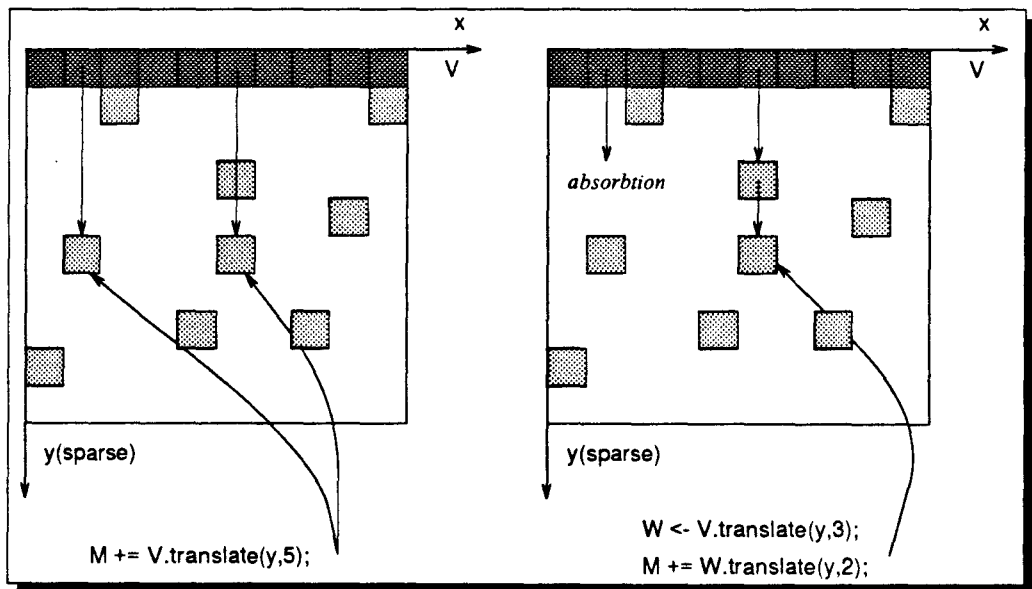


FIG. V.11 - *Domaine source creux*

De plus, les règles sémantiques que nous venons d'adopter s'appliquent aussi à l'appel de réducteurs associatifs. Seuls les points alloués sont sources de la réduction. Pour les réductions avec addition, cela n'a guère d'importance. Par contre, pour les réductions avec un opérateur dont 0 n'est pas élément neutre, le résultat est différent en adoptant cette sémantique. Par exemple, lors d'une réduction avec multiplication, les trous noirs ne sont pas émetteur de valeur. La multiplication se fait donc exclusivement sur les points alloués (cf. figure III.36). Le programmeur peut ainsi calculer le produit des éléments non-nuls de la matrice de départ⁹.

2.4 La compilation du creux

Comme nous l'avons présenté, la compilation est l'étape qui masque la compression des données. Nous utilisons une compression qui introduit sur chaque point l'indice le long de la dimension creuse, perdu lors de cette phase.

2.4.1 Allocation mémoire

Les points de coordonnée 1 dans la dimension compressée sont forcément alloués. Les autres points sont alloués dynamiquement suivant le patron de la matrice de référence.

L'algorithme d'allocation dynamique présenté au chapitre précédent est toujours valable dans le cas du creux. Le calcul de recouvrement est toujours calculé en fonction de la géométrie indépendamment du patron de l'hyper-espace. Puisqu'il n'existe qu'un patron pour tous les objets creux de l'hyper-espace, le fait de ne pas prendre en compte le creux n'apporte aucune pénalité quant au taux d'occupation mémoire.

2.4.2 Accès aux données

L'accès aux éléments est toujours exprimé par la donnée des coordonnées de chaque points accédé. Pour la dimension creuse, le point correspondant à l'indice dans la dimension compressé est recherché par dichotomie, les points étant triés en mémoire suivant leur indice dans la dimension creuse. Une complexité logarithmique est ainsi introduite¹⁰. La projection en

9. Pour retrouver la sémantique de la réduction avec multiplication qui tiendrait compte des éléments nuls (pour calculer si une colonne de la matrice contient ou non un ou plusieurs éléments nuls), le programmeur peut appliquer une réduction avec addition sur un DPO creux constitué de 1. Cette nouvelle sémantique permet donc plus d'expressivité pour le langage.

10. logarithme du nombre de points sur la colonne compressée

mémoire de la dimension compressée permet d'effectuer cette recherche sans introduire de nouvelles communications induites par la compression.

La dimension creuse étant projetée en mémoire, la fonction de distribution qui associe un n -uplet de coordonnées à un couple (numéro de processeur, numéro du point) est toujours valide pour le calcul du numéro de processeur. Pour le numéro du point, il y a recherche dichotomique, à l'intérieur de la mémoire du même processeur physique. Il n'y a donc, conformément à la volonté de garder un langage explicite, aucune communication non explicitée par le programmeur.

2.4.3 Boucle de virtualisation

Pour la gestion de la boucle de virtualisation, l'introduction d'irrégularité est conservée. En fonction du domaine géométrique, chaque processeur recherche les points qui entrent dans un calcul microscopique (ou un appel macroscopique). Le nombre d'itérations nécessaires est donc toujours égal au nombre maximal de points alloués actifs contenus dans le domaine de conformité, pour un processeur.

2.4.4 Appels macroscopiques

L'envoi de valeur par un point source est dirigé vers un processeur. Un message est émis, il contient la valeur à transmettre, et la coordonnée du point destination sur la dimension compressée.

Contrairement au cas du dense, la valeur n'est pas directement affectée au point destination¹¹. Quand un processeur cible reçoit un message, il effectue une recherche dichotomique pour trouver le point destinataire. Si le point est alloué, il y a affectation ; sinon, la valeur est absorbée par le trou noir.

2.4.5 Dynamicité du patron

Dans la version actuelle du langage, le patron associé à un hyper-espace est statique. Établi en entrée du programme, il ne peut être modifié durant l'exécution d'un algorithme. La sémantique d'un appel macroscopique est modifiée par cette convention. Le programmeur qui désire néanmoins allouer de nouveaux points a la possibilité d'utiliser un autre hyper-espace de patron différent et de transférer ses données.

11. par l'utilisation de la bibliothèque MPL de communications

L'étude de l'introduction de dynamicité pour le patron d'un hyper-espace a , pour l'instant, trouvé une solution syntaxique consistant à introduire une nouvelle primitive insérée dans la liste des appels macroscopique. Cette primitive force l'allocation du point destinataire, qu'il soit déjà alloué ou qu'il soit un trou noir. Il n'y a pas de modification de la géométrie du DPO produit.

Cette solution simple au niveau de l'écriture d'un programme engendre un problème de compilation important : comment modifier l'allocation des points sur un processeur physique. Il paraît inévitable que cette modification est fatalement très coûteuse en terme de temps d'exécution (une grande partie de la mémoire d'un processeur, si ce n'est la totalité, doit être réorganisée pour garder l'ordre d'allocation permettant un accès associatif de complexité logarithmique, ce qui nécessite des copies de tranches mémoire).

Ce problème rejoint celui de l'allocation des points pour un hyper-espace. Cette allocation est aussi, dans l'état actuel, réalisé statiquement par la division sur les points de la mémoire disponible. La dynamicité de cette allocation est un problème non-résolu.

```

hspace plan [x=N,y=M(sparse)];

dpo float plan [x=*,y=*] Mat;
dpo float plan [x=*,y=1] Vec;
dpo float plan idem(Vec) Pro;

main(){
  int fd=...;
  PatternCOO(M,fd);

  /* extension du vecteur avec absorption par      */
  /* les trous noirs de l'hyper-espace,          */
  /* puis multiplication sur le domaine creux    */

  Pro = (Mat * Vec.expand(y)) . reduceadd(x).exchabs(y,x);

  /* réduction à partir des points alloués vers  */
  /* un domaine dense (y=1) => pas d'absorption. */
}

```

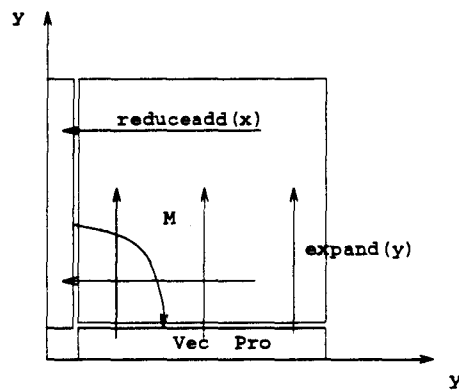


FIG. V.12 - Multiplication matrice/vecteur

2.5 Exemple

Un problème simple est souvent retenu pour illustrer la programmation des problèmes creux : la multiplication matrice/vecteur. Cette opération élémentaire est écrite en C-HELP dans la figure V.12.

Ce code montre le peu de modifications introduites pour la prise en compte du creux. En particulier, l'algorithme s'écrit de façon identique à celui qui aurait pour objet la multiplication matrice/vecteur dans le cas du dense.

3 Conclusion

LE CALCUL creux demande toujours une compression des données. Alors que les développements d'algorithmes sont toujours explicités sur les données denses issues de cette compression, nous avons proposé d'intégrer cette phase dans la compilation, au prix de quelques extensions sur le langage C-HELP. Le programmeur garde ainsi le référentiel avant compression, ce qui lui permet d'écrire ses programmes en s'abstrayant totalement des structures issues de la compression.

Nous avons introduit la notion de trous noirs qui permet une gestion des traitements sur les structures creuses en un temps fonction du nombre d'éléments non-nuls. De plus, nous avons étendu la notion de points sources aux points effectivement alloués, ce qui nous a permis de donner aux appels macroscopiques (et aux fonctions de réduction) une sémantique différente, qui donne au langage plus d'expressivité.

L'accès aux données est réalisé par un accès associatif que l'on retrouve inévitablement dans toute manipulation du creux. La distribution de données par l'adoption d'un format spécifique permet d'éviter la génération de communications liées à la phase de compression. La recherche associative est alors réalisée en un temps d'exécution logarithmique.

L'intégration effective de ces propositions dans notre atelier de compilation va permettre d'évaluer le sur-coût de la gestion du creux par le compilateur C-HELP. Dans la pratique, le développement des applications manipulant du creux, à l'aide de l'utilisation explicite des compression présentées en début de chapitre, engendre un facteur 2 ou 3 dans la baisse des performances.

Chapitre VI

Des exemples

POUR montrer qu'il est simple de programmer à l'aide du modèle géométrique, nous avons choisi en guise de dernier chapitre de cette thèse de proposer quelques exemples de programmes C-HELP. Nous commencerons par un exemple trivial qui montre l'usage d'un référentiel de calcul qui permet de résoudre un problème en grande partie par le placement des données.

Nous proposons ensuite le développement de quelques algorithmes de calculs matriciels. Pour ce faire, nous détaillerons notamment précisément un algorithme d'inversion de matrices (algorithme de Gauss-Jordan) qui permettra de mettre en valeur l'adéquation de HELP à ce type de calculs.

Enfin, nous présenterons deux applications réelles : la squelettisation d'image et un modèle hydro-dynamique 2D.

1 Énumération d'un vecteur

NOUS disposons d'un vecteur de N éléments. Le but est de créer un vecteur de même taille N dont les éléments représentent le rang de chaque élément du vecteur d'origine, suivant l'ordre décroissant.

Pour ce faire, il faut comparer tous les éléments deux à deux, ce qui représente N^2 opérations. Pour obtenir un maximum de parallélisme potentiel pour cette étape de comparaison, nous nous placerons dans un hyper-espace qui comporte N^2 points. Par conséquent, il faudra déclarer un hyper-espace planaire $N \times N$. Le positionnement initial du vecteur est effectué sur la première ligne, et une copie du vecteur sera positionnée sur la première colonne. Ainsi, on est capable, en répliquant les vecteurs le long de la verticale pour le premier et le long de l'horizontale pour le second, d'obtenir deux matrices conformes qui vont permettre le déclenchement en parallèle des comparaisons deux à deux des éléments du vecteur initial (cf. figure VI.1).

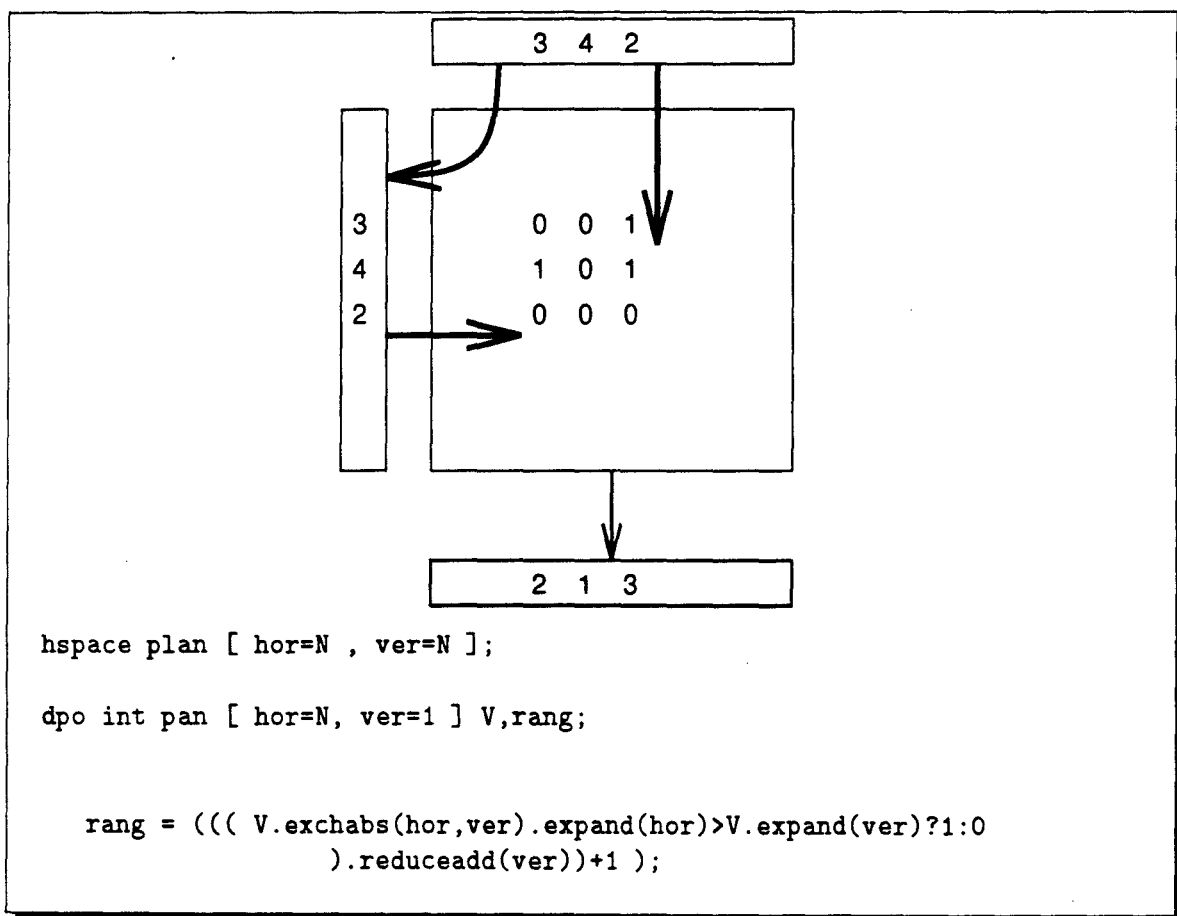


FIG. VI.1 - Calcul du vecteur de rangs : l'algorithme et le programme C-HELP

La matrice résultante de ces comparaisons est booléenne. Quand la première valeur est strictement plus grande que la seconde, le résultat est 1, sinon 0. Le rang de chaque élément est enfin obtenu par réduction avec addition de la matrice résultat, sur la première ligne (en ajoutant 1 pour commencer l'énumération à la valeur 1 pour le plus grand élément).

Pour le tri d'un vecteur, le même algorithme peut être utilisé. Les permutations sont ensuite effectuées par la primitive non-géométrique `scatter` avec le résultat du calcul précédent en argument. Nous avons montré l'utilisation d'un tel tri dans l'exemple illustrant l'adéquation de C-HELP à l'expression d'algorithmes hétérogènes (cf. chapitre sur le langage).

2 Calculs matriciels

2.1 La multiplication de matrices

La multiplication de matrice sur une machine à mémoire distribuée est décrite par un algorithme issu de la programmation à flot de données. Dans une phase préliminaire, les deux matrices à multiplier sont décalées suivant la description de la figure VI.2. Cette opération est décrite en C-HELP par une opération `scatter` permettant la réorganisation en un pas de communications : chaque ligne est décalée toriquement d'une longueur égale à l'ordonnée de la ligne (et réciproquement pour chaque colonne de la deuxième matrice).

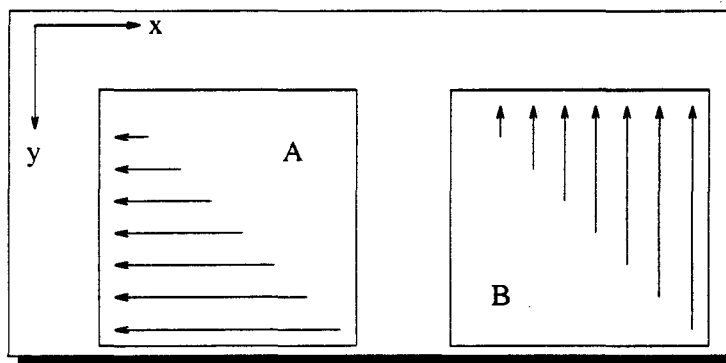


FIG. VI.2 - Décalages toriques irréguliers

Les deux matrices sont ensuite multipliées (opération microscopique) et décalées régulièrement, l'une suivant les abscisses, l'autre suivant les ordonnées. Cette opération est réalisée N fois pour la multiplication de matrices $N \times M$. Voir figure VI.3.

```

hspace plan [ x = N , y = N ];

dpo float plan invmat(dpo float plan A,dpo float B)
{
    int i;
    dpo float plan [x=N,y=N] R;

    /***** Initialisation *****/

    A = A.scatter( A.cabs(x)-(A.cabs(y)-1+N)%N , A.cabs(y) , A);
    B = B.scatter( B.cabs(x) , B.cabs(y)-(B.cabs(x)-1+N)%N , B);

    R = A*B;

    /***** Multiplication *****/
    for (i=1;i<N;i++) {

        A = A . shifttor(x,-1);
        B = B . shifttor(y,-1);

        R += A*B;
    }

    return R;
}

```

FIG. VI.3 - *La multiplication de matrices en C-HELP*

2.2 Algorithme de Gauss-Jordan

L'algorithme de Gauss-Jordan est très utilisé pour les inversions de matrices. Il repose sur le principe de base : soit A une matrice inversible, le produit de A par son inverse A^{-1} donne l'identité.

$$AA^{-1} = Id$$

Dès lors, en appliquant une matrice de passage à A , si on obtient l'identité, on peut déduire que cette matrice de passage est égale à l'inverse de A . L'algorithme repose sur ce principe :

$$AP_1 \quad IdP_1$$

$$AP_1P_2 \quad IdP_1P_2$$

$$AP_1P_2\dots P_n \quad IdP_1P_2\dots P_n$$

$$AP \quad IdP$$

On applique successivement et conjointement des matrices de passages aux deux matrices de départ (A et identité). Quand on obtient l'identité pour le résultat du calcul sur la première matrice, on en déduit que la matrice de passage, composée des matrices successives est égale à l'inverse de A car $(AP = Id) \Rightarrow (P = A^{-1}) \Rightarrow (IdP = A^{-1})$. On retrouve alors la matrice inverse à l'emplacement de l'identité au début du calcul.

De ce principe, on peut déjà définir le cadre de l'algorithme géométrique. Pour inverser une matrice carrée d'ordre N , on se placera dans un hyper-espace $2N \times N$ pour pouvoir appliquer conjointement les traitements sur deux matrices de même taille¹. La première partie de l'hyper-espace supportera la matrice à inverser, tandis que la deuxième partie supportera les traitements appliqués sur l'identité.

Pour opérer les transitions, on se base sur le modèle géométrique. La première phase va consister à diagonaliser la matrice de gauche. Pour cela, on opère des combinaisons linéaires sur les lignes afin d'annuler progressivement les colonnes de gauche. Voir figure VI.4.

1. Il serait possible d'écrire l'algorithme sur un hyper-espace $N \times N$, mais le programme s'en trouverait moins lisible.

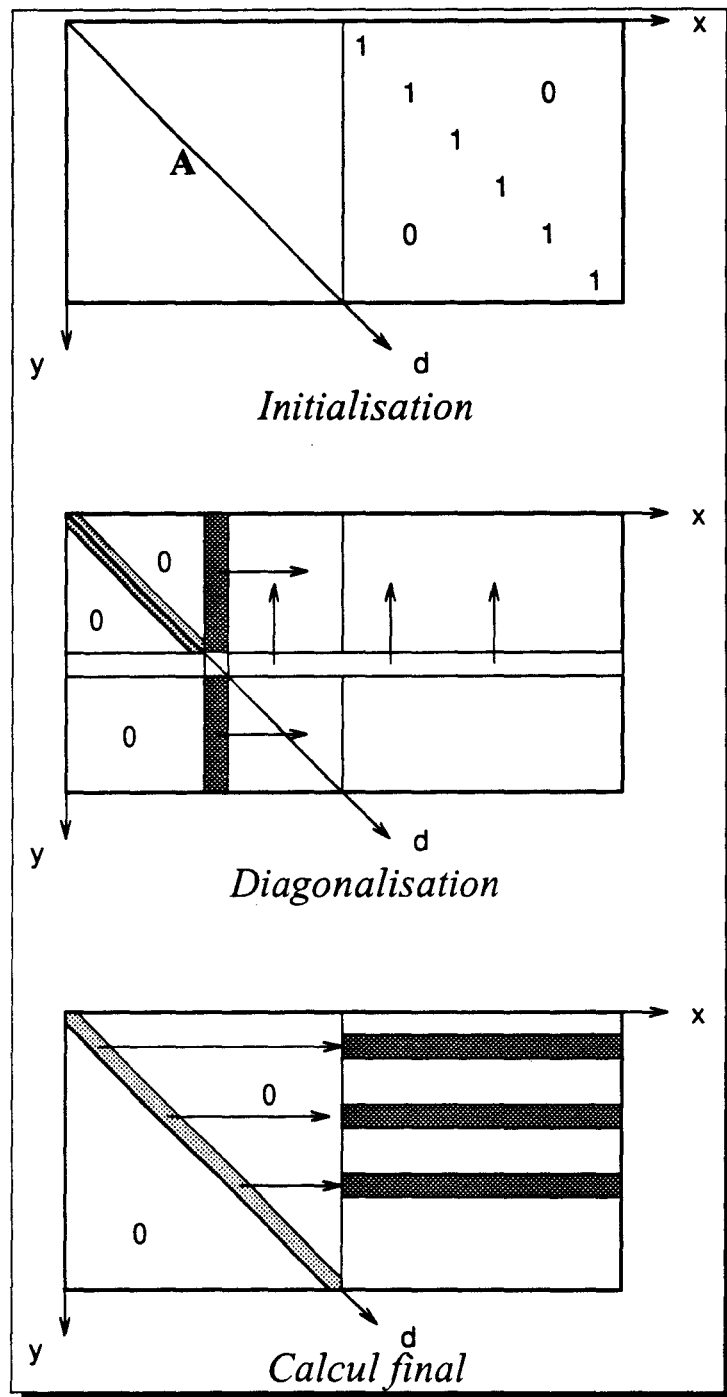


FIG. VI.4 - Gauss-Jordan: l'algorithme géométrique

Pour annuler les éléments situés sur la colonne du pivot, chaque ligne se voit soustraite de la ligne du pivot multipliée par le rapport entre l'élément à annuler et le pivot. Pour ce faire, on diffuse le long de la matrice entière la ligne du pivot et la colonne du pivot. Cette

opération est effectuée pour chaque colonne de la matrice en partie gauche, soit au total N fois.

La partie gauche de la matrice est maintenant diagonale. Pour obtenir l'identité, il suffit de diviser chaque ligne par la valeur de la diagonale. La modélisation géométrique consiste naturellement à diffuser ces valeurs sur la seconde moitié de l'hyper-espace (cf. figure VI.4). La partie droite est maintenant égale à l'identité. La partie gauche contient le résultat. Voir le programme en figure VI.5.

```

hspace plan [ x = 2*N , y = N , d=(x,y) ] (x,(y));

dpo float plan invmat(dpo float plan A)
{
  int i;
  dpo float plan [x=2*A.size(x),y=A.size(y)] M;
  dpo float plan [] A_INV;
  dpo float plan [d=A.size(x)] D;

  /***** Initialisation *****/
  M = A.moveabs(1,1);
  A_INV <- A.moveabs(N+1,1);
  on (A_INV) M = (M.cabs(x) == M.cabs(y)+N ?1:0) ;

  /***** Diagonalisation *****/
  for (i=1;i<=A.size(x);i++)
    where (M.cabs(y) <> i) {
      M -= M.extrabs(y,i).expand(y) *
          M.extrabs(x,i).expand(x) /
          M.scalar(i,i);
    }

  /***** Remontée *****/
  on (D) D = M;
  on (A_INV)
    A_INV = M / D.exchabs(d,y).transabs(x,N).expand(x,A.size(x));

  return A_INV.moveabs(A.cabs(x),A.cabs(y));
}

```

FIG. VI.5 - L'algorithme de Gauss-Jordan en C-HELP

2.3 Factorisation LU

La résolution de système linéaires tri-diagonaux sont souvent issus de problèmes physiques dans lesquels la phase de discrétisation a conduit à l'obtention de ces matrices particulières. Cette résolution peut se faire par la méthode de factorisation LU [CT93].

À partir d'une équation matricielle $Ax = V$, où la matrice A est tri-diagonale, on décompose le problème par la factorisation de A en $L \times U$ avec L et U deux matrices bi-diagonales (L inférieure et U supérieure). La résolution finale du système est obtenue ensuite par la résolution des deux équations obtenues en première phase : $Ly = V$ et $Ux = y$. En un premier temps, l'algorithme calcule le vecteur y par une « descente » le long de L , puis le vecteur x par une « remontée » le long de U (cf. figure VI.6).

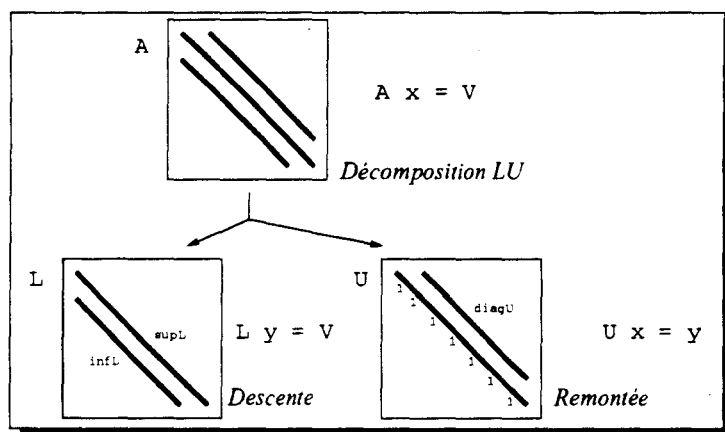


FIG. VI.6 - Les trois phases de la décomposition LU

Cette décomposition peut s'écrire en C-HELP par le code figurant en figure VI.6. Remarquons cet algorithme est écrit (et donc exécuté) en mode séquentiel : la factorisation LU n'est pas une méthode utilisée telle quelle pour le calcul parallèle [CT93]. On peut toutefois tirer partie de sa description en C-HELP qui donne une version plus proche du caractère géométrique de l'algorithme. En effet, les translations des diagonales inférieures et supérieures des matrices L et U lors des deux dernières phases permettent une factorisation des communications qui n'est pas facile de retrouver dans d'autres langages.

Les traitements obtenus sont alors plus nettement découpés suivant les vues macroscopiques et microscopiques.

```

hspace LUspace [ x=N , y=N , d=(x,y)] (x,y);

dpo float LUspace [x=*,y=*] A,L,U;
dpo float [d=*] V,X,Y;
/* les vecteurs de départ et de résultat sont sur la diagonale. */
dpo float LUspace [y=2,d=*] infA;
/* copie de la diagonale inférieure ramenée sur la diagonale */
dpo void LUspace [ x=1, y=1] supL;
dpo void LUspace [x=2:2, y=1] diagU;

/***** Décomposition LU *****/
infA <- (on(infA)A).transrel(x,1);

on (supL) L = A ;
on (diagU) U = A / (on(supL) L).transrel(x,1) ;

for (i=2;i<=N;i++) {
  supL <- supL . moverel(1,1);
  on (supL) L = A - infA *(on(diagU)U).transrel(y,1);
  diagU <- diagU . moverel(1,1);
  on (diagU) U = A / (on(supL)L).transrel(x,1);
}

/***** Descente sur L *****/
dpo void LUspace [d=1] diag;
float pred;

on(diag) Y = V / L;
pred = Y.scalar();

for (i=2;i<=N;i++) {
  diag <- diag.moverel(1,1,0);
  on (diag) Y = (V- infA * pred)/L;
  pred = Y.scalar();
}

/***** Remontée sur U *****/
dpo float LUspace [x=2,d=*] supU;
/* une copie de la diagonale supérieure de U ramenée sur la diag */
supU <- (on(supU)U).transrel(x,-1);

on (diag) X = Y;

for (i=N-1;i>0;i++) {
  diag <- diag.moverel(-1,-1);
  on (diag) X = Y - supU * pred;
  pred = X.scalar();
}

```

FIG. VI.7 - La décomposition LU en C-HELP

Néanmoins, la méthode peut devenir intéressante pour la résolution de multiples systèmes tri-diagonaux reposants sur la même matrice (ce qui peut se produire pour des simulations physiques). Le calcul préliminaire (décomposition LU) est toujours effectué en séquentiel, mais la résolution peut se faire en parallèle pour chaque système à résoudre. C'est dans ce cas que cet algorithme est utilisé dans le calcul vectoriel [CT93], la descente et la remontée sur les matrices L et U se traduisant par des opérations triadiques qui sont indispensables à une architecture vectorielle pour atteindre de très bonnes performances.

Pour écrire en C-HELP cette phase d'exploitation du parallélisme, l'hyper-espace est étendu à trois dimensions. Chaque plan représente maintenant un système à résoudre. La première partie de l'algorithme se fait toujours sur le plan origine et les matrices intermédiaires sont dupliquées le long des plans de chaque système d'équations. L'expression du parallélisme revient donc dans le modèle géométrique à étendre le domaine de conformité à un ensemble de plans homogènes.

2.4 Le gradient conjugué

Le gradient conjugué est une méthode de résolution approximative de problèmes d'algèbre linéaire. Alors qu'un algorithme comme ceux présentés précédemment donnent une solution exacte au système d'équations, le gradient conjugué est une méthode dans laquelle un processus simple est réitéré un certain nombre de fois, jusqu'à l'obtention d'une solution « convenable ». La convergence de l'algorithme conditionne l'arrêt du programme [WD94].

Pour accélérer cette convergence, un traitement est appliqué à la matrice de départ, on l'appelle pré-conditionnement. C'est donc sur un algorithme de gradient pré-conjugué que nous illustrons la programmation C-HELP sur des méthodes itératives.

Le code C-HELP est développé à partir de briques de base comme le produit scalaire et le produit matrice/vecteur. Tant que la convergence n'est pas atteinte, le traitement est réitéré. Voir le programme développé en collaboration avec Christine Weill-Duflos en figure VI.8.

```

hspace plan [x=N,y=N,d=(x,y)];

/***** produit scalaire *****/
float prodscal(dpo float plan U, dpo float plan V) {
  return (U*V).ReduceAdd(x).scalar(1,1);
}

/***** produit matrice/vecteur *****/
dpo float plan matvec(dpo float plan A, dpo float plan V) {
  return (A*V.expand(x)).ReduceAdd(y).exchange(x,y);
}

/***** polynome *****/
dpo float plan poly(dpo float plan M, dpo float plan V, int m) {
  dpo float plan [x=1,y=*] W;
  dpo float plan [d=*] D;

  on (D) D=M;
  W=V;
  for (i=0;i<m;i++) W += V - matvec(M,W) / D.exchangeabs(d,y);
  return W/D.exchangeabs(d,y);
}

/***** gradient conjugué préconditionné *****/
dpo float plan pcg(dpo float plan M, /* [x=*,y=*] */
                  dpo float plan x0, /* [x=1,y=*] */
                  dpo float plan b, /* idem(x0) */
                  int m, float epsilon) {
  dpo float plan [1,*] xres,v,r;
  float alpha,beta,rho,tmp,eps;

  iter=0; xres=x0;
  r=poly(M,b-matvec(M,x0),m);
  rho=prodscal(r,r); v=r; eps=epsilon*rho;
  do {
    W=poly(M,matvec(M,v),m);
    alpha=rho/prodscal(W,v);
    xres+=alpha*v;
    beta=prodscal(r,r)/rho;
    rho=tmp;
    v=r + beta * v;
  } while(rho > epsilon);
  return xres;
}

```

FIG. VI.8 - Le gradient conjugué préconditionné en C-HELP

3 Squelettisation d'une image

LE TRAITEMENT d'image en vue de leur interprétation est un domaine d'application important quant aux retombées possibles dans l'industrie. Souvent, l'objectif à atteindre est la réalisation d'un système temps-réel permettant le contrôle de processus divers (conduite automobile, reconnaissance automatique d'écriture, productique...). La reconnaissance des formes d'objets contenus dans une image passe souvent par une phase préliminaire de « squelettisation » de l'image pour extraire, à partir d'une zone coloriée, un squelette plus facile à reconnaître. C'est cette phase que nous proposons d'écrire en C-HELP.

L'algorithme mis en place est décrit dans [Ols92]. Pour changer la valeur d'un pixel, il faut que son voisinage satisfasse une condition calculée dans la fonction microscopique `new` (nombre de voisins noirs,...). Pour des problèmes de fuite de pixels, l'appel à cette fonction ne peut se faire sur tous les pixels simultanément : il faut, pour chaque étape de calcul effectuer quatre phases pour ne pas traiter des voisins immédiats simultanément. Pour cela, on masque l'appel à la fonction microscopique par un masque représentant l'ordre d'appel. Voir programme en figure VI.9.

```

hspace plan [x=N,y=N];

dpo unsigned char plan [*,*] pixel,change,etape;

micro new(int c,int cN, int cNE, int cE,
          int cSE, int cS, int cSW, int cW, int cNW) {
    if (/* calcul de la condition */)
        { change=1; return 1; }
    else { change=0; return 0; }
}

main()
{
    int i;
    etape=0;                               /* 0 1 0 1 */
    where (pixel.cabs(x)%2!=0) etape++;    /* 2 3 2 3 */
    where (pixel.cabs(y)%2!=0) etape+=2;  /* 0 1 0 1 */
                                         /* 2 3 2 3 */

    change=1;
    do {
        for (i=0;i<4;i++)
            where (etape==i)
                pixel=new(pixel,pixel.shifttor(y,-1),...);
    }while (change.ReduceOr(x).ReduceOr(y).scalar(1,1)!=0);
}

```

FIG. VI.9 - La squelettisation d'une image en C-HELP

4 Un modèle hydro-dynamique

DES CONTACTS avec le « Laboratoire Hydrodynamique et Sédimentologie » de l'IFREMER² ont été entrepris dans le but d'étudier l'adéquation des machines massivement parallèles à la résolution de systèmes linéaires modélisant ce domaine de la mécanique des fluides.

Avec ce type de problèmes réels, nous montrons que le modèle HELP et le langage C-HELP peuvent amener le physicien à programmer plus facilement et plus lisiblement ses applications.

Nous exposons dans un premier temps le modèle considéré à partir des équations de la mécanique des fluides [RK80]. Nous présentons ensuite la discrétisation effectuée et ses suppositions concernant les approximations introduites. Nous proposerons le programme C-HELP qui résout ce modèle. Ce code source permettra de mettre en valeur des concepts du modèle géométrique qui facilitent le développement.

4.1 Le modèle physique

La marée est le processus physique le plus important le long des côtes françaises (Manche et Atlantique). Elle est constituée d'une somme d'ondes astronomiques semi-diurnes (périodes d'environ 12 heures) générées au large et se propageant sur le plateau continental. Les variations de niveaux s'accompagnent de courants alternatifs parfois violents (jusqu'à 3 m/s en certains endroits de la Manche). Pendant très longtemps, seule la mesure en mer (chère et compliquée) permettait de connaître les courants. Depuis les années 60, les modèles numériques se développent et ont atteint maintenant une grande fiabilité.

Pour traiter les phénomènes de marée, la majorité de ces modèles sont de type « bi-dimensionnel horizontal » (2DH). Ce type de modèle suppose implicitement que les courants sont à peu près constants de la surface au fond. C'est le cas pour les courants de marée, mais cette supposition peut devenir erronée pour les courants induits par le vent en surface.

4.1.1 Les équations de la mécanique des fluides

Suivant la supposition 2DH, les équations générales de la mécanique des fluides (Navier-Stokes) sont intégrées sur la verticale. On obtient alors un système de 3 équations (2 pour le mouvement et 1 pour la continuité) à 3 inconnues (\bar{U} vitesse Est-Ouest, \bar{V} vitesse Nord-Sud et ζ hauteur d'eau).

Équations du mouvement :

2. Institut Français de la Recherche et de l'Exploitation de la Mer

$$\frac{\partial U}{\partial t} + \underbrace{U \frac{\partial V}{\partial x} + V \frac{\partial U}{\partial y}}_{\text{termes non-linéaires}} - \underbrace{fV}_{\text{Coriolis}} = \underbrace{-g \frac{\partial \zeta}{\partial x}}_{\text{Gradient de pression}} + \frac{\tau_{\delta x} - \tau_{fx}}{\rho H} + N_H \nabla_H^2 U$$

$$\frac{\partial V}{\partial t} + \underbrace{U \frac{\partial V}{\partial x} + V \frac{\partial U}{\partial y}}_{\text{termes non-linéaires}} + \underbrace{fU}_{\text{Coriolis}} = \underbrace{-g \frac{\partial \zeta}{\partial y}}_{\text{Gradient de pression}} + \frac{\tau_{\delta y} - \tau_{fy}}{\rho H} + N_H \nabla_H^2 V$$

Équation de continuité :

$$\frac{\partial \zeta}{\partial V} + \frac{\partial HU}{\partial x} + \frac{\partial HV}{\partial y} = 0$$

dans lesquelles :

\bar{U} représente le courant moyen :

$$\bar{U} = \frac{1}{H} \int_H U dz$$

f : paramètre de Coriolis (à nos latitudes $10^{-4} s^{-1}$)

g : accélération de la pesanteur ($9.81 m s^{-2}$)

ρ : densité du milieu

H : hauteur d'eau

N_H : viscosité turbulente horizontale

$(\tau_{\delta x}, \tau_{\delta y})$: tension du vent en surface

$(\tau_{fx}, \tau_{fy}) = C_d \|U\| (U, V)$: tension de frottement sur le fond

$\|U\| = \sqrt{U^2 + V^2}$: vitesse du courant

4.1.2 La discrétisation

Il existe plusieurs façons de résoudre les équations que nous venons de décrire. Deux types de modèles sont distingués :

- les modèles dits *explicites* qui calculent une variable à un instant donné en fonction des autres variables au pas de temps précédent ;

- les modèles dits *implicites* qui calculent une variable à un instant donné en fonctions des autres variables au même instant.

Le choix de l'une de ces deux solutions est important pour le développement de l'application. Du point de vue de la facilité de programmation, le modèle explicite est largement supérieur au modèle implicite, qui conduit à résoudre d'importants systèmes linéaires, et en particulier à des inversions de matrices ou des méthodes itératives.

Du point de vue de la signification physique des calculs, le modèle implicite est dit *inconditionnellement stable*, c'est-à-dire qu'il supporte une définition d'un pas de temps très grand. Par contre les modèles explicites possèdent des critères de stabilité contraignants; pour obtenir un bon résultat, il faut conserver l'inéquation :

$$\delta t < \frac{\delta x}{c}$$

- δt pas de temps
- δx pas d'espace, grain de discrétisation
- $c = \sqrt{gH}$: vitesse de propagation des ondes de surface

Pour effectuer la résolution par un modèle explicite, on est donc obligé de réduire le pas de temps d'un facteur linéaire par rapport au pas d'espace (pour une maille de 1km et une profondeur de 100m, $\delta t < 30sec$). Pour un même maillage, quand le modèle implicite calcule une itération, le modèle explicite doit calculer n itérations (avec n le grain de discrétisation).

En considérant que pour une machine parallèle la complexité de l'inversion de matrice est d'ordre n (ce qui néglige les communications introduites par l'algorithme d'inversion³, cf. exemple précédent), pour un temps de simulation global égal la complexité des deux solutions est équivalente : pour le modèle implicite, le calcul demandera n fois plus de temps et pour le modèle explicite, le pas de temps devra être divisé par n .

Par contre, pour une même complexité, la solution du modèle explicite fournit une résolution des n pas de l'algorithme tandis que le modèle implicite ne calcule que l'état du système après le temps de simulation. Le modèle explicite donne donc un résultat plus précis pour une simulation de même temps théorique d'exécution. De plus, dans la pratique, l'inversion de matrice n'est pas de complexité linéaire par rapport au pas de temps, ce qui donne une résolution plus rapide pour le modèle explicite. C'est donc ce modèle que nous avons choisi.

3. Le nombre d'étapes de communications est en $n \log(n)$.

4.2 La programmation par le modèle géométrique

Le modèle utilisé est explicite et les variables sont calculées sur une maille décalée de type « C » très couramment utilisée en océanographie côtière. Le décalage d'un demi pas d'espace entre les trois variables à calculer permet de réduire l'étendue des convolutions à calculer, et par conséquent les communications (cf. figure VI.10).

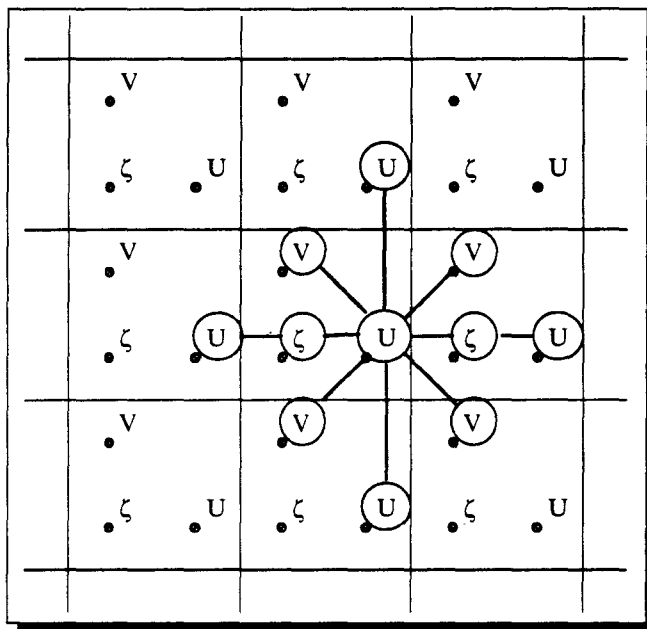


FIG. VI.10 - Le calcul de U pour une maille de l'espace.

Le modèle étant 2DH, le programme C-HELP correspondant sera écrit dans le cadre d'un hyper-espace bi-dimensionnel. L'algorithme consiste ensuite à effectuer des convolutions entre les éléments d'une maille et ses voisins, à chaque pas de temps. L'évaluation des variables se fait en deux étapes : horizontalement (calcul de U) puis verticalement (calcul de V). Pour des critères de stabilité, le pas de temps est divisé par deux et à chaque itération, on calcule deux fois la vitesse horizontale et la vitesse verticale puis les conditions aux limites et enfin, la hauteur d'eau.

Pour effectuer les communications, on utilise naturellement les primitives géométriques du langage C-HELP, qui permettent d'éviter ainsi les descriptions d'objets par les indices. Pour l'écriture des fonctions calculatoires, la vue microscopique nous permet d'écrire le traitement d'une cellule en ne considérant qu'une vision locale (on se débarrasse de tous les indices qui alourdissent les expressions mathématiques⁴). Le code s'en trouve d'autant plus lisible.

4. dans l'exemple qui va suivre, la suppression des indices permet de réduire le nombre de parenthèses de l'expression de 66 %.

De plus, les calculs ne doivent être effectués que sur des parties « centrales » du maillage. En effet, le fonctionnement du modèle est issu de la donnée des conditions aux limites (pour un modèle océanographique côtier, on considère que la marée n'est pas générée à l'intérieur du domaine de simulation). Par exemple, la condition aux côtes consiste à imposer que la vitesse perpendiculaire à la côte soit nulle. Pour le calcul d'une variable, le domaine est donc toujours réduit à un rectangle qui évite certains calculs aux frontières. Pour chaque étape de calcul, le domaine de conformité sera donc explicité par le constructeur `on` et les différents domaines seront donnés par des DPO de type `void` qui définiront les cadres géométriques de calculs (cf. figure VI.11).

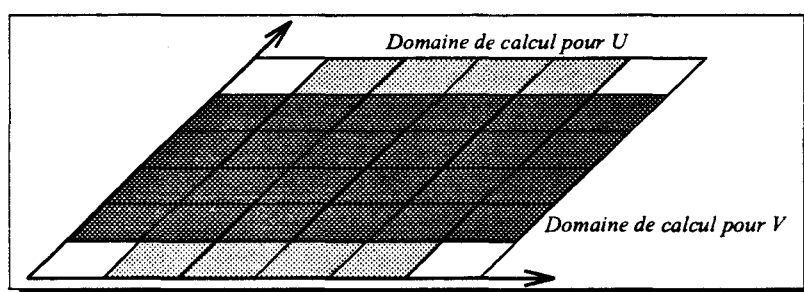


FIG. VI.11 - Les domaines de conformité par demi pas de temps

Le modèle écrit en Fortran 77 comporte environ 700 lignes. Nous ne reproduirons donc qu'une partie correspondante au calcul d'une variable.

```

c.....calcul de XE

      do 200 j=jmin+2,jmax-1
      do 200 i=imin+2,imax-1

      xe4(i,j)=xe2(i,j)-dt*(
1         usdx*(
2         (hx(i,j)+0.5*(xe2(i,j)+xe2(i+1,j)))*u3(i,j)
3         -(hx(i-1,j)+0.5*(xe2(i,j)+xe2(i-1,j)))*u3(i-1,j) )
4         +usdx*(
5         (hy(i,j)+0.5*(xe2(i,j)+xe2(i,j+1)))*v3(i,j)
6         -(hy(i,j-1)+0.5*(xe2(i,j)+xe2(i,j-1)))*v3(i,j-1) ) )
200      continue
    
```

FIG. VI.12 - Du Fortran...

```

#define dt      60
#define dx      200

hspace LaMer [x=N,y=N] (x,y);

#define WES     transrel(x,-1)
#define EAS     transrel(x,1)
#define NOR     transrel(y,+1)
#define SOU     transrel(y,-1)
...
dpo double LaMer [x=*,y=*] hx,hy;
dpo void LaMer [x=imin+2:imax-1,y=jmin+2:jmax-1] domXE;
...

on(domXE) {
    xe4=xe2-dt*(
        (hx+(xe2+xe2.EAS)/2)*u3
        -(hx.WES+(xe2+xe2.WES)/2)*u3.WES
        +(hy+(xe2+xe2.NOR)/2)*v3
        -(hy.SOU+(xe2+xe2.SOU)/2)*v3.SOU
    )/dx;
}

```

FIG. VI.13 - ...à C-HELP

5 Conclusion

C E CHAPITRE consacré à la programmation d'exemples caractéristiques ou réels nous a permis de mettre en application les propositions exposées au long de cette thèse.

Le premier exemple illustre la notion de pensée géométrique qui constitue un cadre d'algorithmique pouvant aboutir à l'expression simple des programmes.

Le second type d'exemples montre l'adéquation forte du modèle HELP et de l'algorithmique numérique matricielle. L'inversion de matrice par l'algorithme de Gauss-Jordan a été transcrite à partir des trois étapes géométriques qui la composent. D'autres exemples de calculs matriciels ont été exposés et ont permis de montrer que le langage C-HELP intègre dans sa sémantique et sa syntaxe le problème parfois difficile de l'expression du placement et de l'alignement des données.

Enfin, les programmes de traitement d'image et d'océanographie côtière nous ont permis de sortir du cadre restrictif des exemples purement « académiques ». Un programme écrit avec le langage C-HELP est plus lisible que son équivalent Fortran, notamment par la séparation introduite entre les cadres d'exécution des calculs et les expressions décrivant ces mêmes calculs.

Conclusion

1 Résumé des travaux

LE PARALLÉLISME de données est aujourd'hui reconnu comme étant le modèle privilégié de programmation candidat au Téra-flops. Nous avons présenté l'état actuel des travaux dans ce domaine basé sur ce modèle simple et facile à programmer.

Par la volonté de proposer un environnement proche de l'algorithmique intensive, nous avons défini ce qui pourrait constituer un modèle de programmation: la **géométrie** qui joue le rôle de fil conducteur entre toutes les phases de développement d'une application, de l'algorithme à l'exécution.

Pour caractériser plus avant ce modèle géométrique, nous avons défini le **modèle HELP** qui se base sur la notion d'hyper-espace, référentiel incontournable de toute opération. Les objets manipulés sont alloués dans cet hyper-espace et les traitements calculatoires sont séparés des communications par l'adoption d'une règle forte de conformité. Le modèle ainsi obtenu est cohérent: le programmeur explicite des informations d'alignement entre les objets qui interagissent, et la règle de conformité permet d'assurer que cet alignement peut être, et sera, exploité pour l'obtention de bonnes performances à l'exécution. Le langage C-HELP a été défini. Il intègre dans le langage C les concepts géométriques de HELP.

Nous avons montré que le langage était compilable (et partiellement compilé) et que le modèle sous-jacent permet effectivement des optimisations dans les techniques de générations de code. En particulier, nous avons décrit l'allocation mémoire qui permet de diminuer sensiblement les temps d'accès mémoire par une factorisation du calculs des adresses des opérandes. Nous avons aussi décrit un algorithme d'allocation mémoire dynamique, lui-même parallèle. Enfin, nous avons proposé de gérer la boucle de virtualisation indépendamment sur chaque processeur physique, réduisant ainsi le nombre d'itérations nécessaire au traitement d'un domaine de l'hyper-espace, et par conséquent le temps total d'exécution d'un programme. Ces optimisations ont été illustrées par des mesures de performances.

Le modèle HELP a été ensuite proposé pour la gestion des structures creuses. Notre

ambition est large : manipuler du creux sans manipuler la compression du creux. Dans ce sens, nous avons étudié la possibilité d'intégrer totalement la compression des données à l'intérieur du compilateur, en gardant tous les concepts géométriques qui font l'efficacité et la simplicité du modèle HELP.

Quelques exemples ont été détaillés. HELP permet bien de programmer suivant le modèle à parallélisme de données.

2 Perspectives

DE PAR l'utilisation potentielle de plusieurs hyper-espaces dans une application, nous avons montré que le langage pouvait aboutir à une méthode d'exécution hétérogène. Un axe de nouvelles recherches est alors envisageable, pour tenter de résoudre l'importante problématique de la définition d'une méthodologie de programmation hétérogène.

Comme nous l'avons présenté au deuxième chapitre, certains langages existants proposent la gestion de structures non régulières. À la vue de l'algorithmique général, il est en effet nécessaire de se pencher sur les structures moins régulières que les polyèdres particuliers proposés dans HELP, pour fournir au programmeur un outils souple qui s'adapte à la configuration de ses données.

Le modèle géométrique doit être étendu à des familles d'objets moins limitatives. Quelques idées peuvent dès aujourd'hui être émises : donner la possibilité de définir des objets géométriques plus complexes, définir des opérations ensemblistes qui permettent l'union et l'intersection de deux domaines.

Nous proposerons ensuite une réflexion sur la possibilité de compléter la vue microscopique des traitements par l'introduction de rééquilibrage dynamique qui pourrait amener de meilleurs performances.

Enfin, nous poserons le problème de l'élargissement de la portée des concepts de base du modèle HELP à des langages existants, pour éviter de confiner le modèle géométrique au langage spécifiquement dédié C-HELP.

2.1 Des objets plus généraux

La définition d'axes secondaires permet la manipulation de diagonales dans l'hyper-espace (cf. figure VI.14). On peut déjà entrevoir un problème : certains DPO peuvent avoir le même domaine d'allocation alors que leurs déclarations diffèrent.

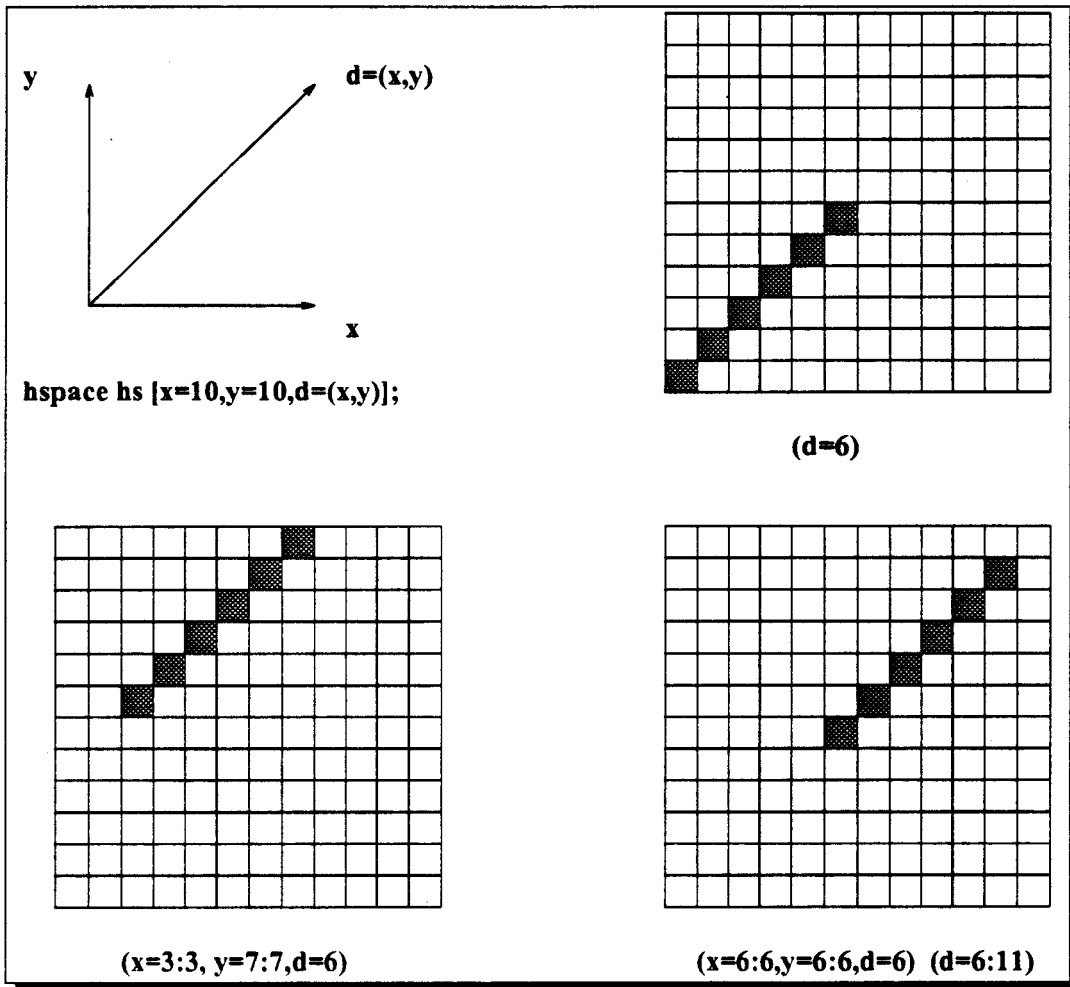


FIG. VI.14 - Les DPO alloués sur une diagonale

Nous introduisons maintenant la possibilité de déclarer des objets comportant une largeur sur une dimension secondaire et une largeur sur les dimensions primaires de construction de la dimension secondaire (cf. figure VI.15). En dehors des problèmes de compilation, cette autorisation permet au programmeur de travailler sur des objets de forme plus générale.

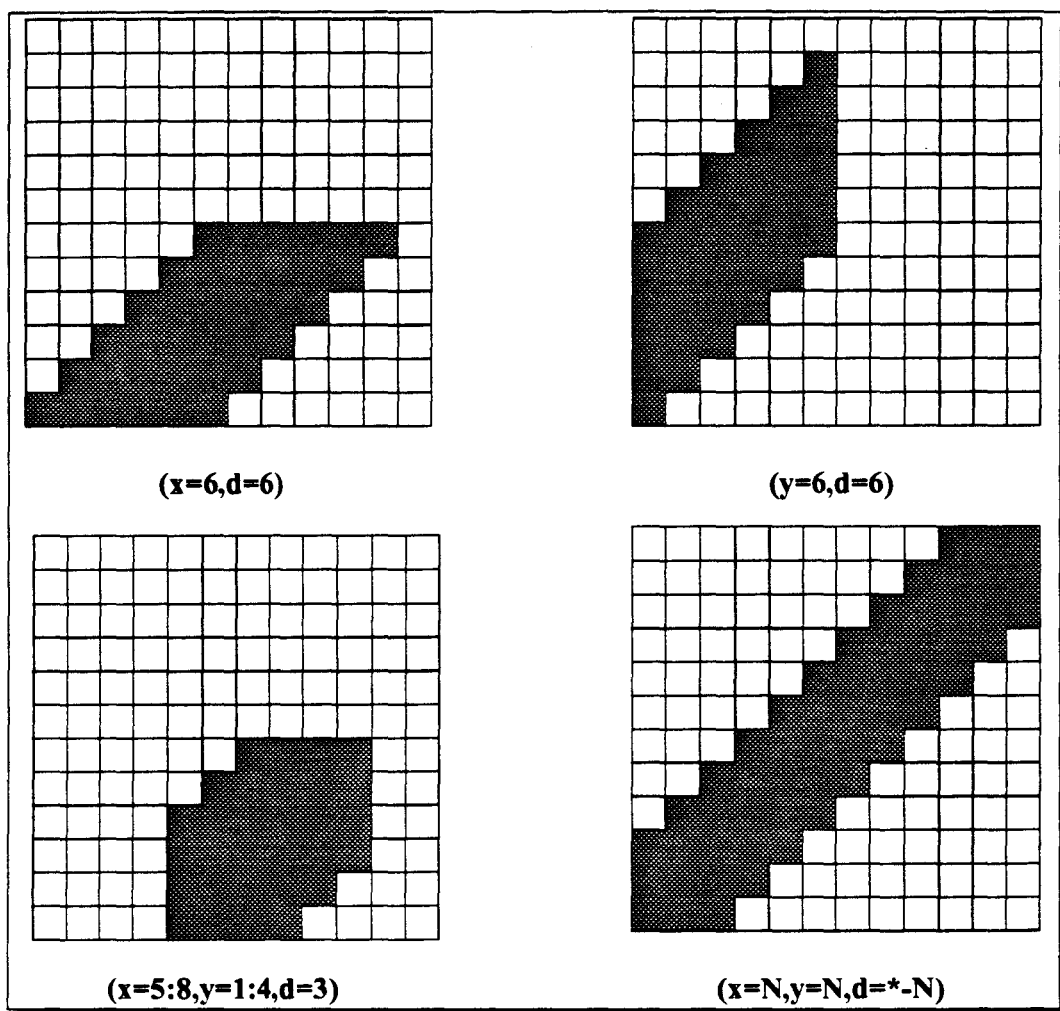


FIG. VI.15 - Les DPO parallélogrammoïdaux

La définition de dimensions anti-diagonales élargira les possibilités de manipulations d'objets dans l'hyper-espace. Nous avons appelé « anti-diagonale » une diagonale secondaire de l'hyper-espace. Au delà de la syntaxe de déclaration (qui mérite sûrement une adaptation), il est intéressant de porter un regard sur les nouveaux objets manipulables (cf. figure VI.16).

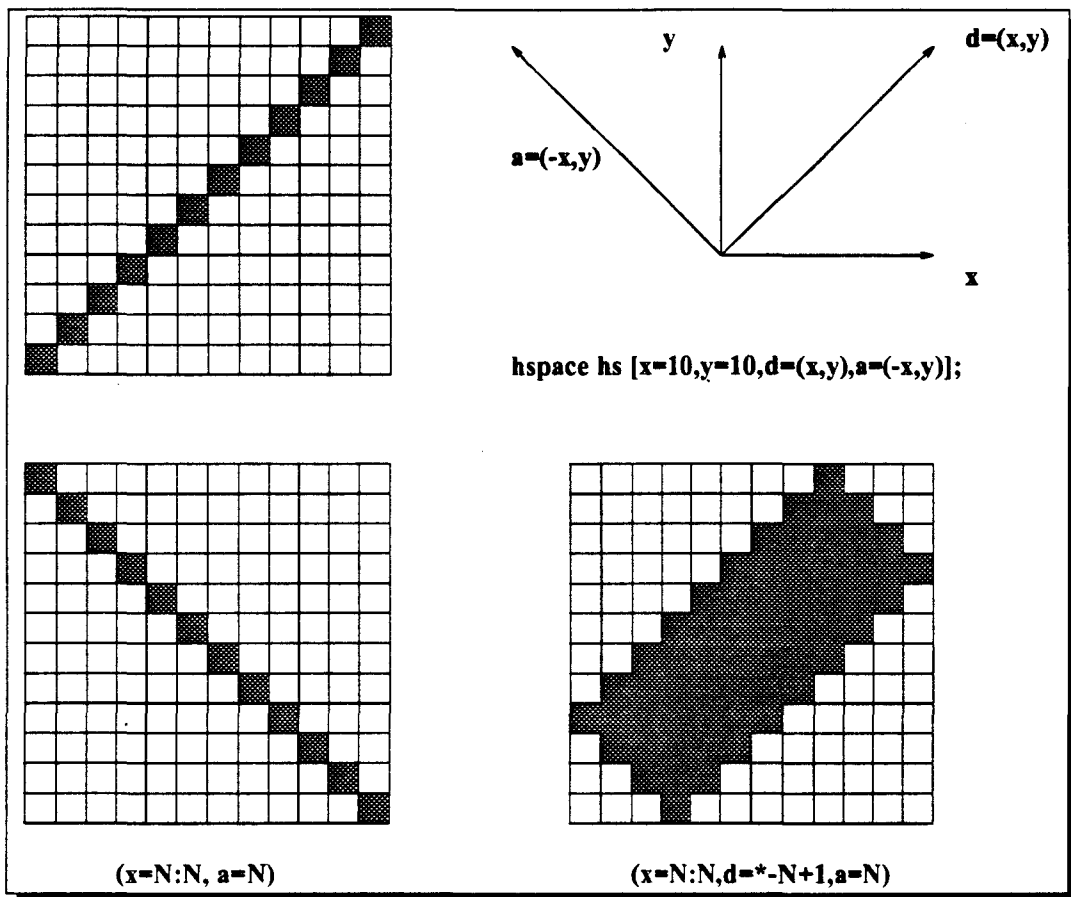


FIG. VI.16 - Les DPO anti-diagonaux

En combinant les axes secondaires anti-diagonaux, et la possibilité de donner une largeur à un DPO déclaré sur une dimension secondaire, on peut déclarer des DPO de formes variées (cf. figure VI.17).

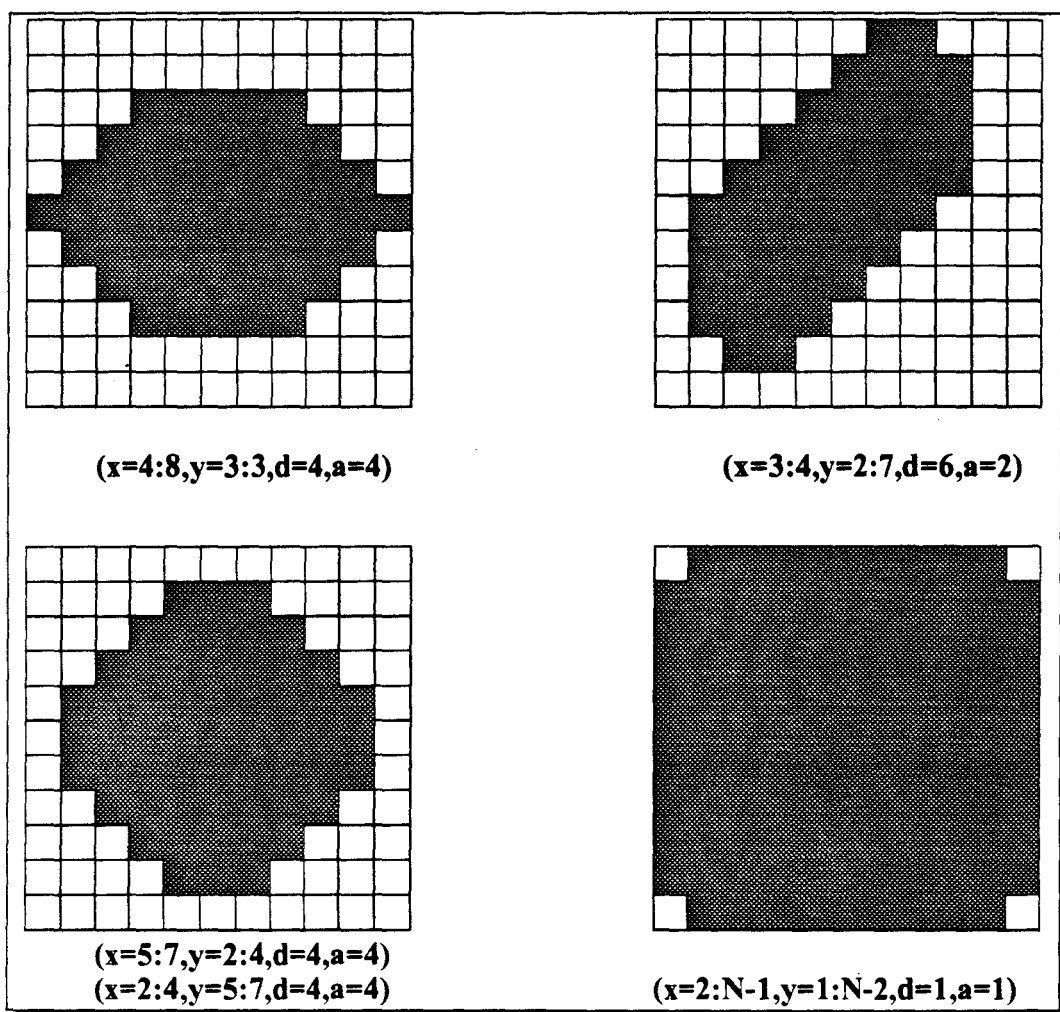


FIG. VI.17 - Les DPO hexagonaux et octogonaux

Manipuler un objet de forme hexagonale ou octogonale n'est pas *a priori* très utile, mais le dernier exemple proposé à la figure VI.17 montre qu'il est ainsi possible de donner une géométrie à un DPO de telle sorte que ce DPO définisse une matrice à laquelle on soustrait les éléments en coins. Ce genre d'objet est utile pour des calculs de voisinage sur un domaine dont les frontières sont souvent traitées séparément.

2.2 Des opérations ensemblistes

Manipuler des objets non convexes peut être envisagé avec les opérations ensemblistes que l'on introduirait au niveau du constructeur `on`. En notant par exemple `and` l'intersection de deux domaines et `or` l'union de deux domaines, nous pouvons construire des expressions entre domaines d'allocation qui amènerait à opérer des traitements sur des domaines plus généraux

comme des triangles⁵, ou des domaines non-convexes comme le cadre d'une matrice (union des première et dernière ligne avec les première et dernière colonne).

Cette possibilité est sûrement un pas important vers l'accroissement de l'expressivité du langage, mais elle pose des problèmes de génération de code, en particulier de gestion de la géométrie de l'activité de la boucle de virtualisation.

2.3 Des polyèdres convexes

La manipulation de polyèdres convexes est souvent étudiée dans le cadre des études de parallélisation de programmes de la famille de Fortran. La donnée, au niveau du langage, d'un tel objet doit être traitée par le compilateur pour retrouver un itérateur permettant la génération du code de balayage de cet objet. Trouver cet itérateur revient à effectuer un changement de référentiel pour retrouver les frontières de l'objet considéré comme limites des variables d'itération du domaine balayé.

L'introduction de tels objets dans le modèle et le langage C-HELP peut amener une solution plus générale que les objets construits sur des diagonales ou anti-diagonales. Néanmoins, elle pose le problème de la génération de la boucle de virtualisation. La reprise dans les travaux existants du mécanisme de changement de référentiel nous amène à perdre le balayage géométrique liée directement à l'hyper-espace. Or, par exemple lors de l'évaluation d'une expression macroscopique, chaque point activé pour une itération de cette boucle doit connaître ses coordonnées dans l'hyper-espace. Bien que cette information puissent être reconstruite, cette modification risque de pénaliser les performances du code produit.

De plus, la volonté de ne générer que des communications explicites nous interdit la redistribution d'itérations, phase qui succède souvent au changement de référentiel. Le balayage de l'objet dans le nouveau référentiel peut alors amener à activer successivement deux points du même processeur. La parallélisation de ce balayage doit donc faire intervenir les informations de distribution des points dans l'hyper-espace, qui ne sont pas exprimées dans le nouveau référentiel. Une conversion sera donc nécessaire sur un des deux types de ces informations.

Ces polyèdres convexes sont donc utiles dans le langage mais ne correspondent pas immédiatement aux techniques de compilation définies au chapitre 4. L'adaptation de ces objets doit néanmoins être l'objet d'une étude approfondie.

5. intersection d'un parallélogramme et d'un rectangle

2.4 Vue microscopique et rééquilibrage

Nous avons montré que la vue microscopique consistait à évaluer un segment conforme dans lequel toutes les opérandes se trouvent au niveau du point. Cette évaluation ne se fait donc que sur les points actifs, sans aucune communication entre eux. Or, le nombre de points actifs par processeur détermine directement le nombre d'itérations nécessaires à la boucle de virtualisation et donc le temps d'exécution pour l'instruction.

On introduit alors un mécanisme de rééquilibrage des données pour l'évaluation d'un (ou plusieurs?) segment conforme. Avec une telle fonctionnalité, le compilateur générerait le code nécessaire au rééquilibrage sur le réseau physique des points actifs pour l'évaluation du segment conforme [Fon94].

La décision d'opérer un rééquilibrage dépend du rapport entre le temps gagné par le rééquilibrage et le temps demandés par les communications. Or, ce rapport est fortement dépendant de l'algorithme mis en place. Il paraît fort improbable de pouvoir déterminer statiquement (à la compilation) si un rééquilibrage est « rentable » pour une opération microscopique. Il faudra donc proposer de laisser le programmeur libre de décider si le rééquilibrage doit être déclenché ou si l'évaluation doit conserver la même distribution des opérandes.

Pour ce faire, le langage est complété avec un nouveau constructeur **zoom** dont la sémantique est exactement identique à celle du constructeur **on**. En entrée d'un **zoom**, le rééquilibrage est effectué, et en fin de traitement, les valeurs résultantes sont rangées sur les points de départ. On a ainsi un parallélisme maximal.

On retrouve avec ce constructeur **zoom** le système de compilation par distribution des itérations (comme le fait MP-FORTRAN pour l'instruction **forall**).

2.5 Extension de la vue macroscopique

La vue macroscopique est constituée d'une batterie de primitives géométriques que nous avons présentées au chapitre 3. La généralisation des possibilités de migration peut s'envisager en laissant au programmeur la possibilité d'explicitier dans son programme les fonctions d'association source/cibles qui modélisent une primitive macroscopique.

Avec l'écriture de telles fonctions, le compilateur est capable de générer le code correspondant par l'évaluation de cette fonction sur les coordonnées de chaque point source. On peut alors disposer d'un éventail infini de primitives.

2.6 Généralisation

Le modèle HELP a pour l'instant été intégré à un langage spécifiquement développé dans ce but : C-HELP. Pourtant, le modèle géométrique n'est pas spécifique à un langage mais peut constituer un modèle de programmation réutilisable dans d'autres langages. Des travaux dans notre équipe [DM94] vont en ce sens : comment définir la conception d'une application suivant le modèle géométrique dans d'autres langages ? Quel langage utiliser pour permettre l'adoption du modèle géométrique en ne définissant pas de nouvelles extensions du langage ? Les langages objets permettent-ils de définir certaines classes d'objets pouvant modéliser la géométrie introduite dans HELP, permettant ainsi de suivre le modèle sans utiliser de langage spécifique ?

Annexe A

Programmes de test

1 Nombre de références dans une expression

Programmes de tests pour l'évaluation de l'influence du modèle basé sur le référentiel qui permet la factorisation des calculs d'adresses.

1.1 Programme C-HELP

```
hspace plan [x=100,y=100];

void test(dpo int plan A, dpo int plan B, dpo int plan C
          , int xl, int xu, int yl, int yu) {
  dpo void plan [x=xl:xu,y=yl:yu] Dom;
  INITTIMER();
  on (Dom) {
    A=0;
    B=0;
    C=0;
  }
  WRITETIMER();
}

main(){
  dpo int plan [x=*,y=*] A,B,C;
  test(A,B,C,50,50,50,50);
}
```


1.2 Programme MP-Fortran

```
      subroutine F00(A,B,C,XL,XU,YL,YU)

      cmpf    mpl INITTIMER
      cmpf    mpl WRITETIMER

      CMPF ONDPU      A
      CMPF ONDPU      B
      CMPF ONDPU      C

      integer XL,XU,YL,YU
      integer A(:, :)
      integer B(:, :)
      integer C(:, :)

      call INITTIMER()
      A(XL:XU,YL:YU)=0
      B(XL:XU,YL:YU)=0
      C(XL:XU,YL:YU)=0
      call WRITETIMER()
end

      program test

      integer A(100,100)
      integer B(100,100)
      integer B(100,100)

      CMPF ONDPU      A
      CMPF ONDPU      B
      CMPF ONDPU      C

      call F00(A,B,C,50,50,50,50)

end
```

2 Virtualisation

Programme de test pour l'évaluation du gain de l'introduction d'irrégularité dans la boucle de virtualisation.

2.1 Programme C-HELP

```
hspace plan [x=300,y=300];

void test(int xl, int xu, int yl, int yu) {
    dpo int plan [x=*,y=*] M;
    M <- M.extrabs(x,xl..xu).extrabs(y,yl..yu);
    INITTIMER();
    M=0;
    WRITETIMER();
}

main(){
    int i,j;
    for (i=1;i<=201;i+=25)
        for (j=1;j<201;j+=25)
            test(i,i+49,j,j+49);
}
```

2.2 Programme MP-Fortran

```
      subroutine TEST(XL,XU,YL,YU)
         integer XL,XU,YL,YU
         integer A(300,300)
CMPF ONDPU A
         call INITTIMER()
         A(XL:XU,YL:YU)=0
         call WRITETIMER()
      end

      program test
      do i=1,201,25
         do j=1,201,25
            call TEST(I,I+49,J,J+49)
         end do
      end do
      end
```

Bibliographie

- [AKMS85] E. Arnould, H.T. Kung, O. Menzilcioglu, and K. Sarocky. A systolic array computer. In *IEEE Int. Conf. ICASSP'85*, pages 232–235, Tampa (FL-USA), 1985.
- [Alp93a] Special issue on the Digital's Alpha chip project. *Communications of the ACM*, 36(2), February 1993.
- [Alp93b] Digital Equipment Corp. *Alpha Architecture Handbook*, 1993.
- [Bac78] John Backus. The history of Fortran I, II, and III. In *proceedings of the ACM SIGPLAN History of Programming Languages Conference*, pages 165–180, June 1978.
- [BB92] Jonathan D. Becher and Kent L. Beck. Profiling on a massively parallel computer. In *Proc. Conpar 92/Vapp V*, pages 97–102, Lyon, France, September 1992. Lecture Notes in Computer Science, vol 634.
- [BBDS94] David H. Bailey, Eric Barszcz, Leonardo Dagum, and Horst D. Simon. Nas parallel benchmark results 3-94. Technical report, NASA Ames Reseach Center, March 1994.
- [BBJ⁺62] J.R. Ball, R.C. Bollinger, T.A. Jeeves, R.C. McReynolds, and D.H. Shaffer. On the use of the solomon parallel-processing computer. In *proceedings of the Fall 1962 Eastern Joint Computer Conference*, pages 137–146, December 1962.
- [BBK⁺68] George H. Barnes, Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Slotnick, and Richard A. Stokes. The Illiac IV computer. *IEEE Transactions on Computers*, 17(8):746–757, August 1968.
- [BDLM92a] Akram Benalia, Jean-Luc Dekeyser, Dominique Lazure, and Philippe Marquet. HelpDraw: An interactive graphical environment for data parallel programming. In *CNI/MAT'92 International Workshop*, Nancy, France, December 1992.
- [BDLM92b] Akram Benalia, Jean-Luc Dekeyser, Dominique Lazure, and Philippe Marquet. HELP for data-parallel scientific programming. In *Supercomputing '92 (poster session)*, Minneapolis, Minnesota, November 1992.

BIBLIOGRAPHIE

- [BDM93] Akram Benalia, Jean-Luc Dekeyser, and Philippe Marquet. HelpDraw graphical environment: A step beyond data parallel programming languages. In *Fifth Int'l Conf. on Human-Computer Interaction*, pages 591–596, Orlando, FL, August 1993. Elsevier Science Publishers.
- [Ben95] Akram Djellal Benalia. *HelpDraw: Un Environnement Visuel pour la Génération Automatique de Programmes à Parallélisme de Données*. Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, February 1995. (en préparation).
- [Bla90] Tom Blank. The MasPar MP-1 architecture. In *Proceedings of the IEEE Compcon Spring 1990*, pages 20–24, San Francisco, CA, February 1990. IEEE Society Press.
- [BLVU94] Luc Bougé, Yann Le Guyadec, Bernard Virost, and Gil Utard. On the expressivity of a weakest preconditions calculus for a simple data-parallel programming language. In *International Conference on Parallel and Vector Processing ConPar'94-VAPP VI*, Linz, Austria. September 1994.
- [Bou88] Luc Bougé. Sémantiques du parallélisme: un tour d'horizon. Technical report, LIP, ENS-Lyon, July 1988.
- [Bou93] Luc Bougé. Le modèle de programmation à parallélisme de données: une perspective sémantique. *Technique et Science Informatiques*, 1993.
- [Brä89] Thomas Bräunl. Structured SIMD programming in PARALLAXIS. *Structured Programming*, 10(3):121–132, July 1989.
- [Brä90] Thomas Bräunl. Transparent massively parallel programming with Parallaxis. In *ISMM Int'l Conf. on Parallel and Distributed Computing and Systems*, New-York, NY, 1990. Acta Press.
- [CaP94] Les langages à objets, June 1994. Numéro spécial de *Calculateurs Parallèles*.
- [CD94] Michel Cosnard and Francis Deprez. Quelques architectures de nouvelles machines. *La Lettre du Transputer et des Calculateurs Parallèles*, 21:29–58, March 1994.
- [Chr90] Peter Christy. Software to support massively parallel computing on the MasPar MP-1. In *Proc. of the IEEE Compcon Spring 1990*, pages 29–33, San Francisco, CA, February 1990. IEEE Society Press.
- [CT93] Michel Cosnard and Denis Trystram. *Algorithmes et Architectures Parallèles*. InterÉdition, 1993.

-
- [DD90] IBM Product Design and Development Advanced Workstations Division. *RISC System/6000 Technology*. IBM, 1990.
- [Dek86] Jean-Luc Dekeyser. *Architectures et algorithmes parallèles pour les méthodes Monté-Carlo en physique des particules*. PhD thesis, Université des Sciences et Techniques de Lille, October 1986.
- [DKV88] David C. Douglas, Brewester A. Kahle, and Alex Vasilevsky. The architecture of the CM-2 data processor. Technical Report TMC-91, Thinking Machine Corp., Cambridge, MA, 1988.
- [DLM93a] J.-L. Dekeyser, D. Lazure, and Ph. Marquet. HELP for parallel scientific programming. In *Proc. of the Euromicro Workshop on Parallel and Distributed Processing*, pages 22–29, Gran Canaria, Spain, January 1993. IEEE Computer Society Press.
- [DLM93b] Jean-Luc Dekeyser, Dominique Lazure, and Philippe Marquet. HELP: A model to think, write, and run data-parallel algorithms for parallel scientific programming. In *Workshop on Parallel and Distributed Processing – WP&DP’93*, Sofia, Bulgaria, May 1993.
- [DLM93c] Jean-Luc Dekeyser, Dominique Lazure, and Philippe Marquet. HELP: Un support géométrique pour la programmation data-parallèle. In *Renpar5, 5es Rencontres sur le Parallélisme*, Brest, May 1993.
- [DLM94] Jean-Luc Dekeyser, Dominique Lazure, and Philippe Marquet. A geometrical data-parallel language. *ACM Sigplan Notices*, 29(4):31–40, April 1994.
- [DM94] Jean-Luc Dekeyser and Philippe Marquet. Data-parallel object classification. In *Franco/British N+N meeting on data-parallel languages and compilers for portable parallel computing*, Villeneuve d’Ascq, France, April 1994.
- [DMP90] Jean-Luc Dekeyser, Philippe Marquet, and Philippe Preux. EVA — a vector explicit language, an alternative language to FORTRAN 90. *ACM Sigplan Notices*, 25(8):53–71, August 1990.
- [Don93] Jack Dongarra. Performance of various computers using standard linear equations software. Cs-89-95, Computer Science Department. University of Tennessee, 1993.
- [Fea93] Paul Feautrier. Toward automatic distribution. In *ACM International Conference on Supercomputing (ICS)*, Tokyo, Japan, July 1993.
- [FLLQ89] P. Frison, D. Lavenier, H. Leverage, and P. Quinton. A vlsi programmable systolic architecture. In *International Conference on Systolic Array*, 1989.
-

BIBLIOGRAPHIE

- [Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 31(9):948–960, September 1972.
- [Fon94] Cyril Fonlupt. *Distribution Dynamique de Données sur Machines SIMD*. Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, December 1994.
- [For93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Rice University, Houston, TX, May 1993.
- [Gav92] Cyril Gavaille. Evaluation of the MasPar performances. Rapport technique 92-01, LIP, ENS Lyon, 1992.
- [Gia91] Jean-Louis Giavitto. *8 1/2: un modèle MSIMD pour la simulation massivement parallèle*. PhD thesis, Université Paris XI Orsay, dec 1991.
- [GMS94] Jack J. Gongarra, Hans W. Meuer, and Erich Strohmaier. *TOP500 Supercomputers sites*. University of Mannheim. Disponible sur <ftp.uni-mannheim.de>, June 1994.
- [GS94] J-L. Giavitto and J-P. Sansonnet. Introduction à 8,5 version 1.0. Technical report, Laboratoire de Recherche en Informatique – Université de Paris sud, mai 1994.
- [Hil85] W. Daniel Hillis. *The Connection Machine*. The MIT Press, Cambridge, MA, 1985. Traduction française, Masson, Paris, 1988.
- [HJ88] Roger W. Hockney and C. R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms*. Adam Hilger Ltd, Bristol and Philadelphia, 1988.
- [HJ90] Samuel P. Harbison and Guy L. Stelle Jr. *Langage C manuel de référence*. Masson, 1990.
- [HKT92] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling FORTRAN D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.
- [Hol90] Allen I. Holub. *Compiler Design in C*. Prentice-Hall, brian w. kernighan edition, 1990.
- [HPCC94] High Performance Computing and Communications. *Technology for the National Information Infrastructure*. Supplement to the Presidents Fiscal Year 1995 Budget, 1994.
- [Hpf92] Proposals for High Performance Fortran, January 1992. High Performance Fortran Meeting.

-
- [Hpf93] High Performance Fortran, final draft, version 1.0, January 1993.
- [Hug91] R. Hughey. *Programmable Systolic Arrays*. PhD thesis, Brown University, 1991.
- [Hwa93] Kai Hwang. *Advanced Computer Architecture*. McGraw-Hill, 1993.
- [Hyp93] HyperParallel Technologies, Palaiseau, France. *HyperC Documentation Kit*, 1993.
- [Int91] Intel. *Paragon XP/S Product Overview*. Supercomputer division - Intel corporation, 1991.
- [Ker92] Ronan Keryell. *POMP: d'un Petit Ordinateur Massivement Parallèle à Base de Processeurs RISC — Concepts, Étude et Réalisation*. PhD thesis, Université de Paris XI Orsay, 1992.
- [KF93] Alexander C. Klaiber and James L. Frankel. Comparing data-parallel and message-passing paradigms. In *ICPP' 93*, pages II:11–20, 1993.
- [KH89] Brewster A. Kahle and W. Daniel Hillis. The Connection Machine model CM-1 architecture. *IEEE Transactions on Systems, Mans, and Cybernetics*, 19(4):707–713, July 1989.
- [KLS90] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel Language and Design*, 8:102–118, 1990.
- [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The ANSI-C langage*. Prentice-Hall, 1988.
- [KTC⁺90] Dexter Kozen, Tim Teitelbaum, Wilfred Chen, John Field, William Pugh, and Brad Vander Zanden. Alex—an alexical programming language. In *Visual Languages and Applications*, pages 147–158. Plenum Press, 1990.
- [Laz90] Dominique Lazure. Faisabilité de l'implantation de PVC sur transputers et sous Helios. Rapport de DEA, Université des Sciences et Techniques de Lille, 1990.
- [Laz94a] Dominique Lazure. Compilation du langage data-parallèle C-HELP. Research Report ERA-152, Laboratoire d'informatique fondamentale de Lille, Université de Lille 1, May 1994. Available on <ftp.lifl.fr>.
- [Laz94b] Dominique Lazure. Un modèle géométrique pour l'algorithmique numérique intensive. In *Journées 1994 du SEH*, January 1994.
- [Laz95] Dominique Lazure. *Programmation Géométrique à Parallélisme de Données: Modèle, Langage et Compilation*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, January 1995.
-

BIBLIOGRAPHIE

- [LER92] Ted G. Lewis and Hesham El Rewini. *Introduction to parallel computing*. Prentice-Hall, 1992.
- [Lev93] Jean-Luc Levaire. *Contribution à l'Étude Sémantique des Langages à Parallélisme de Données; Application à Compilation*. PhD thesis, Université de Paris 7, 1993.
- [Lig87] Robert Ligonnière. *Préhistoire et histoire des ordinateurs, des origines du calcul aux premiers calculateurs électroniques*. Robert Laffont, 1987.
- [LM94] Dominique Lazure and Philippe Marquet. Compilation du langage data-parallèle C-HELP. In *RenPar'6 (présentation sur affiche)*, Lyon, France, June 1994.
- [Mar92] Philippe Marquet. *Langages Explicites à Parallélisme de Données*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, February 1992.
- [Mar93a] Philippe Marquet. Langages et expression du parallélisme de données. *Technique et Science Informatiques*, 12(6):685–714, 1993.
- [Mar93b] Philippe Marquet. Le parallélisme de données en fortran, 1993. Cours en DESS d'informatique parallèle - Université des Sciences et Technologies de Lille.
- [Mas90] MasPar Computer Corp., Sunnyvale, CA. *MasPar Assembly Language (MPAS) — Reference Manual.*, March 1990. Confidential.
- [Mas91a] MasPar Computer Corp., Sunnyvale, CA. *MasPar Fortran — Reference Manual, Software Version 1.0*, March 1991. Doc. 9303-0000, Rev. A1.
- [Mas91b] MasPar Computer Corp., Sunnyvale, CA. *MasPar Parallel Application Language (MPL) — Reference Manual, Software Version 2.0*, March 1991. Doc. 9302-0000, Rev. A4.
- [Mas91c] MasPar Computer Corp., Sunnyvale, CA. *MasPar Parallel Application Language (MPL) — User Guide, Software Version 2.0*, March 1991. Doc. 9302-0100, Rev. A4.
- [Mau89a] Christophe Mauras. Définition de ALPHA : un langage pour la programmation systolique. Technical report, IRISA, Campus de Beaulieu. 35042 Rennes Cédex. France, December 1989.
- [Mau89b] Christophe Mauras. *Alpha: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, Université de Rennes I, December 1989.
- [Met87] M. Metcalf. Fortran 8x – the emerging standard. Technical Report DD/87/1, CERN, 1987.

-
- [Mic94] O. Michel. Manuel de référence du langage 8 1/2. Technical report, Laboratoire de Recherche en Informatique, 1994. À paraître.
- [Mon94] Catherine Mongenet. Data compiling for systems of affine recurrence equations. In *Proceedings of the Franco/British N+N meeting on data-parallel languages and compilers for portable [a]rallel computing*. Université de Lille 1, April 1994.
- [Oed93] Wilfried Oed. The Cray Research massively parallel processor system Cray T3D. Technical report, Cray Research GmbH, November 1993.
- [Ols92] Jan Olszewski. A flexible thinning algorithm allowing parallel, sequential, and distributed application. *ACM Transactions on Mathematical Software*, 18(1):35–45, March 1992.
- [Par92] Nicolas Paris. Définition de POMPC (version 1.99). Rapport de Recherches 92-5, LIENS, École Normale Supérieure, Paris, March 1992.
- [Par93a] Nicolas Paris. Compilation des structures de contrôle de flot pour le parallélisme de données. *Technique et Science Informatiques*, 1993.
- [Par93b] Nicolas Paris. Support pour les données non-structurées dans le langage à parallélisme de données HyperC. In *RenPar5, 5e Rencontres du parallélisme*, Brest, France, May 1993.
- [PE94] Serge Petiton and Guy Edjlali. Data parallel structures and algorithms for sparse matrix computation. In G.R. Joubert, D. Trystram, F.J. Peters, and D.J. Evans, editors, *Parall Computing: Trends and Applications*. Elsevier Science, 1994.
- [Per92a] R. H. Perrott. Parallel languages developments in Europe: An overview. *Concurrency: Practice and Experience*, 4(8):589–617, December 1992.
- [Per92b] R. H. Perrott, editor. *Software for Parallel Computers*. Chapman & Hall, London, 1992.
- [Pet62] C. A. Petri. Fundamentals of a theory of asynchronous information flow. In *proceedings of the IFIP information processing congress 62*, pages 386–388, August 1962.
- [PSWF93] Serge Petiton, Youcef Sadd, Kesheng Wu, and William Ferng. Basic sparse matrix computations on the CM-5. *International Journal of Modern Physics*, 4(1):65–83, 1993.
- [PWD92a] Serge Petiton and Christine Weill-Duflos. Algèbre linéaire et architecture massivement parallèles. In M. Cosnard, M. Nivat, and Y. Robert, editors, *Algorithmique Parallèle*. Masson, 1992.
-

BIBLIOGRAPHIE

- [PWD92b] Serge Petiton and Christine Weill-Duflos. Massively parallel preconditioners for the sparse conjugate gradient method. In *Proc. CONPAR 92/VAPP V*, Lyon, France, September 1992. Lecture Notes in Computer Science, vol 634.
- [QR89] P. Quinton and Y. Robert. *Algorithmes et architectures systoliques*. Masson, 1989.
- [QR91] P. Quinton and Y. Robert. *Parallel Algorithms and VLSI Architectures II*. Elsevier, June 1991.
- [Rai94] Frédéric Raimbault. *Étude et réalisation d'un environnement de simulation parallèles d'algorithmes systoliques*. PhD thesis, Université de Rennes 1, 1994.
- [RK80] H.G. Ramming and Z. Kowalik. *Numerical modelling of marine hydrodynamics*. Elsevier oceanography series, 1980.
- [RL93] F. Raimbault and D. Lavenier. ReLaCS for systolic programming. In IEEE Computer Society Press, editor, *International Conference on Applications-Specific Array Processors*, pages 132–135, Venice (I), October 1993.
- [Saa90] Youcef Saad. Sparskit: a basic toolkit for sparse matrix computations. Technical report, University of Minnesota, Minneapolis, 1990.
- [SBM62] Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. The solomon computer. In *proceedings of the Fall 1962 Eastern Joint Computer Conference*, pages 97–107, December 1962.
- [SH94] Donald A. Smith and Timothy J. Hickey. Multi-sld resolution. In *LPAR'94*, 1994.
- [SKz⁺94] Robert J. Souza, P. G. Krishnakumar, Cüneyt M. Özveren, Robert J. Simcoe, Barry A. Spinney, Robert E. Thomas, and Robert J. Walsh. GIGAswitch system: A high-performance packet-switching platform. *Digital Technical Journal*, 6(1):9–22, 1994.
- [SPNR91] Joel Saltz, Serge Petiton, Harry Nerryman, and Adam Rifkin. Performance effects of irregular communication patterns on massively parallel multiprocessors. *Journal of Parallel and Distributed Computing*, 13:202–212, 1991.
- [Ste93] Alan Stewart. Data parallel array assignment. Technical report, Departement of Computer Science, The Queen's University of Belfast, 1993.
- [Thi89] Thinking Machines Corporation, Cambridge, MA. *Paris Reference Manual*, October 1989. Version 5.0, release 5.2.
- [Thi90] Thinking Machines Corporation, Cambridge, MA. *Connection Machine Model CM-2 Technical Summary*. November 1990. Version 6.0.

-
- [Thi91] Thinking Machines Corporation, Cambridge, MA. *Getting Started in C**, February 1991.
- [TMP] Charles J. Turner, David Mosberger, and Larry Peterson. Cluster-C*: Understanding the performance limits. pages 229–238.
- [TW91] Arthur Trew and Greg Wilson, editors. *Past, Present, Parallel: A Survey of Available Parallel Computing System*. Springer-Verlag, 1991.
- [Ung58] S. H. Unger. A computer oriented toward spatial problems. In *Proceedings of the IRE*, pages 1744–1750, October 1958.
- [VB94] V. Van Dongen and C. Bonello. Data parallelism with high performance C. In *Supercomputing symposium '94*, Toronto, June 1994.
- [VBF94] V. Van Dongen, C. Bonello, and C. Freehill. High performance C, language specification, version 0.8.9. Technical Report CRIM-EPPP-94/04-12, CRIM, Montréal, April 1994.
- [Vie94] Éric Vieillot. À chaque programme sa machine. In *6^{èmes} Rencontres francophones du parallélisme*, pages 263–266, Lyon, June 1994.
- [VMQ91] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991. Kluwer Academic Publishers - Boston.
- [VP92] Éric Violard and Guy-René Perrin. Pei: a language and its refinement calculus for parallel programming. *Parallel Computing*, 18:1167–1184, 92.
- [Wav91] WaveTracer, WaveTracer, Inc. *The multiC Programming Language*, 1991. PUB-00001-001-1.00.
- [WD94] Christine Weill-Duflos. *Optimisation de méthodes de résolution itératives de grands systèmes linéaires creux sur machines massivement parallèles*. PhD thesis, Université de Paris VI, fev 1994.
- [WDL94] Christine Weill-Duflos and Dominique Lazure. Irregular structures in data parallel language. In *Franco/British N+N meeting on data-parallel languages and compilers for portable parallel computing*, Villeneuve d'Ascq, France, April 1994.
- [Wil91] Shirley A. Williams. *Programming Models for Parallel Systems*. Wiley – Series in parallel computing, 1991.
- [YT87] Akinori Yonezawa and Mario Tokoro. *Object-Oriented Concurrent Programming*. The MIT Press, 1987.
-

Table des figures

I.1	Le modèle d'exécution SISD	12
I.2	Le problème des dépendances	13
I.3	Le modèle d'exécution pipe-line	15
I.4	Deux exemples de pipe-lines	16
I.5	Le modèle d'exécution SIMD	18
I.6	Le modèle d'exécution MIMD	23
I.7	Quelques machines massivement parallèles	26
I.8	Le parallélisme de données pour la programmation logique	33
II.1	Les langages virtuels ou non-virtuels	43
II.2	Les langages explicites ou implicites	44
II.3	Les deux types de machines virtuelles	46
II.4	Les deux interprétations sémantiques	48
II.5	Projection par blocs ou par cycles	50
II.6	Le modèle de programmation HPF	53
II.7	HPF : un exemple de TEMPLATE	54
II.8	Les différentes distributions d'HPF	55
II.9	HPF : langage basé sur les indices	57

TABLE DES FIGURES

II.10 Un <i>tissu</i> du modèle 8 1/2	62
II.11 Exemples de topologies PARALLAXIS	66
II.12 Irrégularité en HyperC : calcul de convolution hexagonale	67
II.13 Les différentes phases de l'élaboration d'un algorithme	71
II.14 Géométrie et environnement de programmation: HelpDraw	72
II.15 Quelques hyper-espaces	75
II.16 Le référentiel support d'activité	76
II.17 La modélisation géométrique des communications	78
III.1 Les objets du langage C-HELP	82
III.2 Exemples de déclarations d'hyper-espaces comportant 2 ou 3 dimensions primaires	83
III.3 Exemples d'axes secondaires	84
III.4 Exemples d'utilisation des arbres de priorité	87
III.5 Déclarations de DPO	89
III.6 Allocation sur une dimension secondaire	90
III.7 Allocation d'un plan diagonal	91
III.8 Les pointeurs en C-HELP	93
III.9 Les deux types d'affectations	95
III.10 Opérateurs d'incrémentations	96
III.11 Le découpage hiérarchique des expressions	97
III.12 Domaine contraint	99
III.13 Constructeur <code>on</code> sur un bloc	99
III.14 Imbrication de domaines contraints	100

III.15	Règle de conformité	101
III.16	Un segment conforme complètement spécifié	102
III.17	Masquage et conformité	103
III.18	Sémantique du elsewhere	103
III.19	Exemple de découpage hiérarchique	105
III.20	Appel macroscopique et conformité	106
III.21	Translation vers une position absolue	110
III.22	Translation avec déplacement relatif	110
III.23	Changement absolu d'origine	111
III.24	Changement relatif d'origine	111
III.25	Échange absolu d'orientation	112
III.26	Échange relatif d'orientation	112
III.27	Décalage torique	113
III.28	flip	113
III.29	Extraction d'une coordonnée absolue	114
III.30	Extraction d'un intervalle de coordonnées absolues	114
III.31	Extraction d'une coordonnée relative	115
III.32	Extraction d'un intervalle de coordonnées relatives	115
III.33	Répliquions sur une dimension complète	116
III.34	Répliquions n fois vers les coordonnées décroissantes	117
III.35	Répliquion par étirement	117
III.36	Réductions associatives	118
III.37	Erreur de programmation : DPO non valide	120

TABLE DES FIGURES

III.38Exemple de fonction microscopique	123
III.39Le passage de paramètres par valeur ou adresse de la référence	126
III.40Une téléportation	128
III.41Fonction à géométrie générique	129
III.42Appel de fonction à géométrie générique	131
III.43import	132
III.44export	132
III.45C-HELP et l'hétérogénéité	135
III.46La tâche de départ...	136
III.47...la tâche de tri...	137
III.48...et la tâche de fusion	138
III.49Algorithme de fusion	139
IV.1 L'architecture de la MasPar	142
IV.2 Atelier de développement du compilateur	144
IV.3 Les DPO à l'exécution	145
IV.4 L'hyper-espace à l'exécution	146
IV.5 Allocation par les points	149
IV.6 La factorisation du calcul d'adresse	150
IV.7 L'allocation avec recouvrement	154
IV.8 Deux alternatives pour l'allocation des points	156
IV.9 La boucle irrégulière de virtualisation	159
IV.10Efficacité du modèle basé sur le référentiel	165

IV.11 Performances de la boucle régulière de MP-FORTRAN	167
IV.12 Performances de la boucle irrégulière de C-HELP	168
IV.13 Une position critique, gain maximal de l'exemple	169
V.1 Le format de compression COO	173
V.2 Le format de compression CSR	174
V.3 Le format de compression ELL	176
V.4 Le format de compression SGP	177
V.5 Le format de compression S^3	178
V.6 Les interférences du calcul creux	180
V.7 Le creux par la compilation	181
V.8 La création du patron de l'hyper-espace	184
V.9 Les objets denses et creux	184
V.10 La compression des points	185
V.11 Domaine source creux	186
V.12 Multiplication matrice/vecteur	190
VI.1 Calcul du vecteur de rangs : l'algorithme et le programme C-HELP	194
VI.2 Décalages toriques irréguliers	195
VI.3 La multiplication de matrices en C-HELP	196
VI.4 Gauss-Jordan: l'algorithme géométrique	198
VI.5 L'algorithme de Gauss-Jordan en C-HELP	199
VI.6 Les trois phases de la décomposition LU	200
VI.7 La décomposition LU en C-HELP	201

TABLE DES FIGURES

VI.8 Le gradient conjugué préconditionné en C-HELP	203
VI.9 La squelettisation d'une image en C-HELP	204
VI.10Le calcul de U pour une maille de l'espace.	208
VI.11Les domaines de conformité par demi pas de temps	209
VI.12Du Fortran...	209
VI.13...à C-HELP	210
VI.14Les DPO alloués sur une diagonale	215
VI.15Les DPO parallélogrammoïdaux	216
VI.16Les DPO anti-diagonaux	217
VI.17Les DPO hexagonaux et octogonaux	218

