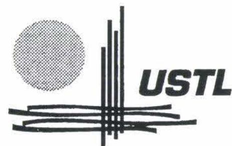
the 2000 0035

50376
1998
311



Numéro d'ordre : 2322

THÈSE

Nouveau Régime

Présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

Pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Christian LEFEBVRE

HPF-BUILDER
-
UN ENVIRONNEMENT VISUEL DE PLACEMENT ET DISTRIBUTION DÉDIÉ À HPF

Thèse soutenue le 30 octobre 1998, devant la Commission d'Examen

Président	:	Vincent CORDONNIER	LIFL, USTL
Rapporteurs	:	François BODIN	IRISA, RENNES
		Thomas BRANDES	GMD/SCAI, SANKT AUGUSTIN, ALLEMAGNE
Examinateurs	:	Laurent COLOMBET	CEA, GRENOBLE
		Jean-Luc DEKEYSER	LIFL, USTL

Remerciements

Un soir d'octobre 1991, Patrick LEBÈGUE m'a expliqué en quoi consistait son travail de Maître de conférences à l'IUT Informatique de Lille. C'est après cette discussion innocente que j'ai décidé de poursuivre dans la voie universitaire et de continuer jusqu'ici. Il ne se doutait sûrement pas avoir fait naître une vocation, mais je tiens à l'en remercier.

C'est à la suite d'une discussion presqu'aussi innocente avec Jean-Luc DEKEYSER que l'idée de développer une interface graphique autour d'HPF est née. Je le remercie de m'avoir encadré pendant mon mémoire de DEA et pendant les trois années de thèse qui ont suivi.

Je tiens également à remercier :
Vincent CORDONNIER pour avoir présidé le jury de la soutenance,
Thomas BRANDES et François BODIN pour avoir accepté de rapporter mon mémoire de thèse,
Laurent COLOMBET pour sa présence comme examinateur.

Un grand merci encore à Thomas, pour m'avoir accueilli en stage intensif d'utilisation de la boîte à outils COCKTAIL et de son *parser*.

Je ne suis pas le seul développeur d'HPF-BUILDER. Je tiens à remercier les étudiants de DEA et d'IUP GMI qui ont participé à l'élaboration de ce projet : Gilles puis Christopher puis Denis, ainsi que les binômes François et Frédéric puis Nicolas et Sébastien.

Je ne suis pas non plus le seul à avoir participé à la rédaction de ce document. Je tiens à remercier Dominique BIRMAN de *Météo-France* et Charlene GLATKOWSKI, d'*AVS Inc.* pour m'avoir fourni des informations et aussi Nathalie REVOL pour sa relecture impressionnante d'efficacité.

Merci aussi à Yeon JOON CHUNG, de l'université de Sud Californie, mon plus fidèle client, bien que je ne le connaisse que par E-mail interposé ! Grâce à lui et ses nombreuses questions, j'ai pu connaître les premières impressions d'un de ces êtres étranges qu'on appelle les utilisateurs ...

Je tiens enfin à remercier toutes les personnes avec qui j'ai partagé le bureau 326 tout au long de ces trois années et aussi les habitués de la cafetière du 326 ainsi que les scotchés du mois d'août, grâce à qui le laboratoire m'a semblé un peu moins vide en cette période de rédaction intensive .

Je m'excuse de vous avoir fait subir mon rire tonitruant à longueur de journées ...

À celui/celle
qui n'a pas encore de prénom
à l'heure où j'écris ces lignes,
et à sa mère ...

Table des matières

Introduction **9**

1 Calcul scientifique et parallélisme **13**

1.1 Introduction 13
1.2 Brève histoire de l'informatique 14
1.3 Le parallélisme 17
1.3.1 Introduction 17
1.3.2 Les machines parallèles 18
1.3.3 Les modèles d'exécution 22
1.3.4 Les modèles de programmation 24
1.3.5 Conclusion 25
1.4 Le calcul scientifique 26
1.4.1 Origines 26
1.4.2 Présentation 27
1.4.3 Les *grand challenges* 28
1.4.4 Conclusion 29
1.5 Les langages 30
1.5.1 Historique 30
1.5.2 FORTRAN 30
1.5.3 HPF 34
1.6 HPF 34
1.6.1 Historique 34
1.6.2 Les directives 35
1.6.3 Les constructions parallèles 36
1.6.4 Les placements 37
1.6.5 Les compilateurs HPF 42
1.6.6 Conclusion 42

1.7 Conclusion 43

2 Les environnements d'aide à la programmation 45

2.1 Introduction 45

2.2 La programmation séquentielle 47

2.2.1 La programmation visuelle 48

2.2.2 La réutilisation 49

2.2.3 L'édition du source 49

2.2.4 La définition des interfaces graphiques 50

2.2.5 L'intégration des outils annexes 52

2.2.6 Plus loin dans l'intégration 53

2.3 La programmation à parallélisme de tâches 54

2.3.1 Le découpage des tâches 56

2.3.2 Le déverminage 57

2.4 L'utilisation de FORTRAN et HPF 59

2.4.1 Le passage de FORTRAN 77 à FORTRAN 95 60

2.4.2 La parallélisation automatique 61

2.4.3 Le placement de données 61

2.4.4 La visualisation de données 63

2.4.5 Évaluation des performances 65

2.4.6 Intégration d'outils 66

2.5 Conclusion 68

3 HPF-Builder 71

3.1 Introduction 71

3.2 Concept, fonctionnalités 73

3.2.1 Niveau global 76

3.2.2 Niveau hiérarchique 77

3.2.3 Niveau directive 77

3.2.4 Niveau localité 78

3.3 Mise en œuvre 79

3.3.1 Organisation générale 79

3.3.2 L'interface graphique 80

3.3.3 Le serveur d'informations 92

3.3.4 Le module d'analyse de source 94

3.3.5 Le module `Analyser` et la structure `context` 95
3.3.6 Le module `Proj` 95
3.4 Modélisation des placements 96
3.4.1 Introduction 96
3.4.2 Modélisation 97
3.4.3 Calcul des pré-images 101
3.4.4 Conclusion 105
3.5 HPF-BUILDER : Conclusion 106

4 Pré-évaluation de performances 107
4.1 Introduction 107
4.2 Idée générale 108
4.2.1 Quelle information sait-on obtenir ? 109
4.2.2 Quelle information doit-on visualiser ? 111
4.2.3 Que faut-il afficher ? 111
4.2.4 Comment afficher ? 112
4.2.5 Que doit-on calculer ? 114
4.2.6 Comment calculer ? 116
4.3 Mise en œuvre 119
4.3.1 Calcul direct 120
4.3.2 Calcul par simulation 120
4.3.3 Utilisation du calcul polyhédrique 123
4.4 Conclusion, perspectives 123

5 Démonstration 125
5.1 Présentation 125
5.2 Démarrage de la démonstration 127
5.3 À suivre 128

Conclusion 129

Annexe

CD-ROM de démonstration

Introduction

> Il est indigne d'hommes remarquables de perdre des heures à un travail d'esclave, le calcul, qui pourrait fort bien être confié à n'importe qui, avec l'aide de machines
>
> *Leibniz*

Les scientifiques de tout domaine (météorologie, aéronautique, physique des particules ...) font un usage de plus en plus massif des ordinateurs. Leurs besoins en capacités de calculs vont croissant et les informaticiens n'ont jamais de répit dans la course à la puissance.

C'est pourquoi le parallélisme est vite devenu indispensable pour répondre à la demande en quantité de calculs qu'une machine peut effectuer en un temps raisonnable.

Cette idée, bien que simple dans sa conception, pose de gros problèmes quant à sa mise en œuvre pratique. La puissance théorique d'une machine parallèle (puissance de chaque nœud de calcul multipliée par leur nombre) est rarement possible à atteindre en pratique.

En effet, on se heurte à un problème de définition des modèles et des langages de programmation de telles machines : comment définir un langage de programmation qui soit suffisamment proche de la machine pour profiter de ses caractéristiques, tout en faisant abstraction de son architecture afin de garder un langage portable et simple à utiliser ?

En réponse à cette question et en visant les applications de calcul scientifique, le modèle de programmation à parallélisme de données a été proposé. Il offre un moyen d'exprimer des algorithmes dans lesquels les ensembles réguliers de données sont traités dans leur globalité. La compilation peut alors gérer ces ensembles de façon à paralléliser les traitements selon une méthode adaptée à chaque architecture cible.

Parallèlement à ces recherches sur de nouvelles méthodes de programmation et de compilation, une autre demande vient à contre-courant : les scientifiques ne veulent pas perdre de temps à apprendre des langages complètement nouveaux à chaque innovation sortie de la communauté informatique. Ils ont pris le temps de se mettre à FORTRAN et aimeraient continuer à l'utiliser.

Comme compromis entre ces deux aspects, un projet d'extension de FORTRAN a été initié

par un forum réunissant des constructeurs de machines, des auteurs de compilateurs, des chercheurs en informatique et des utilisateurs potentiels, c'est-à-dire des scientifiques. Ce projet a abouti au langage HPF.

Ce langage est une extension de la dernière norme en date de FORTRAN. Dans un programme HPF, on trouve un code tout à fait identique à un programme FORTRAN, mais un certain nombre de directives y sont ajoutées pour guider le compilateur dans les méthodes de parallélisation qu'il peut employer. Ainsi, même les programmes existants peuvent être transformés petit à petit en programmes HPF, en y insérant progressivement des directives d'optimisation.

Bien que cette approche n'oblige pas à apprendre un langage complètement nouveau, les directives HPF constituent une notion très spécifique au parallélisme. L'insertion de directives efficaces n'est donc pas évidente pour un non spécialiste.

D'un autre côté, dans toutes les phases de la programmation, aussi bien séquentielle que parallèle, on trouve des outils d'aide au développement.

Il en découle immédiatement l'idée de construire un tel outil d'aide spécialisé dans l'insertion des directives HPF. Cet outil permettrait de guider le programmeur dans ses choix, en lui proposant une visualisation intuitive des optimisations qu'il définit et de leurs effets.

C'est à partir de cette idée qu'est né le projet HPF-BUILDER. Ce document a pour but d'en d'exposer le déroulement et plus particulièrement de décrire la mise en œuvre du prototype qui en a résulté.

Cette description sera effectuée progressivement. Dans un premier chapitre, nous retracerons l'évolution du calcul scientifique. Pour cela, nous parlerons des machines inventées et nous en ferons ressortir les architectures et les modèles de programmation qui en ont découlé. De cet historique, nous pourrons ainsi isoler les problèmes de programmation auxquels sont confrontés les scientifiques, afin d'introduire les solutions qu'on peut y apporter. Nous porterons ensuite notre attention sur les langages FORTRAN et HPF.

Dans le second chapitre, nous exposerons quelques outils existants dans le domaine de l'aide à la programmation. La construction de ce chapitre consistera à accompagner le programmeur dans les étapes du développement d'une application parallèle et à dégager de chacune d'elles les problèmes auxquels il est confronté.

Pour chaque étape, nous citerons un exemple d'outil qui aide le programmeur à résoudre ces problèmes. Ainsi, nous pourrons faire ressortir de cet état de l'art les caractéristiques intéressantes à reprendre dans le cadre de notre projet.

La conclusion de ce chapitre sera un cahier des charges d'un prototype réunissant les idées vues dans l'état de l'art et dont un des composants sera *une interface graphique d'aide à l'insertion, la visualisation, l'évaluation et la modification de directives HPF.*

Le troisième chapitre reprendra alors en détail le cahier des charges de ce composant pour en déduire la mise en œuvre d'un prototype. Nous détaillerons les concepts que nous avons définis pour déterminer les fonctionnalités précises du prototype, c'est-à-dire les méthodes de visualisation et d'édition d'un programme dans son ensemble et des placements de données en particulier.

Ensuite, l'organisation du logiciel qui a été écrit sera décrite, ainsi que son interface graphique et le module qui lui fournit les informations qui lui sont nécessaires. Ce chapitre se

terminera par l'explication du modèle mathématique de modélisation des directives HPF qui permet la manipulation de ces informations.

Le quatrième chapitre présentera les travaux préliminaires à l'ajout d'un module de pré-évaluation des performances du programme obtenu à partir d'HPF-BUILDER. Ce module a pour but de fournir des informations supplémentaires au programmeur, afin de mieux le guider dans ses choix. Nous exposerons les principes de cette notion de pré-évaluation avant d'en détailler quelques mises en œuvre.

Le cinquième chapitre a pour but de montrer plus précisément le fonctionnement du prototype d'HPF-BUILDER. Pour cela, nous avons décidé de mettre en place une démonstration interactive du logiciel, sur un CD-ROM. Ce chapitre n'est donc qu'une rapide introduction au contenu du CD-ROM.

Enfin, nous résumerons l'ensemble des travaux qui ont amené à ce mémoire et nous présenterons quelques perspectives d'évolution de ce projet. Nous reviendrons notamment sur le projet global dans lequel le prototype actuel pourrait venir s'insérer.

Une dernière remarque avant d'entrer dans le vif du sujet : il est actuellement plus rapide et plus facile de trouver des documents, ou au moins des références de documents « papier », sur Internet qu'en faisant le tour des bibliothèques et des bases de données bibliographiques.

C'est pourquoi une grande partie des documents auxquels nous ferons référence dans la suite du mémoire seront des adresses Internet.

Internet est un support très mouvant; un site Internet peut disparaître du jour au lendemain, alors qu'il y a toujours moyen de retrouver un document écrit. Cependant, nous avons essayé de nous référer à des sites sûrs et reconnus, existant depuis déjà un certain temps et dont on pourrait facilement retrouver la trace via un moteur de recherche.

Chapitre 1

Calcul scientifique et parallélisme

Quand un président de la république s'intéresse aux ordinateurs :

Nous avons la certitude d'avoir une arme totalement dissuasive et sûre et des capacités d'évolution puisque nous avons acquis aussi les techniques dites de la simulation, ce qui nous permettra de faire dorénavant des expériences en ordinateur.

Jacques Chirac, le 22 Février 1996

1.1 Introduction

Les techniques de simulation citées en exergue consistent principalement à manipuler des énormes quantités d'équations et à les résoudre de façon approchée au moins. Pour cela, les scientifiques font appel à l'informatique.

Ainsi, de la simulation de *crash-tests* automobiles à la prévision météorologique, l'utilisation de l'ordinateur par les scientifiques, pour effectuer les calculs intensifs, est devenu incontournable.

Actuellement, plusieurs pays ont une forte volonté politique de développement des méthodes de simulation autour des armes nucléaires. Or il n'existe pas encore de machines assez puissantes pour réaliser les simulations en 3D dans un temps raisonnable.

Il est donc clair que l'avenir de la recherche en architectures hautes performances et en méthodes de programmation efficaces de ces machines nouvelles est assuré.

Dans ce chapitre, nous allons chercher à montrer que les scientifiques n'ont pas attendu

les progrès de l'outil informatique pour profiter de la puissance des ordinateurs.

Ils ont activement participé à son développement. Nous verrons que les tous premiers ordinateurs ont été construits spécifiquement pour le calcul intensif et que ce n'est qu'après que leur utilité dans d'autres domaines est apparue.

Nous allons donc retracer brièvement l'histoire de l'informatique et en particulier celle de l'informatique scientifique et du parallélisme. Nous présenterons ensuite les langages de programmation en détaillant ceux dédiés au calcul scientifique : FORTRAN et son dernier successeur, HPF.

1.2 Brève histoire de l'informatique

Depuis l'apparition des nombres et des premières notions de mathématiques, l'homme a cherché à se faciliter l'effort nécessaire pour effectuer des calculs. Ainsi, depuis les premières abaques et bouliers, des mécanismes de plus en plus complexes ont été inventés pour calculer.

Toutes ces inventions ont mené à l'apparition d'une catégorie de machines à part entière : les ordinateurs.

Bien que leurs frontières soient souvent difficiles à discerner, l'historique de l'informatique est généralement divisé en générations, que nous allons brièvement énumérer [1].

Les prémisses

Aux tous débuts, les prémisses de ce qui deviendra des ordinateurs sont les machines à calculer.

En effet, dès 1623, W. SCHICKARD invente une *horloge calculante* qui sait effectuer mécaniquement des additions, soustractions et multiplications et en 1642, B. PASCAL construit la plus célèbre *pascaline* qui sait additionner et soustraire[Wil97, Ch. 3].

Bien plus tard, la deuxième voie explorée par l'informatique est ensuite le stockage et le traitement en masse d'informations. Ainsi, en 1884, H. HOLLERITH crée une tabulatrice à cartes perforées afin de réaliser le recensement américain de 1890. Devant le succès de cette expérience, il crée la firme *Tabulating Machines Corporation* [2]. En 1924, cette société est rebaptisée *International Business Machine*, plus connue sous le sigle *IBM* ...

La première machine « programmable » qui ne soit pas dédiée à un problème particulier est la *Machine à Différences* de C. BABBAGE, qu'il commence à concevoir en 1823 mais qui ne fut jamais mise au point, tout comme sa *Machine Analytique*, en 1842. Malgré son insuccès dans la construction effective de ces machines, BABBAGE et ses associés (dont notamment la célèbre Ada LOVELACE) ont apporté de nombreuses nouveautés dans l'informatique telles que les branchements conditionnels, les boucles itératives et les variables d'index.

C'est à partir de ces travaux que G. et E. SCHEUTZ construisent une machine qui remporte une médaille d'or à l'Exposition de Paris de 1855.

1. Cet historique a été principalement constitué à partir de [Wil97] et de documents d'internet [Ros][OV, Ch. 3.1].

2. à ne pas confondre avec l'autre TMC : *Thinking Machines Corporation* !

Les premiers ordinateurs

C'est ensuite autour de la seconde guerre mondiale que les avancées scientifiques et technologiques commencent à se précipiter.

En 1937, A. TURING introduit les notions de calculabilité et de complexité et publie son modèle de *Machine de* TURING. L'année suivante, la thèse de SHANNON établit le lien entre algèbre Booléenne et circuits binaires.

Dans la même période, plusieurs équipes, comme K. ZUSE et H. SCHREYER (*Z1* en 1938, puis *Z2* ...), J. ATANASOFF et C. BERRY (additionneur binaire en 1939), G. STIBITZ et S. WILLIAMS (*Model I* en 1940)[Wil97, Ch. 6&7] construisent des calculateurs électromécaniques.

Pendant la guerre, l'armée allemande fait passer ses messages par *Enigma*, une machine de cryptage. Pour casser ce code, les Anglais rassemblent une équipe impressionnante de spécialistes sur le site de Bletchley Park, où ils mettent au point successivement les machines *Robinson* et *Colossus*. Ces machines intègrent de nombreux concepts tels que l'arithmétique binaire, l'horloge et les sous-programmes (ces travaux resteront secret défense jusqu'en 1975 ...).

Enfin, vers la fin de la guerre, l'armée américaine finance l'équipe de P. ECKERT et J. MAUCHLY pour construire l'*Eniac*, afin d'effectuer des calculs de balistique. Cette machine sera ensuite utilisée pour la mise au point de la bombe H.

L'électronique

Dans les années 50, les progrès de l'électronique (transistor (1947), mémoire à tores de ferrite (1953), disque dur (1956)) commencent à démocratiser l'ordinateur.

P. ECKERT et J. MAUCHLY lancent en 1951 le premier ordinateur commercial : l'*Univac 1*. Le premier fut vendu 750 000 $ au bureau de recensement américain. En 1952, trois quarts d'heures après la clôture des votes et à partir de 7% des dépouillements, un *Univac* prédit la victoire de EISENHOWER avec 438 voies; le résultat final fut 442.

L'ordinateur devient interactif et les langages de programmation « évolués » comme FORTRAN (1956), ALGOL (1958) et COBOL (1959) apparaissent.

De nouvelles notions comme les unités de calcul flottant et le recouvrement calcul/accès mémoires laissent entrevoir les débuts du parallélisme.

Les circuits intégrés

La génération suivante fait un énorme pas en avant avec l'arrivée des circuits intégrés (1958) et des mémoires à semi-conducteurs, qui vont permettre de mettre en place de nouvelles techniques comme la micro-programmation, le pipeline et le temps partagé.

Après quelques balbutiements, le parallélisme apparaît clairement en 1964 avec le *CDC 6600* conçu par Seymour CRAY. Il utilise 10 unités fonctionnelles pouvant accéder simultanément à 32 bancs mémoire et atteint le MégaFlops[3].

3. Million d'opérations en virgule flottante par seconde. À titre de comparaison, l'*Eniac* pouvait effectuer

L'utilisation de l'ordinateur sort des domaines du calcul et du traitement d'information en 1960, avec SPACEWAR sur *PDP-1* qui fait émerger un nouveau domaine d'application : les jeux !

La deuxième moitié des années soixante voit aussi apparaître les prémisses des ordinateurs personnels et des interfaces graphiques.

La micro-informatique

Les années 70 voient l'informatique devenir accessible aux particuliers, notamment grâce à l'apparition des microprocesseurs qui réduisent considérablement les coûts des ordinateurs.

Les premiers kits de micro-ordinateurs arrivent sur le marché et les *Apple*, *Atari* et autres *Commodore* ont un énorme succès : chacun peut découvrir les joies du BASIC et des PACMAN stockés sur cassettes.

En 1975, avec ELECTRIC PENCIL de M. SHRAYER, la dernière grande utilisation de l'ordinateur apparaît : la bureautique.

Dans le même temps, C et UNIX apparaissent et les super-ordinateurs vectoriels dépassent les 100 MFlops. Très vite, UNIX devient le système d'exploitation incontournable pour tous les ordinateurs dédiés au calcul scientifique.

Les années 80 voient les machines massivement parallèles entrer en concurrence avec les machines vectorielles. Enfin, dans les années 90, le boum des réseaux amène à l'apparition des réseaux de stations de travail. Ces derniers points seront détaillés dans la section 1.3.

Aujourd'hui

Actuellement, en nombre d'ordinateurs et d'utilisateurs, l'informatique de gestion et la bureautique représentent une majorité écrasante. On considère qu'il y a actuellement dans le monde environ 150 millions de micro ordinateurs, contre 500 super-calculateurs [Qui98].

Cependant, depuis le *CDC 6600* jusqu'aux projets TeraFlops, les machines les plus puissantes sont toujours utilisées pour le calcul intensif. C'est parce que les scientifiques demandent des puissances de calcul toujours croissantes que les ordinateurs ont évolué dans ce sens. Ces progrès ne sont repris qu'ensuite dans les ordinateurs grand public.

Ainsi, et surtout grâce aux énormes progrès des cinquante dernières années qui ont mené à une grande démocratisation de l'informatique, il est maintenant possible d'automatiser des calculs lourds, pour des applications de plus en plus complexes et de plus en plus variées.

Pour cela, les progrès de la technologie électronique ne sont pas les seuls responsables. L'architecture des ordinateurs et les modèles de programmation ont énormément évolué.

Les principales améliorations de performances sont dues à la parallélisation des machines. C'est ce point que nous allons détailler dans la section suivante, en présentant les divers modèles existants et en retraçant leur historique.

---

environ 330 multiplications entières par seconde.

1.3 Le parallélisme

1.3.1 Introduction

En 1947, J. VON NEUMANN décrit le principe de base d'une architecture d'ordinateur. Dans son modèle, la même mémoire contient les données à traiter et les instructions décrivant les traitements à effectuer. Ces instructions sont interprétées une à une par une unité de calcul en suivant un cycle de 5 étapes : lecture de l'instruction en mémoire, décodage de cette instruction, lecture des opérandes, exécution de l'instruction et enfin rangement du résultat en mémoire.

Depuis, la majorité des ordinateurs construits suivent ce modèle. Sa simplicité et son adéquation au raisonnement algorithmique commun lui ont assuré ce succès.

Bien que les progrès de la technologie aient permis de rendre les ordinateurs de plus en plus puissants, ce modèle ne permet pas de satisfaire toute la demande.

En 1965, G. MOORE a évalué que la montée en puissance des ordinateurs suit une loi linéaire: le taux d'intégration des transistors ainsi que la puissance de calcul sont environ multipliés par deux chaque année. Jusqu'à présent, cette loi est toujours vérifiée [Vui98], mais pour cela, nous sommes actuellement arrivés à des composants dont l'épaisseur des pistes est d'environ 20 atomes. Si on ajoute à cela les problèmes d'effets quantiques et autres parasitages thermiques, il est actuellement considéré qu'on ne pourra plus augmenter la puissance que d'un facteur inférieur à 100.

Malgré cette évolution technologique impressionnante, certains traitements très lourds demandent plus de puissance qu'on ne peut en fournir actuellement. Des projets à long terme planifient pour d'ici 2004 des besoins de puissance 100 fois supérieurs à ce qu'on sait faire de mieux aujourd'hui (voir la section 1.4.3 page 28 sur le projet ASCI).

Il faut donc faire évoluer non seulement la technologie, mais aussi la façon de l'exploiter, c'est-à-dire l'architecture des ordinateurs. L'idée de faire travailler simultanément plusieurs processeurs dans le même but est donc apparue.

Bien sûr, la mise en application de cette idée est loin d'être évidente et multiplier par dix le nombre d'unités de traitement est loin de diminuer d'autant le temps de calcul. En effet, certaines opérations d'un algorithme nécessitent l'utilisation des résultats d'une autre étape. Multiplier indéfiniment le nombre d'unités de traitement mène donc forcément à les sous-utiliser.

Malgré ce rendement généralement assez faible, il s'agit du seul moyen d'augmenter la puissance des ordinateurs de façon très importante et plus vite que les progrès de l'électronique ne peuvent le faire.

Nous allons maintenant retracer l'histoire de cette évolution des architectures, avant de détailler les modélisations qui en ont été faites. Nous verrons ainsi que ces modélisations suivent de très près les évolutions des machines.

1.3.2 Les machines parallèles

Les machines pipeline

Les premières notions de parallélisme apparaissent simultanément sur deux machines, en 1959.

La première, l'*Univac-Larc*, possède une unité d'entrées/sorties qui peut travailler parallèlement à ses deux unités de calcul. Elle introduit donc les notions de recouvrement calcul/communications.

La seconde, issue du projet *Stretch* d'*IBM*, introduit les premières notions de *lookahead*, c'est-à-dire le principe de lire en avance une instruction pendant que la précédente se termine.

En effet, on a vu que le fonctionnement du processeur est basé sur un cycle à cinq étapes. Or, quand le processeur lit les opérandes d'une instruction, rien n'empêche de commencer à décoder l'instruction suivante en même temps.

C'est ce qu'on appelle le fonctionnement en pipeline. Cette technique a l'énorme avantage de ne rien changer au fonctionnement global du processeur. On peut donc aller jusqu'à cinq fois plus vite sans rien changer au modèle de programmation.

Bien sûr, tout ne va pas toujours bien. Ainsi, quand une instruction doit lire un opérande qui est écrit à l'instruction immédiatement précédente, il est nécessaire d'attendre la fin de cette instruction avant de continuer. De même, quand on arrive sur une instruction de branchement conditionnel, il faut bien attendre pour savoir où est l'instruction suivante.

En 1964, le *CDC 6600* résout ce problème en introduisant une table de disponibilité des unités de calcul, qui permet de gérer les conflits d'accès aux différentes unités du processeur. De plus, certaines de ces unités sont présentes en plusieurs exemplaires afin de réduire les conflits. Cette table préfigure donc le système du *vecteur de collision*.

Actuellement, cette technique est bien maîtrisée et se retrouve dans la quasi-totalité des micro-processeurs.

Au niveau supérieur, cette méthode peut également être utilisée lors du traitement d'ensembles réguliers de données, tels que les vecteurs. Cette solution est appliquée dans les ordinateurs dits vectoriels qui proposent des instructions de traitement sur des vecteurs.

Prenons l'exemple de l'addition terme à terme de deux vecteurs de nombres flottants : l'addition se décompose en 4 étapes qui sont la comparaison des exposants, le décalage des mantisses, leur addition puis la normalisation. Dans un ordinateur vectoriel, on trouve donc une unité arithmétique vectorielle qui comporte 4 sous-unités utilisées en pipeline pour effectuer ce type d'opérations.

Les premiers ordinateurs construits spécifiquement sur ce modèle apparaissent au milieu des années 70 et tournent autour de 40 millions d'opérations par seconde. Il s'agit du *CDC Star-100* (1973) et de l'*ASC* de *Texas Instruments* (1972).

Une deuxième génération d'ordinateurs de ce type apparaît avec le *Cray-1* en 1976 (successeur de la série des *CDC 6600* et *7600*), le *Cyber-205* en 1981 (successeur du *Star-100*) et le *VP-200* de *Fujitsu* en 1982.

Ces machines évoluent ensuite vers des solutions à deux degrés de parallélisme, en utilisant plusieurs processeurs vectoriels en parallèle. C'est le cas du *Cray X-MP* (1984), qui contient

4 processeurs vectoriels et de ses successeurs.

Les machines SIMD

Parallèlement à ces développements, une autre approche du parallélisme fut proposée dès 1958 par UNGER, qui proposa une structure d'ordinateur composée de processeurs élémentaires disposés en grille et contrôlés par un maître commun.

Dans ce type d'architecture, on dispose d'un ensemble de processeurs élémentaires possédant chacun leur mémoire, mais qui effectuent tous la même instruction à un instant donnée. On appelle cette structure l'architecture parallèle synchrone, ou SIMD [4].

Chaque processeur a une architecture très simplifiée, puisqu'il suffit d'une seule unité de gestion des instructions qui distribue directement le micro-code à chaque processeur.

Cette simplification permet une meilleure intégration, et autorise donc la présence d'un grand nombre de ces processeurs élémentaires.

Cette idée est à la base de la conception de la série des *Illiac*, vers la fin des années 60, qui aboutit à l'*Illiac-IV*.

L'architecture de l'*Illiac-IV* et de son successeur, la *BSP*, ont jeté les bases de nombreuses machines plus ou moins spécialisées (*CLIP-4*, *DAP*, *MPP* ...), ainsi que de machines plus générales comme la *CM-2* de *Thinking Machines* (1986), qui regroupe 65536 processeurs 1 bit et la *MasPar* de *DEC* qui regroupe jusqu'à 16384 processeurs 4 bits.

Cette architecture donne de très bons résultats pour des machines dédiées à une tâche précise. Mais leur évolution a été arrêtée par des problèmes de synchronisation, car au delà d'une certaine fréquence, la distribution du micro-code pose de gros problèmes techniques.

De plus, le rendement de telles machines est assez faible. En effet, dans une architecture SIMD, le parallélisme fournit une accélération théorique égale au nombre de processeurs. Cependant, la gestion des parties séquentielles des codes comme les contrôles d'indices de boucles doit être effectuée par un seul processeur. Pendant ce temps, le reste de la machine est inoccupé.

De même, pour pouvoir effectuer des opérations conditionnelles cette architecture synchrone nécessite un principe d'inhibition de certains processeurs qui réduit là encore le degré de parallélisme.

Là encore, l'évolution s'est ensuite dirigée vers des solutions hybrides mettant en œuvre plusieurs groupes SIMD en parallèle. On parle alors d'architecture MSIMD.

C'était déjà le cas du projet *Illiac IV* qui devait comporter 4 quadrants indépendants, mais les limites de temps et de budget avaient alors réduit le projet à un seul de ces quadrants.

Les machines MIMD

Dans les années 70, un autre principe émerge. Il consiste à utiliser plusieurs processeurs complètement désynchronisés et reliés par un réseau interne à haut débit. Ainsi, le *C.mmp*,

4. *Single Instruction Multiple Data*, une même instruction sur plusieurs données.

de l'université de Carnegie Millon est composé de 16 *PDP-11* qui se partagent une mémoire commune via un cross-bar 16×16.

La *CM-5* constitue un compromis entre les deux architectures. Elle peut fonctionner en mode synchrone (donc SIMD) ou asynchrone (donc MIMD). Virtuellement (car la configuration maximale n'a jamais été construite) cette machine peut atteindre le TeraFlops.

L'architecture MIMD se divise en deux branches, selon l'organisation de la mémoire : les machines à mémoire partagée possèdent un espace d'adressage unique et le système doit gérer lui-même les problèmes d'accès; dans les machines à mémoire distribuée, chaque processeur possède une mémoire privée et les échanges d'informations doivent donc se faire explicitement par des passages de messages.

Dans les deux cas, une autre caractéristique qui définit chaque machine est la topologie du réseau qui relie les processeurs.

Ainsi, le *Cray T3D* propose jusqu'à 2048 processeurs placés en grille 3D avec lesquels on peut utiliser du passage de message, ou laisser le système simuler une mémoire partagée.

D'un autre côté, les *IBM SP-2* proposent jusqu'à 512 processeurs reliés par une hiérarchie de cross-bars permettant de connecter n'importe quelle paire de processeurs. On a donc une topologie virtuelle fortement connexe.

Actuellement, la plupart des machines parallèles fonctionnent sur le principe MIMD. De plus, grâce à l'évolution des performances des machines séquentielles et des réseaux rapides, ainsi que la baisse de leurs coûts, de plus en plus de machines parallèles « bon marché » sont apparues.

En effet, définir et construire un processeur performant, à partir de zéro, dans le seul but de l'intégrer à une machine parallèle est une tâche très lourde. Beaucoup de constructeurs se retrouvent avec des machines nouvelles dont les processeurs sont déjà dépassés par ceux des machines séquentielles.

C'est pourquoi il est devenu fréquent d'utiliser un processeur existant, voire toute une machine, comme brique de base d'une machine parallèle. Il suffit alors de relier ces machines par un réseau performant pour obtenir une machine parallèle de puissance comparable aux machines « sur mesure ».

Les grands progrès des dernières années en technologie des réseaux rapides permettent de créer des réseaux ou groupes de stations de travail[5] très performants pour un coût beaucoup moins élevé. De plus, ces machines sont bien plus évolutives, puisqu'il devient possible de changer indépendamment les processeurs, les mémoires ou le réseau.

Conclusion

La figure 1.1 page suivante présente les principales machines qui ont marqué l'évolution du parallélisme. Les puissances indiquées sont relatives aux performances théoriques, dans les plus grosses configurations construites.

Comme on peut le remarquer sur la figure 1.2 page 22 qui résume l'évolution des machines selon leur architecture, la montée en puissance suit une loi exponentielle relativement parallèle

5. aussi nommés NOW, pour *Network Of Workstations*, ou COW, pour *Cluster Of Workstations*.

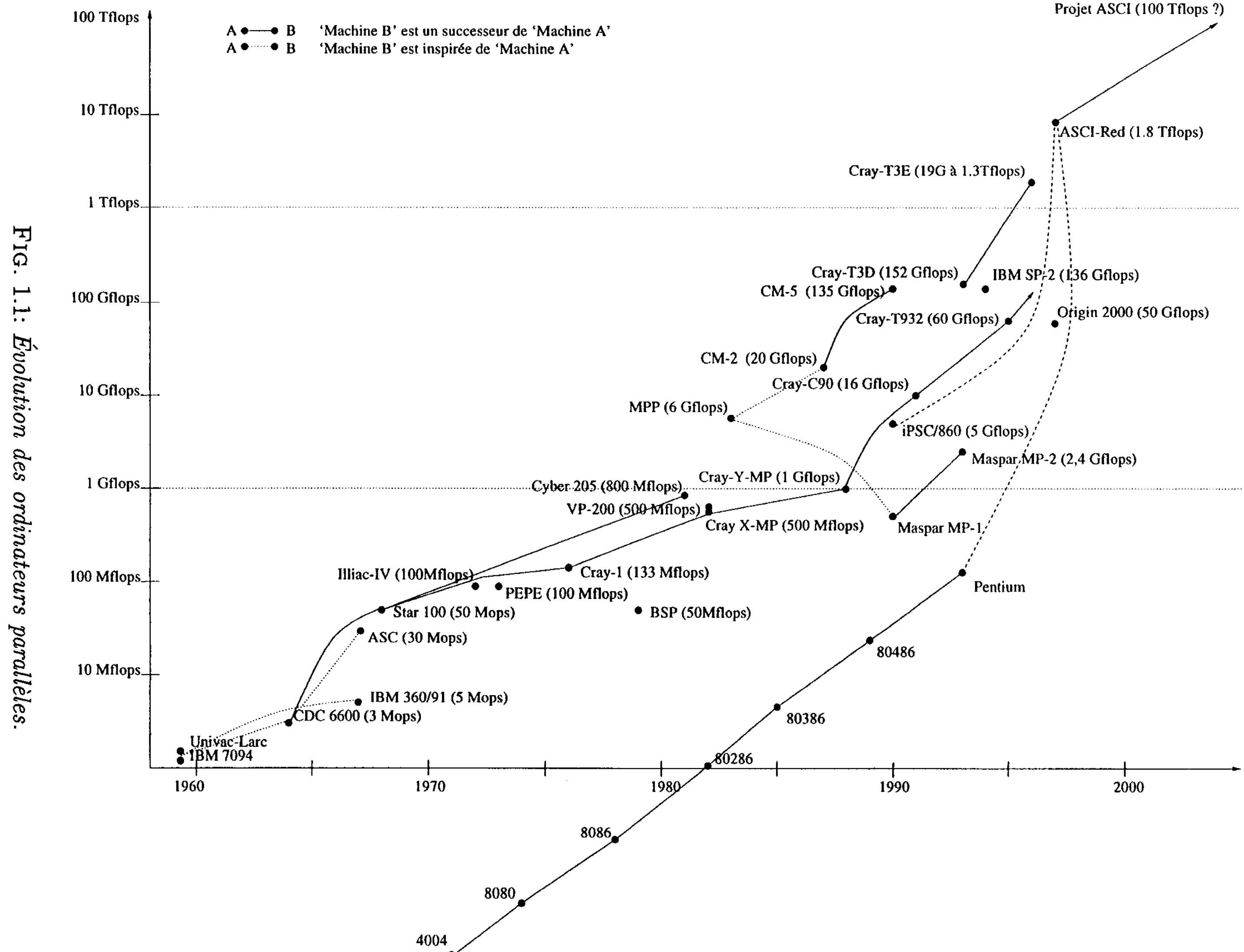


FIG. 1.1: *Évolution des ordinateurs parallèles.*

à celle des microprocesseurs séquentiels, mais avec un écart de trois ou quatre ordres de grandeur.

Ainsi, en 1993, les 500 machines les plus puissantes de la planète atteignaient la même puissance cumulée que la machine la plus puissante aujourd'hui à elle toute seule, soit 1,3 TeraFlops selon le *benchmark* LINPACK [Top].

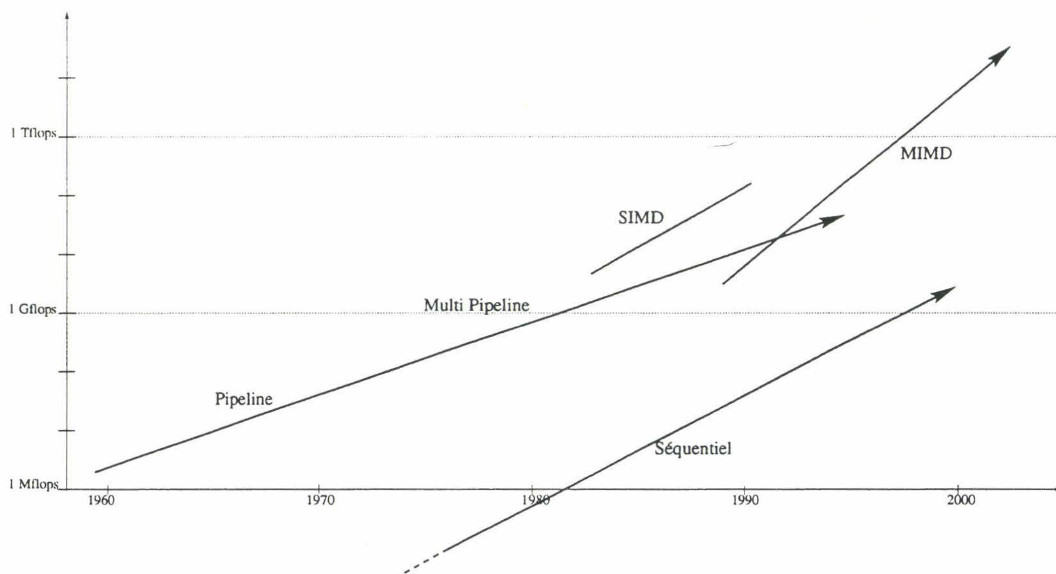


FIG. 1.2: *Évolution des architectures.*

Contrairement à la courbe de puissance des ordinateurs séquentiels, qui devrait avoir tendance à s'écraser du fait des limites de l'électronique, il semble que rien ne permette d'envisager actuellement une diminution de cette montée en puissance des architectures parallèles.

Il est clair que seules les machines de ce type permettent d'atteindre les performances demandées par les scientifiques pour leur donner la possibilité d'effectuer des calculs intensifs tels que nous les décrirons dans la section 1.4.

1.3.3 Les modèles d'exécution

Comme nous l'avons vu dans cet historique des machines parallèles, on discerne clairement trois tendances qui ce sont enchaînées : le pipeline, le SIMD et le MIMD. Ces tendances sont apparues au fil des progrès techniques et ne sont pas issues d'une réflexion théorique sur les modèles de programmation parallèle.

La modélisation des machines parallèles et l'apparition de principes de programmation sont donc postérieurs aux machines (tout comme le modèle de VON NEUMANN et les langages de programmation évolués ne sont apparus que bien après les premiers ordinateurs).

Les premiers travaux dans ce sens ont consisté à définir de façon plus formelle le comportement d'un programme lors de son exécution sur une machine parallèle donnée. On est arrivé ainsi à la définition de modèles d'exécution associés à chaque grande famille de machines parallèles.

Ces modèles d'exécution permettent de cerner les problèmes qui se posent lorsqu'on cherche à programmer sur une famille de machines parallèles donnée. Il est ainsi possible de définir des méthodes d'optimisation valables pour toute cette famille.

Le modèle d'exécution en pipeline

L'exploitation des machines pipeline pose des problèmes nouveaux par rapport à la programmation séquentielle. 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 du pipeline soit négligeable par rapport au temps total de calcul. Plus le nombre de données est important, plus on approche du gain idéal où le temps de calcul pour un élément du flot de données est divisé par le nombre d'étages du pipeline.

Du point de vue du programmeur, il s'agit donc de regrouper les données dans des vecteurs qui feront office de flot d'entrée.

De plus, il devient nécessaire de prendre en compte le problème des dépendances de données : une donnée modifiée dans l'algorithme ne peut être utilisée par la suite qu'à partir de sa sortie du dernier étage du pipeline. Dès que cet impératif n'est plus respecté, la chaîne doit être rompue en attendant la sortie de la donnée à réutiliser.

Les compilateurs ou des outils de pré-compilation doivent donc être capables d'optimiser l'ordonnancement des instructions pour réduire ces conflits. De plus, le langage peut proposer des directives qui permettent au programmeur d'assurer qu'il n'y a pas de contraintes de dépendance sur une instruction donnée et qu'il est donc possible de l'effectuer en pipeline.

En résumé, ce modèle d'exécution nécessite une programmation favorisant l'utilisation de vecteurs et les traitements « en blocs » de ces vecteurs.

Le modèle d'exécution SIMD

On a vu que l'architecture SIMD, fournit une accélération théorique égale au nombre de processeurs, mais que les nombreuses contraintes imposées par la synchronisation forte empêchent d'accéder à ce niveau de performances.

Pour réduire cet écart entre puissance théorique et effective, il faut donc être capable de minimiser certains conflits assez proches de ceux rencontrés dans le modèle vectoriel.

En effet, ce sont les traitements globaux, sur des ensembles de données réguliers, qui doivent être favorisés. Dans de telles architectures, il est même souvent préférable de répliquer une donnée dans toutes les mémoires des processeurs élémentaires plutôt que d'utiliser un code séquentiel.

Là encore, les compilateurs et les langages doivent donc fournir la possibilité d'optimiser les placements de données et les structures de contrôle.

Le modèle d'exécution MIMD

Dans le modèle MIMD, on dispose d'un certain nombre de nœuds de calcul, complètement désynchronisés. On peut donc y exécuter des programmes différents. Chaque programme a la

possibilité de communiquer avec les autres, soit par échanges de messages, soit en utilisant une zone de mémoire commune.

Cependant, il est alors nécessaire de gérer toute l'organisation des interactions entre les processeurs, y compris les synchronisations nécessaires pour éviter qu'une tâche n'utilise une donnée avant qu'une autre tâche ne l'ait produite.

Une façon plus simple d'utiliser ce modèle consiste à exécuter le même programme sur chaque nœud de la machine, mais en affectant à chacun une partie des données à traiter. Cette méthode, dite SPMD[6], permet donc de travailler comme en SIMD, sauf qu'il n'est plus nécessaire d'être complètement synchrone.

Ainsi, il devient possible d'occuper tous les processeurs, sans être obligé de les faire s'attendre à chaque instruction. On obtient donc un équilibrage des charges qui mène à un meilleur taux d'utilisation des processeurs.

En résumé, on arrive à deux façons complètement différentes de programmer ces machines, la seconde étant très proche de celle proposée dans les deux sections précédentes.

1.3.4 Les modèles de programmation

Les modèles d'exécution constituent une abstraction des machines parallèles, mais restent liés à leurs architectures. Pourtant, on a vu que ces modèles ont des points communs (traitement des données par blocs, ordonnancement ...). Il semble donc nécessaire de passer à un niveau d'abstraction encore supérieur, afin d'être complètement indépendant de l'architecture cible.

Pour cela, la notion de modèle de programmation est apparue. Cette fois, il s'agit d'exprimer un algorithme de façon à en faire ressortir les points qui pourront être utilisés pour améliorer la compilation vers un modèle d'exécution donné.

Le modèle de programmation est donc un cadre dans lequel peut s'exprimer un algorithme, c'est-à-dire une boîte à outils qui permet de décrire la façon dont un problème peut être résolu. Ensuite, le modèle d'exécution définit la façon dont un programme sera implanté sur l'architecture de la machine, c'est-à-dire qu'il décrit comment seront réalisés les concepts nécessaires au modèle de programmation.

Le modèle de programmation à parallélisme de tâches, aussi appelé parallélisme de contrôle, consiste à découper un algorithme en traitements élémentaires, puis à les organiser de façon à pouvoir en effectuer un maximum en parallèle. Les contraintes apparaissent alors au niveau des dépendances entre traitements.

Ces contraintes sont résolues, par exemple, en plaçant des barrières de synchronisation qui obligent à attendre la fin de certaines tâches avant de démarrer les suivantes.

Ce modèle de programmation n'est intéressant qu'au-dessus du modèle d'exécution MIMD.

Le modèle de programmation à parallélisme de données consiste à effectuer un traitement unique sur un certain nombre de données homogènes. L'algorithme reste donc

6. *Single Program, Multiple Data.* Un même programme sur plusieurs données.

globalement séquentiel, ce n'est qu'au niveau des opérations sur des ensembles de données tels que les tableaux que le parallélisme apparaît.

Un intérêt de ce modèle vient du fait qu'on garde un flot d'instructions unique. On n'a donc pas à gérer de problèmes de synchronisation.

On peut noter que ce modèle est dit parallèle dans le sens où les traitements effectués sur chacun des éléments des ensembles de données sont équivalents, qu'il y ait simultanéité ou pas. On peut donc parler de parallélisme de données dans un cadre d'exécution séquentielle.

Ce modèle est utilisable sur toutes les architectures. Un traitement de tableau peut être effectué en pipeline aussi bien qu'en SIMD ou en SPMD. Dans le dernier cas, il devient même possible de répartir les charges de calcul et de mettre en place des recouvrements entre calcul et communications.

En résumé, le modèle à parallélisme de données semble donc s'adapter efficacement à plus de machines et semble plus simple à manipuler.

Par contre, en général, le parallélisme de tâches peut être mis en place par des librairies, alors que le parallélisme de données, qui est un concept de plus haut niveau, nécessite une intervention au niveau du langage de programmation lui même.

En effet, le parallélisme de tâches laisse au programmeur le soin de placer, organiser et synchroniser les travaux. Ce modèle est donc plus simple à mettre en place et permet souvent de meilleures performances, mais il est plus complexe à maîtriser.

1.3.5 Conclusion

Cette section n'avait pas pour but de détailler en profondeur l'architecture des machines parallèles et leur histoire, ni d'expliquer en détail les techniques de programmation parallèle [7].

Nous avons simplement cherché à montrer que le parallélisme est devenu rapidement inévitable pour suivre la demande de puissance de certaines applications, notamment pour rendre calculable en temps raisonnable les problèmes des scientifiques.

On a vu que les modèles et les langages sont souvent apparus après les machines et encore actuellement beaucoup d'aspects ne sont pas encore unifiées.

Du point de vue de l'architecture, on semble s'orienter actuellement vers la solution MIMD, avec deux tendances : les machines dédiées et les réseaux de stations de travail.

Du point de vue logiciel, le choix entre mémoire partagée ou distribuée est encore ouvert.

Enfin, du point de vue du modèle de programmation, aucun modèle ne prend clairement l'avantage sur un autre.

On peut considérer le modèle à parallélisme de données comme une évolution du modèle à parallélisme de tâches, dans le sens ou le parallélisme y est exprimé à un plus haut niveau. On peut associer cette évolution à celle qui a eu lieu de l'assembleur vers les langages de programmation. Dans les deux cas, une abstraction plus importante de la machine sous-jacente permet une utilisation plus simple, au détriment de quelques pertes de performances.

7. Pour cela, le lecteur pourra se référer à des livres comme [HB85] qui fait référence en architecture depuis 10 ans et à des ouvrages plus récents comme [GUD96].

Cependant, là encore, des solutions hybrides sont parfois utilisées. Certains projets mettent en œuvre des groupes de programmes SPMD coopérant selon un parallélisme de contrôle.

Tout comme il a fallu longtemps pour que certains langages et familles d'outils s'imposent dans le monde du séquentiel, il est clair que beaucoup de travail reste à faire pour rendre accessible l'utilisation des machines parallèles aux non-spécialistes.

Ce n'est que lorsque certains modèles se seront vraiment imposés que les compilateurs deviendront suffisamment optimisés. Il sera alors possible d'obtenir des performances comparables à celles accessibles actuellement en programmant au niveau le plus bas. On a vu la même évolution avec les langages séquentiels, qui obtiennent actuellement des résultats comparables aux performances possibles en s'attaquant directement au niveau de l'assembleur.

Les principaux utilisateurs du parallélisme sont les scientifiques travaillant sur des modélisations nécessitant beaucoup de calcul. Nous allons maintenant présenter leur domaine d'activité afin d'essayer de cerner l'utilisation que font les scientifiques des machines parallèles. Pour cela, nous commencerons par présenter les problèmes auxquels ils sont confrontés en général et nous essaierons d'en déduire leur besoins en modèles et en outils d'aide à la programmation.

1.4 Le calcul scientifique

1.4.1 Origines

En général, modéliser un phénomène physique a principalement pour but d'être capable de prévoir son comportement. Pour cela, on met en équations le phénomène en passant par certaines simplifications afin de rendre les calculs abordables.

On peut par exemple exprimer la chute d'une pomme en suivant chacun de ses atomes, mais la loi de NEWTON suffit amplement pour déterminer sa trajectoire avec une quantité de calculs nettement inférieure.

Par contre, optimiser le profil d'une aile d'avion nécessite de modéliser finement le comportement de l'air. Plus on pourra découper l'espace autour de l'aile en petites sections, plus les résultats seront précis. De même, en physique corpusculaire, simuler une collision de particules est beaucoup moins coûteux qu'un test sur accélérateur et permet de « rejouer » son déroulement afin d'en observer les moindres détails. Malheureusement, les calculs deviennent alors complètement inabordables à la main.

Ainsi, pour de nombreux domaines d'application, on retrouve un certain nombre de classes de problèmes mathématiques à résoudre, mais dont la complexité est telle qu'on ne peut les utiliser que sur des cas d'école. Depuis quelques décennies, l'utilisation des ordinateurs permet d'effectuer de tels calculs sur des exemples de taille réelle. C'est pourquoi des modèles spécifiquement adaptés à une utilisation sur ordinateur sont apparus. Des algorithmes peu utilisés auparavant, car trop gourmands en calcul, ont été sortis des archives. On trouve également certains algorithmes qui ont été spécifiquement développés pour une utilisation sur ordinateur comme les FFT (Fast Fourier Transform), les algorithmes multi-grilles et l'algorithme Metropolis.

Dans le cas des résolutions d'équations linéaires, par exemple, la méthode du pivot de

GAUSS ne s'utilise quasiment jamais à la main car elle entraîne des calculs plus complexes que les méthodes de substitution, mais son côté systématique la rend bien plus facile et efficace à programmer. De même, les méthodes itératives ne sont utilisées que parce qu'elles sont adaptées au traitement informatisé.

L'utilisation de l'ordinateur est donc devenue incontournable. Cependant, bien qu'il permette d'atteindre des vitesses de calcul largement supérieures à quelques cerveaux humains autour d'un tableau, les scientifiques n'en ont jamais assez. Dès que les progrès leur permettent de calculer un nouveau modèle, ils le rendent encore plus complexe et demandent plus de puissance aux informaticiens.

C'est pourquoi le calcul scientifique est le domaine de prédilection du parallélisme. C'est presque uniquement pour des applications scientifiques que les architectures parallèles et les modèles de programmation associés sont développés[8].

1.4.2 Présentation

Dans divers domaines tels que la physique, la chimie et la mécanique, des modèles mathématiques sophistiqués de plus en plus complexes ont été mis au point. On retrouve dans chacun d'eux certains points communs tels que la résolution d'équations dans $\mathbb{R}^n$, l'optimisation, les équations aux dérivées partielles, la méthode des différences finies, la méthode des éléments finis ... L'étude et la généralisation de ces modèles adaptables à plusieurs types de problèmes ont entraîné l'émergence d'une nouvelle branche des sciences nommée *informatique scientifique*, ou *calcul scientifique*.

On peut définir le calcul scientifique comme étant *l'utilisation d'ordinateurs pour résoudre des problèmes scientifiques*. Cette discipline ne doit donc pas être confondue avec l'informatique (qui cherche à définir de nouvelles architectures et de nouvelles méthode d'utilisation de ces machines), ni avec la théorie et l'expérimentation (les branches traditionnelles de la science).

On remarque immédiatement que l'informatique scientifique est intrinsèquement multidisciplinaire: elle demande bien sûr la connaissance de l'informatique, mais aussi des différentes branches pour lesquelles elle doit s'appliquer. Une de ses difficultés est donc de s'adapter à la terminologie propre à chaque discipline, afin de pouvoir en isoler les points communs.

On peut donc considérer l'informatique scientifique comme une mise en commun des méthodes utilisées par les scientifiques de tous horizons pour leur fournir des outils génériques de calcul sur ordinateur.

En 1986, K. Wilson[Wil86] résuma les particularités qui identifient un problème de calcul scientifique :

- avoir des bases mathématiques solides,
- être insoluble par des méthodes classiques en temps raisonnable,
- avoir un domaine d'application réel,

8. L'utilisation des machines parallèles commence à se développer dans le domaine des bases de données, mais cette utilisation reste pour l'instant assez limitée.

– demander une connaissance profonde des sciences et techniques.

Un exemple typique est celui de la prévision météorologique : dès 1949, CHARNEY et ELIASSEN mettent au point un système non-linéaire bi-dimensionnel qu'ils adaptent afin de l'utiliser sur l'ordinateur le plus puissant de l'époque : l'*Eniac* [Neb95, pp 145-148]. Ce dernier possédait une vingtaine de registres de 10 chiffres et une table de 208 nombres à 6 chiffres, le reste du stockage se faisant sur cartes perforées.

Au bout de cinq semaines de calcul (et environ 100 000 cartes perforées), quatre prévisions à 24 heures et deux prévisions à 12 heures sont obtenues.

Il est évident que pour qu'une telle méthode soit utilisable, il faut être capable de finir les calculs *avant* la date de la prévision ! Mais les résultats obtenus sont d'une précision jamais atteinte à l'époque. Ce succès va entraîner la communauté météorologique vers l'utilisation de ce nouvel outil.

L'augmentation rapide des performances des machines et l'amélioration des environnements de développement, parallèlement à la création de modèles de plus en plus précis, mène à des résultats de plus en plus fiables, dans des délais de plus en plus courts.

Actuellement, Météo-France dispose d'un *Fujitsu VPP700E* de 26 processeurs vectoriels disposant chacun d'une mémoire de 2 Gigaoctets et atteignant 2,4 GigaFlops, le tout relié par un *cross-bar* à 670 Moctets/s dans chaque sens.

Cette machine permet de faire tourner en deux heures et demie un modèle de prévision utilisant un maillage de 27 niveaux de 100 000 points de grille chacun, couvrant la totalité du globe.

Bien que déjà impressionnante, cette machine sera remplacée fin 1999 par un autre modèle quatre fois plus puissant. Météo-France affinera alors les prévisions à 5 jours en utilisant un maillage de 200 000 points sur une trentaine de niveaux[9].

1.4.3 Les *grand challenges*

Comme on l'a vu précédemment, les scientifiques demandent toujours plus de puissance. Actuellement, certains modèles ne sont pas utilisés car tout jeu de données exploitable dépasse les capacités des ordinateurs les plus puissants.

Afin de cerner les besoins des chercheurs pour les quelques années à venir et fixer des buts à atteindre, un certain nombre de problèmes qu'on ne sait pas résoudre en un temps acceptable avec les technologies actuelles ont été répertoriés dans ce qu'on appelle les *grand challenges.*

On trouve parmi ces défis les problèmes de structure électronique de la matière, le séquençage du génome, la modélisation globale du climat, les études d'évolution de la pollution et les simulations de dynamique des fluides.

Aux États-Unis, le projet *High Performance Computing and Communications* regroupe une sélection de ces projets et plusieurs centres de recherche ont été créés spécialement pour avancer vers des solutions à ces problèmes.

9. merci à Dominique Birman, de Météo-France, pour toutes ces informations.

Ces solutions ne consistent pas seulement à construire des machines plus puissantes, mais nécessitent aussi l'étude de nouveaux systèmes d'exploitation, des langages, des compilateurs et des outils d'optimisation et de déverminage, des analyseurs de performances, des outils de communication et des environnements de mise en forme et de visualisation des données.

Il apparaît clairement que tous ces outils sont rendus indispensables par la volonté d'utiliser le parallélisme pour tenter de résoudre ces problèmes. Ainsi, l'étendue des besoins en outils montre que l'évolution de la puissance des machines doit être accompagnée du développement de toute une gamme d'outils permettant de rendre accessibles et conviviales de telles architectures.

En 1995, le président des États-Unis annonce la création du projet ASCI (*Accelerated Strategic Computing Initiative*) qui a pour but la mise au point de machines et de logiciels pour la simulation autour de l'armement nucléaire.

On sait réaliser certaines simulations actuellement, mais le but de ce projet est d'arriver à la simulation de systèmes physiques complets, tridimensionnels et à haute résolution d'ici 2010.

Pour cela, trois laboratoires et quelques constructeurs de super-ordinateurs sont engagés dans le développement de techniques ayant approximativement les même directions de recherche que les Grand Challenges.

L'un des premiers défis à relever, qui est déjà atteint, est le TeraFlops. La machine *ASCI Red*, regroupant 9192 processeurs *Intel* a en effet franchi la barrière cette année en atteignant 1,3 Tflops.

De même, en France, le développement des projets autour de la simulation (comme le laser MégaJoule de Bordeaux) témoignent d'objectifs semblables.

1.4.4 Conclusion

On vient de voir que des initiatives gouvernementales financent des projets mettant en œuvre l'informatique scientifique. Pour cela, les scientifiques ont besoin de puissance, mais aussi de moyens de manipuler cette puissance.

Il est clair que les scientifiques sont les principaux demandeurs de machines parallèles, mais leur métier ne consiste pas à se mettre constamment à jour des dernières évolutions des techniques du parallélisme.

Ils ont donc besoin de modèles, de langages et d'outils afin de pouvoir programmer les super-ordinateurs d'une façon relativement unifiée et le plus possible indépendante de la machine cible.

Afin d'introduire une étude plus précise de ces besoins, nous allons maintenant présenter l'un des outils de base du programmeur : les langages de programmation. Nous commencerons par un bref historique avant d'étudier plus précisément FORTRAN, le langage le plus utilisé par les scientifiques.

1.5 Les langages

1.5.1 Historique

On considère généralement que l'*Eniac*, conçu en 1946, est la première machine à pouvoir être appelée ordinateur [Wil97, Ch. 7.3]. Cependant, programmer cette machine consistait à câbler entre eux ses différents éléments. C'est pourquoi *IBM* revendique le titre de premier *vrai* ordinateur avec le *Ssec*, construit par l'équipe de W. ECKERT en 1948. En effet, cette machine lisait ses instruction sur une bande de papier ou en mémoire. Il s'agit donc de l'apparition de la première machine obéissant au modèle de VON NEUMANN et aussi de l'apparition du premier langage de programmation : le langage machine.

Ce n'est qu'autour de 1950, lors de la mise au point de *Edsac* par l'équipe de M. WILKES que l'assembleur apparaît. C'est notamment D. WHEELER qui permet ce grand pas, en inventant les notions de sous-programme, de code relogeable et de *bootstrap* chargeant un interpréteur d'assembleur [Wil97, Ch. 8.3.3].

Cependant, très vite, le besoin de pouvoir utiliser un ordinateur sans être un spécialiste de la machine se fait sentir, d'autant plus que tout changement de matériel oblige à reprendre à zéro l'apprentissage du langage spécifique à chaque modèle d'ordinateur.

En 1951, G. HOPPER crée sur *Univac I* le langage A0, dédié à la résolution de problèmes mathématiques. Ce langage est le premier pour lequel un compilateur est écrit.

À partir de ce moment, de nombreux langages de plus en plus évolués apparaissent à une cadence de plus en plus rapide (LISP (1958), ALGOL (1958), COBOL (1960), BASIC (1964), PL/1 (1964) ...). Les efforts d'unification qui mèneront à des langages normalisés comme COBOL et ADA ne suffisent pas à empêcher l'apparition d'innombrables langages plus ou moins spécialisés ([Kin] en répertorie 2350 !).

1.5.2 Fortran [Edg92, HPFa]

Ainsi, en 1954, une équipe d'*IBM* dirigée par J. BACKUS commence le développement d'*un langage dédié à la traduction de formules scientifiques ou algébriques dans une forme utilisable par un ordinateur*. Ce travail aboutit à une première version en 1957, appelé FORTRAN, pour *FORmula TRANslator*.

Comme cette version est dédiée à l'ordinateur sur lequel il a été développé, l'*IBM 704*, plusieurs de ses particularités sont directement inspirées des fonctionnalités de la machine (longueur des noms de variables et `if` arithmétique).

L'année suivante, la deuxième version ajoute le `if` logique, les *subroutines* et les fonctions.

La version IV, sortie en 1962 (la version III est restée « interne » à *IBM*), ajoute la notion de variables logiques et connaît un grand succès. Fortran commence alors à devenir le langage de référence dans le domaine du calcul scientifique.

IBM continue ensuite à ajouter des fonctionnalités au langage, notamment pour l'étendre au domaine de l'informatique de gestion et aboutit ainsi à FORTRAN V et VI, qui deviennent ensuite un nouveau langage : PL/1.

Parallèlement à cette évolution, l'*American Standards Association* commence à travailler

sur une version standard du langage, qui permettrait de le rendre complètement portable. Ceci aboutit à FORTRAN 66, puis à FORTRAN 77 en 1978 [Ame78].

Aussitôt après cette dernière normalisation, l'ASA, devenue entre temps l'ANSI, commence à étudier une nouvelle génération de FORTRAN, d'abord intitulée FORTRAN 8X, puis FORTRAN 90 [AC92].

Contrairement aux précédentes évolutions qui consistaient en quelques ajouts et modifications de divers points du langage, FORTRAN 90 constitue une refonte profonde du langage.

En effet, l'aspect général du langage n'avait pas du tout évolué depuis 1954 et de nombreux outils syntaxiques utilisés par la plupart des langages apparus depuis manquaient à FORTRAN. Voici une liste des principaux manques de FORTRAN 77 et des solutions qui y ont été apportées :

Syntaxe : En FORTRAN 77, les lignes sont limitées à 72 caractères. Ceci est directement issu de l'époque héroïque des cartes perforées et n'a plus aucune utilité aujourd'hui. De plus, le numéro de colonne des caractères a son importance (labels, commentaires ...), les identificateurs sont limités à 6 caractères et uniquement en majuscules.

FORTRAN 90 autorise une syntaxe beaucoup plus libre et allonge la taille limite des lignes à 132 caractères. De plus, FORTRAN 90 ajoute des structures syntaxiques (`select`, `while` ...) et améliore les existantes (`do...enddo`)

On peut remarquer qu'une instruction doit quand même tenir sur une ligne, ou que la suite de la ligne doit être annoncée par un caractère `&` en fin de ligne. Cet archaïsme n'a pourtant plus cours dans les langages de programmation actuels. Ce problème vient du fait que le langage n'utilise pas de symbole de séparation ou terminaison d'instruction et que c'est donc la fin de ligne qui sert de marqueur.

Notions de parallélisme au niveau du langage : FORTRAN est fait pour aller vite et aujourd'hui, performance implique parallélisme. En FORTRAN 77, pour utiliser le parallélisme, il faut utiliser des librairies spécifiques, ce qui est peu confortable à utiliser pour des non-spécialistes.

FORTRAN ajoute une notion de parallélisme dans ses structures syntaxiques en permettant d'accéder aux tableaux en tant qu'entité globale et d'effectuer des opérations dessus sans expliciter un par un les accès aux éléments.

Ainsi, deux tableaux `A` et `B` de même taille peuvent être additionnés terme à terme en écrivant simplement `A+B`. Cependant, c'est au compilateur que revient la charge de mettre en place ces constructions parallèles comme il le veut (peut?).

On peut remarquer que FORTRAN s'est donc orienté vers une solution à parallélisme de données.

Allocation dynamique de mémoire : FORTRAN 77 ne permet aucune allocation dynamique, ce qui rend difficile l'utilisation de variables temporaires et oblige à définir d'avance toutes les structures « suffisamment grosses ». FORTRAN 90 ajoute cette fonctionnalité et permet la manipulation de pointeurs et donc de structures chaînées, de sections de tableaux ...

Portabilité des formats numériques : Beaucoup de compilateurs FORTRAN améliorent la gestion des formats de nombres pour accroître la précision. FORTRAN 90 unifor-

mise ces extensions en les intégrant dans le langage par un attribut de description des types (construction `kind`).

Définitions de types et de structures : La plupart des langages permettent de définir des types de données « sur mesure » ou groupés. FORTRAN 90 définit pour cela les types `derived`.

Récursivité : En FORTRAN 77, la récursivité ne peut qu'être simulée à travers une gestion explicite de pile de données. En FORTRAN 90, la gestion des appels de sous-programmes autorise la récursivité.

Modularité : FORTRAN 90 introduit une notion de module qui permet de définir des paquetages contenant une interface et une implémentation séparées, à la ADA. Il est ainsi possible également de partager des variables entre modules, autrement qu'en utilisant les `common` sources d'erreurs difficiles à corriger.

Interfaces des procédures : En FORTRAN 90, la définition d'un tableau ou d'une section contient sa géométrie; il est donc possible de le passer en argument de procédure sans avoir à passer explicitement sa taille. De plus, les contrôles de types sont plus stricts et la taille d'un tableau passé en arguments peut être héritée pour définir une variable locale.

En 1995, la norme a été revue afin de mettre au point quelques détails oubliés ou insuffisamment précis dans la première version. Cette mise à jour a été publiée et est donc devenue la norme FORTRAN en vigueur. Elle est connue sous le nom de FORTRAN 95.

Comme c'est la dernière version normalisée de FORTRAN et qu'il y a compatibilité ascendante avec les autres, c'est à cette version que nous nous référerons dans la suite du texte quand nous parlerons de FORTRAN sans préciser la version.

La principale innovation de FORTRAN 95 est l'ajout du `forall`. Dans FORTRAN 90, la possibilité d'effectuer des opérations entre tableaux souffre d'un manque d'expressivité.

Par exemple, si on veut affecter à un tableau bi-dimensionnel `A`, de taille 100×50, une partie d'un tableau B de taille 100×100, on peut écrire : `A=B(:, 2:51)`. Par contre, il est impossible d'affecter directement la transposée d'un bloc, ou une valeur dépendant de l'indice.

Le `forall` apporte cette possibilité, en permettant d'exprimer l'affectation avec des indices nommés :

```
forall (i=1:100,j=1:50) A(i,j)=B(j,i)+i+j
```

Il permet aussi d'effectuer des opérations plus complexes comme des sommes terme à terme, des affectations à travers une indirection ou des appels à une fonction pour chaque terme. Dans ce dernier cas, il faut assurer au compilateur que la fonction n'a aucun effet de bord (pas d'utilisation de variables globales, d'entrées/sorties ...) et qu'il est donc possible de l'appeler pour chaque itération du `forall`, dans n'importe quel ordre, voire en parallèle. Cette assertion est effectuée par un attribut de fonction défini par le mot clé `pure`.

La sémantique des opérations de tableau et du `forall` est plus complexe qu'il n'y paraît à première vue. En effet, l'ordre d'exécution des itérations peut avoir un effet sur le résultat.

Prenons l'exemple suivant :

```
forall(i=2:10) A(i)=A(i-1)
```

Selon l'ordre d'évaluation des itérations, le résultat sera une recopie de `A(1)` sur tout le vecteur, un décalage des valeurs, ou à peu près n'importe quoi comme le montre la figure 1.3.

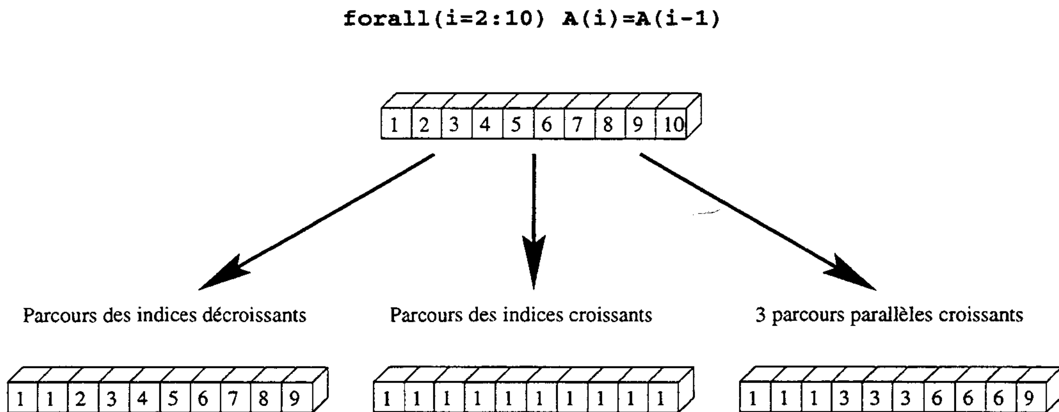


FIG. 1.3: *forall sans sémantique.*

Pour donner un comportement déterministe au langage, la sémantique choisie consiste à évaluer tous les termes droits, puis à affecter tous les termes gauche ensuite (voir figure 1.4). Ceci oblige à plus de synchronisation entre les étapes et passe éventuellement par l'utilisation d'une variable temporaire qui entraîne une perte de performance, mais assure un résultat déterministe.

Le programmeur est donc assuré que l'exemple précédent effectue un décalage à droite du vecteur.

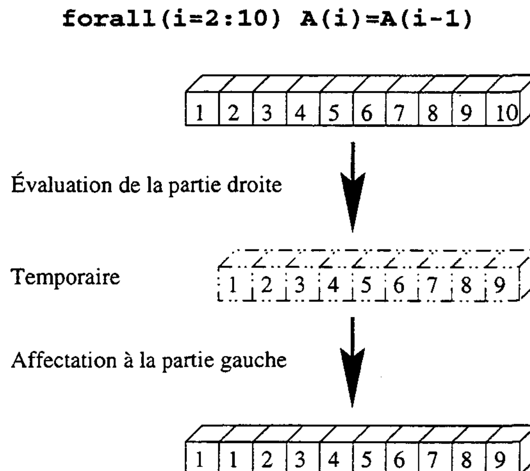


FIG. 1.4: *Sémantique du forall en* FORTRAN *95.*

FORTRAN 95 constitue donc une évolution de FORTRAN assez importante, mais qui était devenue nécessaire. La syntaxe même, tout droit sortie des contraintes des cartes perforées, donnait une allure antédiluvienne au langage. De plus, ce lifting a permis d'intégrer des

fonctionnalités devenues classiques qui en font un langage moderne. Pour sa prochaine version, FORTRAN 2x[10], il est prévu d'y ajouter des aspects orientés objet.

1.5.3 HPF

Parallèlement aux diverses normalisations, des versions spécifiques de FORTRAN ont été développées. Ces variantes sont principalement des extensions pour un modèle de programmation spécifique, ou des versions dédiées à une machine.

On peut citer dans la première catégorie FORTRAN-M [FC94] qui permet la gestion de processus et de communications, FORTRAN-S [BKP93] qui permet un parallélisme de contrôle sur machines à mémoire partagée, ainsi que VIENNA FORTRAN [CMZ92] et FORTRAN-D [HKK+91], deux versions dédiées au parallélisme de données.

Dans la deuxième catégorie, chaque constructeur a développé son FORTRAN propriétaire comme CM-FORTRAN[Thi91] pour la *Connection Machine*, MP-FORTRAN[Mas91] pour la *Maspar* et CFT dédié au vectoriel pour les machines *Cray* [Cra84].

La plupart de ces « FORTRAN étendus » sont en rapport avec le parallélisme et c'est dans l'espoir d'unifier tous ces efforts d'amélioration de Fortran qu'un regroupement de constructeurs et de vendeurs de compilateurs a été fondé : le *High Performance Fortran Forum* [HPFb].

Nous allons maintenant détailler l'historique et les fonctionnalités de ce qui est sorti de ce forum.

1.6 HPF

1.6.1 Historique

Après une première réunion du *High Performance Fortran Forum* lors de *SuperComputing'91*, de nombreuses discussions ont abouti à un premier brouillon de définition de HIGH PERFORMANCE FORTRAN version 1.0 (HPF v1.0), officiellement distribué en mai 1993 [For93].

Les principaux buts à atteindre lors de cette discussion étaient de définir un langage de programmation dédié au calcul scientifique sur ordinateur haute performance et respectant les restrictions suivantes :

- être compatible avec FORTRAN;
- être suffisamment généraliste pour ne pas refléter uniquement un type d'architecture ou une machine précise;
- pouvoir proposer à l'utilisateur des fonctionnalités facilement abordable pour mettre en œuvre le parallélisme, de façon à garantir un minimum de performances, quelle que soit la machine cible.

Malgré la volonté d'être généraliste, le langage a été fortement orienté, dès le début, vers le parallélisme de données, mettant de côté les notions de parallélisme de tâches, celles-ci étant

10. Remarquons que l'ANSI s'est laissée 3 chiffres de marge pour la date de publication !

considérées trop difficiles à manipuler et quasiment impossibles à exprimer à un haut niveau sans sacrifier les performances.

La version 1.0 définit HPF comme étant un FORTRAN 90 auquel sont ajoutés un ensemble de directives permettant de spécifier des conseils de parallélisation et un ensemble de fonctions intrinsèques principalement pour manipuler les tableaux. Le langage reprend principalement les spécifications de VIENNA FORTRAN.

Le compilateur est libre dans le choix du modèle d'exécution du code généré, mais le modèle SPMD est celui qui correspond le mieux au modèle de programmation du langage. Au niveau d'HPF, les communications sont complètement implicites : pour chaque instruction, c'est au compilateur que revient la charge de réunir tous les opérandes nécessaires sur un même nœud de calcul.

La version 1.0 définit également un sous-ensemble du langage, le *subset HPF*, qui constitue le noyau minimum du langage. Ainsi, les auteurs de compilateurs disposent d'un cahier des charges minimal qu'ils peuvent ensuite étendre petit à petit.

Après une nouvelle série de réunions, discussions et mises au point, une autre version de travail est publié, puis, en 1997, la version 2.0 paraît [For97].

Cette version est divisée en deux parties.

La première, à quelques modifications près, définit le langage comme étant le *subset* de la première version. Seuls quelques éléments sont renvoyés dans les extensions (directives dynamiques) et quelques détails sont simplifiés (`inherit` et comportements implicites).

Dans la deuxième partie, toute une série *d'extensions approuvées* sont ajoutées. Cette partie reprend la plupart des fonctionnalités de la première version qui n'étaient pas incluses au *subset* et y ajoute d'autres fonctionnalités apparues intéressantes lors des discussions (distributions généralisées, notions de sous-ensembles de processeurs ...).

Ces extensions sont des « idées à creuser » pour les versions futures et servent à uniformiser les éventuelles implémentations dans les compilateurs. On peut résumer le principe de ces extensions par *"vous n'êtes pas obligés de savoir compiler ça, mais si vous voulez essayer, il faut que ça s'écrive comme ça"*.

Comme entre les deux versions d'HPF, la norme de FORTRAN a été réactualisée, quelques propositions d'HPF v1.0 sont devenues standard dans FORTRAN et ont donc été supprimées de HPF v2.0. C'est notamment le cas du `forall` et de l'attribut `pure`.

La figure 1.5 résume l'imbrication des différentes versions.

1.6.2 Les directives

Le langage HPF contient donc toutes les fonctionnalités de FORTRAN, auxquelles s'ajoutent des directives de parallélisation, de placement de données et des fonctions intrinsèques.

La syntaxe des directives est faite de façon à ce qu'un compilateur FORTRAN les voit comme des commentaires, afin de ne pas les traiter comme des erreurs. Ainsi, on peut compiler un source HPF sur un compilateur FORTRAN.

D'un autre côté, si dans le futur HPF est admis comme nouvelle norme de FORTRAN,

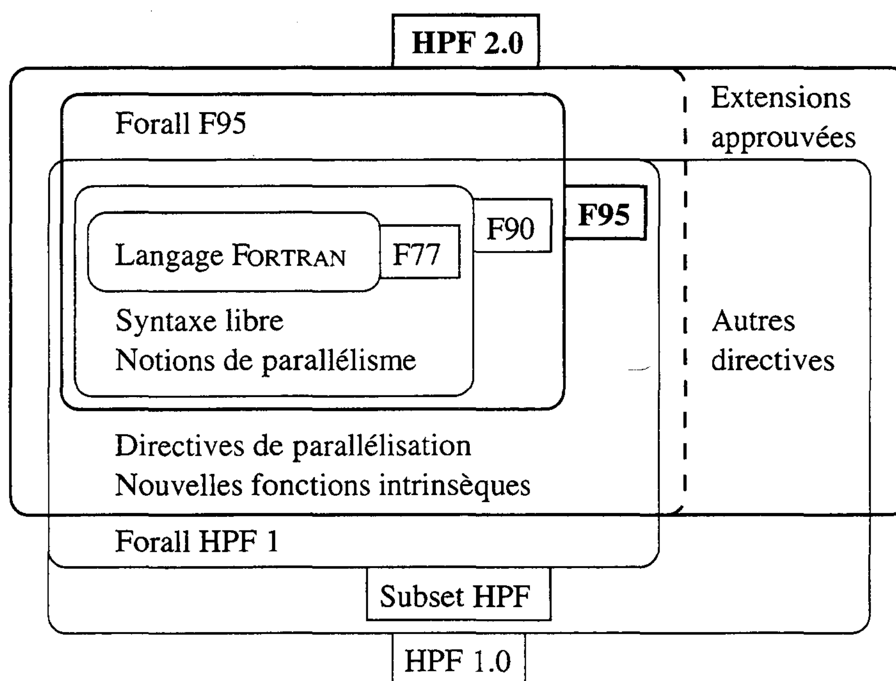


FIG. 1.5: *Les générations de* FORTRAN.

il suffit d'ôter les marques de commentaire pour en faire des directives cohérentes avec la syntaxe générale de FORTRAN.

Par exemple, l'extrait de code suivant contient une directive à la troisième ligne, spécifiant la façon dont le tableau M doit être distribué.

```
program bidon
  real, dimension (10,10) :: M
  !HPF$ distribute (Cyclic,*) :: M
end program
```

Il est important de noter que HPF ne définit que des directives, c'est-à-dire des conseils fournis au compilateur pour générer un code plus efficace. Si le compilateur ne sait pas en profiter, ou considère qu'il sait mieux faire, rien ne l'oblige à respecter ces directives.

1.6.3 Les constructions parallèles

Comme il a été vu précédemment, FORTRAN 95 introduit la possibilité d'effectuer des opérations entre tableaux et définit un `forall` qui permet de généraliser ce type d'opérations.

Cependant, le fait de devoir opérer en deux temps (évaluation des parties droites, puis affectation) peut entraîner une perte d'efficacité.

De même, dans le cas d'une boucle séquentielle, il est parfois possible de paralléliser les itérations, mais ce type d'optimisation est très complexe à détecter.

Ainsi, dans le cas d'une affectation à travers d'un vecteur d'indirection, si le vecteur ne

contient pas de doublons, l'utilisateur sait qu'il n'y a aucune dépendance entre les itérations, mais comme cette opération dépend des valeurs d'un vecteur, le compilateur n'a pas les moyens de le détecter.

C'est pourquoi HPF fournit la directive `INDEPENDENT`, qui donne un moyen de préciser au compilateur qu'il n'y a pas de dépendances entre les itérations successives d'une boucle `do...enddo` ou d'un `forall` et qu'il peut donc paralléliser l'opération sans risque.

Il est important de noter que si le programmeur assure à tort au compilateur qu'il n'y a pas de dépendances, le résultat sera un programme faux, voire indéterministe (voir l'exemple de `forall` sans sémantique, figure 1.3 page 33)

1.6.4 Les placements

Dans le modèle de programmation à parallélisme de données, la répartition des données à travers les différents nœuds de calcul est cruciale. Définir ce placement automatiquement lors de la compilation est reconnu comme étant un problème difficile. Il est donc nécessaire de pouvoir spécifier cette répartition au niveau du langage si on veut obtenir un minimum de performances. De plus, comme HPF n'est pas dédié à une machine particulière, il faut un moyen souple et portable de définir un placement quelle que soit l'architecture matérielle cible.

Les placements influent sur les performances à deux niveaux : la répartition de la charge des processeurs et la quantité de communications nécessaires à la récupération des opérandes.

En effet, pour chaque instruction, le code doit rassembler les opérandes sur un même nœud de calcul, effectuer l'opération, puis ranger le résultat. Si les données sont organisées de telle sorte qu'un maximum d'opérandes sont déjà présents sur le processeur qui doit effectuer l'opération, on gagne un temps de communication précieux. De plus, si ces données sont placées de telle sorte que tous les processeurs ont une charge de travail comparable, le parallélisme est maximum (voir figure 1.6).

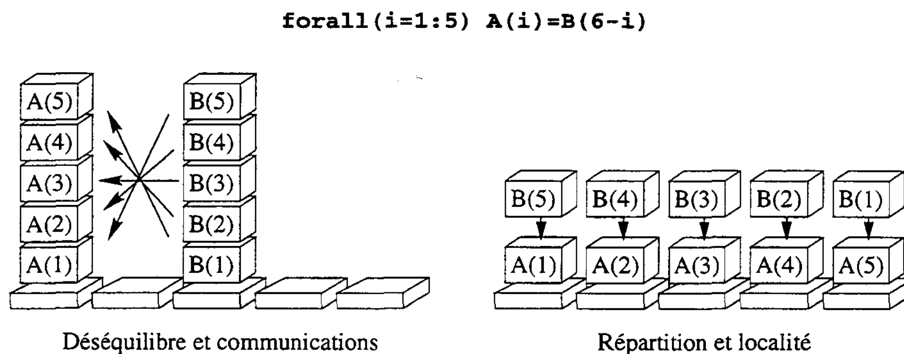


FIG. 1.6: *Effets des placements en charge et en communications.*

En HPF, le modèle de placement des données est donc défini en deux parties :

- Les *alignements* spécifient le placement des tableaux les uns par rapport aux autres et permettent donc de réduire les communications implicites;

- Les *distributions* spécifient la répartition de ces ensembles de données et permettent donc d'équilibrer la charge des processeurs.

Les alignements : Pour spécifier la façon dont les données doivent être placées les unes par rapport aux autres, HPF introduit la notion de `template`. Un `template` est une géométrie, c'est-à-dire un tableau qui n'a pas de type, mais juste un nombre de dimensions et une taille.

Ces `templates` servent de référentiel pour spécifier les placements de tableaux. De plus, pour simplifier l'écriture, tout tableau est associé à un `template` implicite de même géométrie sur lequel il est aligné.

Les alignements consistent donc à définir la façon dont un tableau doit être aligné sur un `template`. Cette spécification doit être utilisée par le compilateur pour qu'il place les données de telle sorte que tout ce qui est aligné sur un même élément de `template` soit placé sur le même processeur physique.

Dans l'exemple suivant, on précise que le tableau `A` doit être aligné sur le tableau `B` (c'est-à-dire le `template` implicite qui lui est associé) de telle façon que pour tous `i` et `j`, `A(i,j)` soit aligné avec `B(*,j,i*2-1)` :

```
dimension(10,10), real :: A
dimension(10,20,20), real :: B
!HPF$ align A(i,j) with B(*,j,i*2-1)
```

L'étoile indique que chaque élément de `A` doit être répliqué sur la totalité de la dimension correspondante de `B`.

Graphiquement, cet alignement peut être visualisé comme montré dans la figure 1.7.

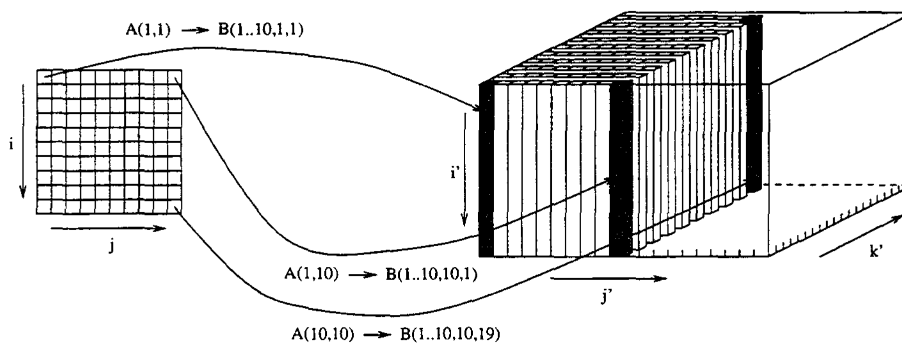


FIG. 1.7: *Exemple d'alignement.*

Les alignements peuvent également contenir des spécifications d'effondrement d'une dimension, comme montré dans l'exemple suivant :

```
dimension(10,10), real :: A
dimension(10), real :: B
!HPF$ align A(i,j) with B(11-j)
```

Le fait que l'indice `i` ne soit pas cité dans la partie droite de la spécification indique que la totalité de chaque colonne de `A` doit être projetée comme un seul élément.

Ainsi, `A(1,8)`, `A(2,8)` ... `A(10,8)` sont tous projetés sur `B(3)` (voir figure 1.8).

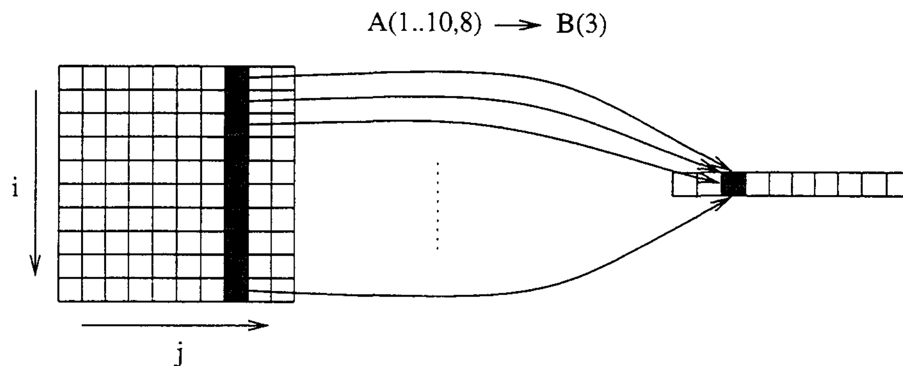


FIG. 1.8: *Autre exemple d'alignement.*

Les distributions : Avant de définir une distribution, il est évidemment nécessaire de préciser sa cible. Pour cela, HPF définit une structure abstraite, le `processors`, qui modélise une machine cible virtuelle. Sa structure est une grille multidimensionnelle et homogène de processeurs élémentaires nommés *processeurs abstraits.*

Rien ne définit la structure des liens de communication entre ces processeurs. Les placements ne permettent donc d'optimiser que les nombres de mouvements de données, sans savoir si une communication avec un voisin est plus ou moins coûteuse qu'avec un processeur à l'autre bout de la grille.

De plus, le placement des processeurs abstraits sur les processeurs physiques est à la charge du compilateur. HPF n'impose qu'une seule restriction : le compilateur doit au moins accepter les `processors` d'un seul élément et ceux ayant autant d'éléments qu'il y a de processeurs physiques.

Pour cela, quelques fonctions intrinsèques permettent, à l'exécution, d'obtenir des renseignements sur la géométrie effective de la machine cible. Il est donc possible de déclarer un `processors` dont la taille est déterminée par un appel à une telle fonction. Ainsi, il n'est pas nécessaire de connaître la topologie de la machine cible lors du développement d'un programme HPF.

Une distribution correspond à une répartition régulière des éléments d'un tableau ou d'un `template` sur les éléments d'un `processors`. Ces répartitions sont faites pour chaque dimension de l'objet source sur les dimensions successives du `processors` destination et peuvent avoir un des types suivant :

`block` : On découpe l'objet source en autant de blocs de même taille qu'il y a d'éléments dans la dimension de la cible, et on place successivement chaque bloc sur l'élément correspondant.

`cyclic` : On place successivement chaque élément de la source sur l'élément de la cible de même rang et, arrivé au dernier élément de la cible, on repart du premier.

`cyclic(`n`)` : On utilise la même méthode que pour la distribution `cyclic`, mais en prenant les éléments de la source par blocs de n éléments.

`block(n)` : C'est encore le même principe, mais ce type de distribution assure, en plus, que n est suffisamment grand pour qu'on épuise les éléments de la source avant d'entamer un deuxième tour dans les éléments de la cible.

`*` : Ce type correspond à l'effondrement vu pour les alignements. Chaque colonne de cette dimension de la source est entièrement placée sur le même processeur abstrait.

La figure 1.9 montre graphiquement la signification de chacun de ces types de distributions.

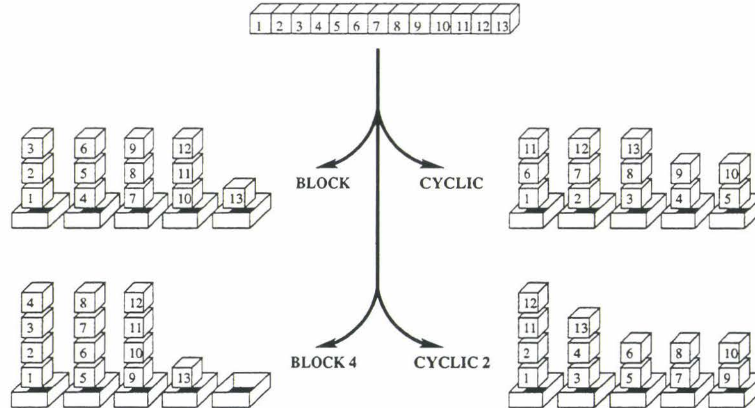


FIG. 1.9: *Exemples de distributions.*

Directives Dynamiques : Les directives d'alignement et de distribution permettent de définir la façon dont les tableaux doivent être placés le long des processeurs abstraits. Ces spécifications doivent être vues comme des attributs fixés lors de la déclaration des tableaux, c'est-à-dire qu'ils sont figés de la même manière que pour la taille ou le type de ces tableaux.

Pourtant, certains types d'algorithmes se déroulent en plusieurs phases pour lesquelles les accès aux données ne se font pas selon les mêmes motifs. Il est donc intéressant de pouvoir modifier le placement des données en cours d'exécution.

Pour cela, HPF propose des directives dynamiques de placement, qui permettent de spécifier un nouveau placement à un endroit donné du programme. Ces directives sont donc considérées comme des instructions exécutables et non comme des attributs.

Ces directives sont le `redistribute` et le `realign` et utilisent la même syntaxe que le `distribute` et l'`align`.

Malgré cette ressemblance, ces deux types de directives sont complètement différents dans leurs conséquences. Les directives statiques expliquent au compilateur comment et où il doit réserver l'espace mémoire sur chaque processeur abstrait. Par contre, les directives dynamiques demandent explicitement un déplacement des données pendant l'exécution.

Dans HPF 2.0, ces directives font partie des extensions approuvées.

Interfaces procédurales : Un autre problème se pose lors de l'appel à des procédures ou à des fonctions : comment les paramètres du sous-programme doivent-ils être placés ?

Pour cela, HPF prévoit trois formes de directives :

prescriptive : La directive définit explicitement le placement que doit avoir le paramètre formel. Si celui-ci a un placement différent de celui du paramètre effectif, c'est au compilateur d'effectuer les replacements nécessaires, puis de faire l'opération inverse au retour.

Ce type de placement s'effectue en utilisant des directives classiques lors de la déclaration des paramètres formels. Ces directives sont donc statiques au niveau de la procédure, mais impliquent en réalité un replacement équivalent à un appel de directive dynamique.

descriptive : Dans ce type de directive, le programmeur indique le placement du paramètre, mais assure qu'il est déjà placé comme indiqué et qu'il n'y a donc pas de replacement à effectuer.

L'assertion est dite faible, c'est-à-dire que le compilateur doit quand même vérifier que le paramètre est réellement déjà bien placé. Si ce n'est pas le cas, il doit le replacer comme dans le type prescriptif.

Ce type de placement s'indique en précédant les spécifications par un *.

transcriptive : Cette fois, le programmeur veut que le paramètre reste placé tel qu'il était chez l'appelant.

Ce type de placement s'indique en remplaçant les spécifications par un *.

Dans l'exemple suivant, les paramètres `A` et `B` doivent être distribués de façon `block` et `cyclic`, mais l'utilisateur assure que c'est déjà le cas pour `B`. `C` doit être distribué de la même façon que le paramètre effectif.

```
subroutine bizarre(A,B,C)
 real, dimension(10) :: A,B,C
!HPF$ distribute A(block)
!HPF$ distribute B *(cyclic)
!HPF$ distribute C *
 . . .

program appelant
 real, dimension(10) :: X,Y,Z,T
!HPF$ distribute X(block)
!HPF$ distribute Y(cyclic)
!HPF$ distribute Z(cyclic(2))
!HPF$ distribute T(cyclic(3))

bizarre(X,Y,Z)                appel 1
 . . .
bizarre(Y,Z,T)                appel 2
 . . .
```

Lors du premier appel, `X` a déjà la même distribution que `A`, le code n'a donc rien à faire; il en est de même pour `Y` et `B`, l'assertion est donc exacte; pour `C`, le code doit le gérer comme `Z`, c'est-à-dire avec une distribution `cyclic(2)`.

Lors du deuxième appel, `A` doit être redistribué de `cyclic` vers `block`; l'assertion sur `B` est fausse et il faut donc maintenant le redistribuer également; enfin, `C` garde la distribution de `T`.

En résumé :

- La première déclaration impose une distribution. Il y aura donc généralement redistribution en entrée et en sortie de procédure.
- La deuxième donne une indication au compilateur. Ainsi, il peut mettre en place un test afin de ne pas redistribuer quand l'assertion est exacte.
- La dernière solution laisse la distribution telle qu'elle est. On gagne donc forcément du temps lors de l'appel et du retour. D'un autre côté, cette solution oblige à générer un code qui puisse s'adapter à n'importe quelle distribution. Ceci impose certainement un code plus complexe et donc moins performant.

1.6.5 Les compilateurs HPF

Une fois que le consortium s'est mis d'accord sur une version de HPF, encore faut-il écrire des compilateurs pour l'utiliser.

Actuellement, les compilateurs commerciaux principaux sont ceux de *Portland Group Inc.*, *NAG*, *IBM* et *DEC*. Les compilateurs d'*IBM* et *DEC* sont respectivement développés pour les *SP-2* et les machines à base d'*Alpha*. Ceux de *PGI* et *NAG* peuvent générer des codes pour plusieurs types d'architectures dont les *Intel*, les *Cray* et les *IBM*, ainsi que du code utilisant PVM ou MPI.

De plus, quelques laboratoires ont développé des compilateurs universitaires. Citons HPFC, développé à l'École des Mines de Paris par Fabien COELHO, VIENNA FORTRAN, qui est très proche d'HPF (HPF est directement inspiré de VIENNA FORTRAN et FORTRAN-D) et sHPF, qui reconnaît un subset assez complet.

Citons également ADAPTOR, écrit par l'équipe de Thomas BRANDES [Bra96]. Ce compilateur génère du code FORTRAN 90 effectuant des appels à des fonctions de passage de messages. Les fonctions utilisées sont choisies selon l'architecture cible (PVM, MPI ou bibliothèque spécifique à la machine), via une bibliothèque générique de gestion de tableaux distribués : la `dalib` (Distributed Arrays LIBrary).

Des expériences de portage d'applications ont montré qu'ADAPTOR permet d'obtenir des performances comparables à celles des compilateurs commerciaux [BZBB98b].

Nous reparlerons de cet outil dans la partie 3.3.4 page 94.

1.6.6 Conclusion

HPF est donc une extension de FORTRAN, à la compatibilité ascendante totale et qui permet d'optimiser le comportement parallèle d'un programme. Il semble donc idéal pour le portage de programmes existants vers des machines parallèles.

De plus, il apporte un niveau d'abstraction suffisamment haut pour permettre l'exécution d'un même programme sur différentes architectures sans avoir à y changer quoi que ce soit.

Plusieurs expériences de portage ont montré qu'il était beaucoup plus facile de porter une application scientifique vers HPF que vers une solution à passage de messages [BZBB98a].

Cependant, un portage efficace d'un programme FORTRAN existant nécessite un « nettoyage » des sources. Les vieilles applications, écrites en FORTRAN 77, contiennent des « bidouilles » nécessaires pour contourner les manques d'allocation dynamique, de partage de variables entre procédures et de gestion interprocédurale des tableaux multidimensionnels.

Ces codes utilisent donc des constructions `common` et des linéarisations de tableaux qui empêchent d'utiliser les directives de placement. De plus, les `goto` et autres constructions syntaxiques peu élégantes empêchent la détection de boucles parallélisables.

Une part importante du portage consiste donc à transcrire le programme en FORTRAN 95 propre. Des outils permettent d'aider le programmeur dans cette tâche [For], mais une grande part du travail doit être faite à la main.

D'un autre côté, les expériences de portage précitées ont montré des performances plus faibles pour les programmes HPF. On peut espérer que ce désavantage soit surtout dû au fait qu'HPF est un langage jeune et que les compilateurs, encore peu nombreux, ne sont pas encore arrivés à un bon niveau d'optimisation.

De plus, un portage vers HPF peut se faire par étapes, en gardant constamment un programme fonctionnel, alors qu'un portage vers le passage de messages nécessite une refonte totale du programme.

C'est pourquoi on peut s'attendre à une bonne évolution de ce langage dans un avenir proche et développer des outils autour d'HPF ne constitue sûrement pas un pari perdu d'avance.

1.7 Conclusion

Dans ce chapitre, il a été montré que les scientifiques modélisent des phénomènes physiques par des méthodes nécessitant de nombreux calculs. Seule l'utilisation des ordinateurs leur a permis de rendre ces calculs abordables.

Ce sont les scientifiques qui ont permis à l'informatique d'émerger et ce sont eux qui ont fait naître un besoin de puissance qui va toujours croissant. Ce besoin a mené naturellement vers le parallélisme.

Cependant, cette évolution a amené une grande diversité dans les architectures et dans les modèles de programmation. Cette diversité rend difficile l'utilisation de l'outil informatique par des non-spécialistes.

C'est pourquoi l'uniformisation de la programmation proposée par le langage FORTRAN a eu un tel succès. De même, HPF, son extension au parallélisme, rend la programmation parallèle accessible.

Malgré cette amélioration, la programmation n'est pas un travail facile. C'est pourquoi il est nécessaire de disposer d'outils facilitant cette tâche. Ces outils peuvent aider le programmeur dans toutes les étapes du développement d'applications.

Le chapitre suivant va s'attacher à définir ces différentes étapes, et cherchera à cerner les points sur lesquels il est essentiel de disposer d'une aide au développement.

Nous prendrons quelques outils existants en exemple et nous chercherons à en déduire la forme que devrait avoir un environnement général d'aide au développement d'applications parallèles.

Chapitre 2

Les environnements d'aide à la programmation

L'inconvénient avec les machines, ce sont les hommes ...

Un présentateur de CBS-TV lors d'une soirée électorale en 1952, alors que l'Univac I chargé des estimations venait de tomber en panne.

2.1 Introduction

Nous avons vu au chapitre précédent que l'informatique a évolué très vite ces cinquante dernières années, mais que ces progrès se sont déroulés de façon un peu anarchique. Ainsi, des architectures nouvelles sont apparues et les programmeurs ont dû les utiliser sans modèle précis pour les aider à s'y adapter.

De même que les langages de programmation sont apparus bien après les premiers ordinateurs, les méthodes de développement de logiciels ont été complètement absentes pendant longtemps.

Actuellement, dans le domaine de l'informatique industrielle (systèmes embarqués, automatique ...) et en informatique de gestion, le cycle de vie d'un logiciel est bien maîtrisé. Pourtant les énormes problèmes que pose le passage à l'an 2000 montrent bien que la maintenance de produits écrits il y a plusieurs années est loin d'être évidente.

Dès qu'il s'agit d'applications scientifiques, on retrouve le même problème. On rencontre des programmes écrits en FORTRAN 77 il y a vingt ans et dont plus personne ne sait ce

qu'ils contiennent. On continue à les utiliser tels qu'ils sont, sans oser y mettre les mains et en essayant de contourner leurs *bugs*, simplement parce que personne ne sait comment les corriger[1]. C'est pourquoi il est nécessaire de fixer des méthodes de développement en informatique scientifique. Or, pour pousser les programmeurs à utiliser ces méthodes, il faut leur fournir des outils permettant de les y aider.

En informatique scientifique, le codage d'un problème, une fois l'algorithme défini, se résume généralement à une boucle entre programmation, tests et déverminage, suivie d'une autre boucle entre modifications, tests et optimisations.

Nous ne nous intéresserons pas à la définition de l'algorithme. Le déroulement de cette étape est spécifique à chaque problème.

Pour les étapes suivantes, chacune d'elle confronte le programmeur à des problèmes spécifiques. Aussi, un certain nombre d'outils sont apparus afin de faciliter le développement, le déverminage, l'optimisation et la maintenance des programmes.

Dans le cas de la parallélisation d'un programme existant, ou même quand il s'agit d'écrire un programme parallèle en partant de zéro, on part d'un algorithme séquentiel dans lequel on cherche à déterminer ce qu'il est possible d'améliorer par des traitements parallèles. Pour cela, la méthode sera différente selon le modèle de programmation que l'on souhaite utiliser.

Pour une solution à parallélisme de tâches, Ian FOSTER décrit dans [Fos95] une méthodologie qui consiste à isoler des sous-traitements dans l'algorithme puis à les organiser de façon à répartir le travail entre les nœuds de calcul.

Dans le cas du parallélisme de données, il n'existe pas de méthode bien établie, mais on peut résumer le principe comme suit :

- isoler les ensembles réguliers de données,
- isoler les traitements qui sont effectués sur ces ensembles,
- définir les dépendances entre les données au niveau de ces traitements,
- déduire des transformations (notamment sur les boucles) qui exploitent la régularité des données afin d'obtenir des traitements parallèles (transformation des boucles en `forall`, opérations globales sur les tableaux ...),
- déduire un placement des données qui favorise la localité entre les éléments qui interagissent et qui équilibre la charge des nœuds de calcul.

L'ordre dans lequel les deux dernières parties de cette méthode doivent être effectuées (réorganisation des boucles et placement des données) reste encore un problème ouvert. Dans la pratique, ce sont les tests qui permettent de juger de l'efficacité du résultat et donc de boucler entre ces deux étapes afin d'arriver peu à peu à une solution satisfaisante.

Là encore, l'utilisation d'un certain nombre d'outils permet de faciliter le développement de telles applications, aussi bien dans le domaine du parallélisme de tâches qu'avec le parallélisme de données.

1. Je n'invente rien; c'est du vécu !

Parallèlement au calcul scientifique, d'autres utilisations de l'outil informatique sont apparues au fil des années.

Parmi ces nouvelles utilisations, on trouve la bureautique, qui est actuellement l'unique activité de la majeure partie des ordinateurs dans le monde. La bureautique a entraîné une telle démocratisation des ordinateurs qu'il est devenu nécessaire de les rendre accessibles à tous, avec un minimum d'apprentissage.

L'ergonomie et l'intuitivité sont donc devenus des éléments sans lesquels un logiciel grand public n'a que très peu de chance de survivre.

C'est entre autres dans le but de développer ces notions, qu'en 1970, *Xerox* crée le PARC (Palo Alto Research Center) [PAR]. Les chercheurs du PARC ont pour mission de créer *l'architecture de l'information*, mais n'ont aucun objectif commercial à atteindre.

C'est de ce centre que sort en 1973 le *Alto Dynabook*, un ordinateur dans lequel apparaissent les notions d'écran *bitmap*, d'interface graphique, de souris (la première souris commerciale) et de WYSIWYG[2].

Cette notion d'interface graphique se généralise très vite et les logiciels ont alors la possibilité de représenter des données sous une forme visuelle.

C'est aussi cette aide autre que textuelle que peuvent apporter les outils d'aide à la programmation. Nous allons chercher à la faire ressortir dans ce chapitre. Pour cela, nous allons présenter quelques-uns des outils existant, dans l'informatique en général, puis autour du calcul scientifique et du parallélisme. Nous terminerons en regardant plus précisément les outils dédiés à FORTRAN et HPF.

Le fil conducteur de ce chapitre sera l'étude, étape par étape, de la démarche qu'utilise un programmeur pour développer un programme de calcul.

De cette façon, pour chacune des étapes de cette analyse, nous pourrons déterminer les besoins du programmeur et en déduire les points pour lesquels des outils peuvent apporter une aide.

Exemple : *Les exemples*

Des outils existants seront présentés à titre d'exemple. Ces présentations seront encadrées comme ce paragraphe.

idée 0 *À chaque fois qu'un concept présenté sera considéré comme une idée à retenir ou à améliorer, il sera repéré de la même façon que ce paragraphe.*

2.2 La programmation séquentielle

La conception d'un programme commence par la définition d'un algorithme, à partir d'un cahier des charges.

Une fois cet algorithme défini, il faut le traduire dans un langage de programmation donné. Le choix du langage est un autre problème dans lequel les goûts et habitudes du programmeur

2. *What You See Is What You Get* : vous obtenez ce que vous voyez.

entrent souvent plus en jeu que les spécificités des langages. Nous ne nous étendrons pas plus sur ce sujet, qui est plutôt du ressort d'une subjectivité quasi-religieuse[3].

Cette étape de traduction est pour certains points assez pénible, dans le sens où il s'agit quasiment de traduction ligne à ligne.

Dans le domaine de l'informatique de gestion, il existe des ateliers de génie logiciel qui automatisent cette phase : à partir de l'analyse organique du projet (description des fichiers et des traitements), du code est généré automatiquement[4].

2.2.1 La programmation visuelle

Pour faire disparaître complètement cette phase, des projets ont essayé de mettre au point une façon d'exprimer directement un algorithme dans une forme non-textuelle, mais qui soit utilisable par un interpréteur ou un compilateur.

Il s'agit donc de représenter les algorithmes d'une façon purement graphique. On appelle ce concept la programmation visuelle.

L'utilisation de ce concept reste assez limitée, c'est pourquoi nous ne nous étendrons pas plus sur ce sujet. Le lecteur peut se reporter sur [Bur] qui présente une bibliographie très complète sur ce thème.

On peut faire une remarque sur ce type de programmation : la plupart de ces projets cherchent trop à atteindre le « tout visuel ». En général, il est quand même intéressant de garder un support textuel quelque part.

idée 1 *Il est difficile de dire si cela est dû au fait que les utilisateurs sont souvent déjà habitués à la programmation textuelle et sont donc un peu perdus, ou s'il y a vraiment un problème de compréhension dans le tout visuel, mais il semble nécessaire de garder une référence écrite à associer aux visualisations graphiques.*

Exemple *:* ARRAY-OL

ARRAY-OL (array oriented language) est un langage de programmation dédié au traitement du signal multi-dimensionnel. Son principe repose sur la manipulation de flux de tableaux. Ces tableaux sont envoyés dans un graphe dont chaque nœud constitue une transformation élémentaire.

Un niveau global permet de décrire graphiquement l'organisation des nœuds et des flux (voir figure 2.1). Ensuite, un niveau local permet de spécifier un motif sur les tableaux. Ce motif permet de décrire graphiquement un mode de parcours du tableau. Enfin, le corps de la boucle créée par ce motif est écrit en C++.

Ce langage constitue donc un compromis entre programmation visuelle et programmation textuelle : l'organisation générale des traitements est spécifiée à partir d'une interface gra-

3. Demandez à un intégriste du FORTRAN de programmer en ADA ou en JAVA et vous comprendrez ce que je veux dire ...

4. Comme il s'agit en général de COBOL, on comprend que les développeurs n'aient pas trop envie de faire ça à la main.

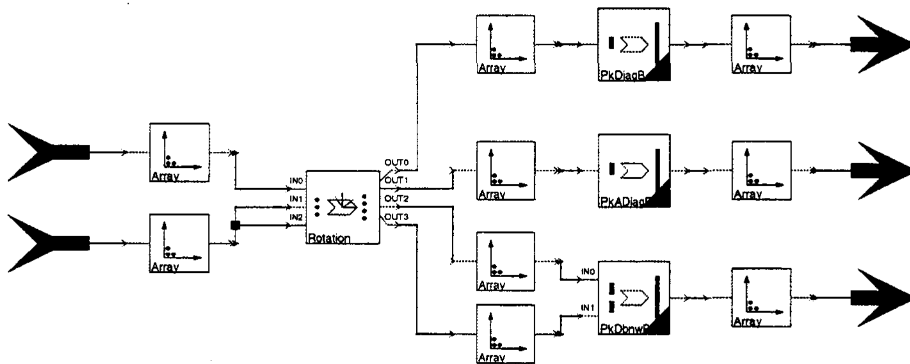


FIG. 2.1: *Description de flux de tableaux en* ARRAY-OL.

phique, mais le corps des « briques de base » à travers lesquelles passent les tableaux sont écrites en C++.

2.2.2 La réutilisation

Nous venons de voir qu'une fois qu'il a défini un algorithme, le développeur d'un programme de calcul scientifique commence par le programmer.

Comme il a été vu dans le chapitre précédent, le simple fait de disposer d'un compilateur est déjà d'une aide considérable, par rapport à la programmation en langage machine.

De plus, les langages de programmation ont beaucoup évolué et sont de plus en plus confortables à utiliser. La modularité puis la conception par objets par exemple permettent de constituer des bibliothèques d'outils réutilisables.

Ces concepts ont permis l'émergence de bibliothèques contenant des ensembles d'outils relatifs à un thème. Des bibliothèques de calcul comme BLAS ou LAPACK, par exemple, allègent les programmes de calcul d'algèbre linéaire et évitent de « réinventer la roue » à chaque inversion de matrice.

2.2.3 L'édition du source

En dehors des outils ou bibliothèques qui étendent les fonctionnalités d'un langage, ou simplifient son utilisation, il est également important d'aider à la lisibilité des programmes.

La façon dont est présenté le texte même du programme peut améliorer sa lisibilité. L'indentation est déjà un grand pas et des éditeurs de textes ont été spécialisés pour l'écriture de programmes. Ceux-ci permettent d'aider le programmeur en lui donnant accès en ligne à des documentations et en faisant ressortir les éléments syntaxiques dans des couleurs ou polices spécifiques.

Le célèbre EMACS est un des premiers éditeurs à avoir exploité cette notion en profondeur. Les modes spécifiques à chaque type de fichier permettent de présenter tout programme sous

une forme adaptée et l'indente automatiquement.

idée 2 *Ainsi, bien qu'un éditeur de texte ne fasse, par définition, que manipuler du texte, l'usage de couleurs permet d'aider l'utilisateur à se repérer plus facilement dans le code.*

2.2.4 La définition des interfaces graphiques

Si le programme doit être utilisé dans un environnement graphique comme X11 ou WINDOWS, il faut définir l'interface graphique du programme.

Pour cela, des kits de développement permettent d'accéder à l'interface de programmation de l'environnement fenêtré, par l'intermédiaire d'une bibliothèque. Ces kits nécessitent généralement un modèle de programmation événementielle.

La définition d'une interface graphique consiste à définir des fenêtres contenant des objets appelés *widgets* (boutons, menus, ascenseurs ...) et à mettre en place les gestionnaires d'événements. Dans ce travail, il est pénible de devoir entrer en profondeur dans les programmes à chaque changement d'un détail du *design*.

Des outils comme DEVGUIDE (associé à XVIEW sous X11) ou XF (associé à TCL/TK) permettent de se libérer complètement de cette partie de la programmation.

L'utilisateur définit de façon WYSIWYG son interface graphique, puis le code correspondant est automatiquement généré.

Exemple : INTERFACE BUILDER

INTERFACE BUILDER [GM] est le générateur d'interfaces graphiques fourni dans l'environnement de développement de NEXTSTEP. Il ne s'agit pas du premier outil de ce genre, mais c'est le premier à avoir eu un succès commercial.

L'utilisateur dessine ses fenêtres et ses *widgets* et leur associe interactivement des actions à effectuer quand un événement donné survient (voir figure 2.2 page suivante).

Ensuite, l'outil génère automatiquement le code correspondant et l'utilisateur n'a plus qu'à « boucher les trous » avec la partie traitement de son application, sans avoir à programmer toute la partie interface.

Changer un détail dans l'interface est alors possible interactivement, sans avoir à replonger dans les méandres des *callbacks*.

idée 3 *On constate donc qu'il est possible de libérer le programmeur de toute une partie de la transcription du concept en code, le laissant ainsi se concentrer sur le design de l'application, sans avoir à s'occuper de la façon dont il est mis en œuvre.*

De tels éditeurs proposent en général un moyen de simuler le programme généré : un mode de test permet d'essayer l'interface graphique et de tester les réactions aux manipulations des

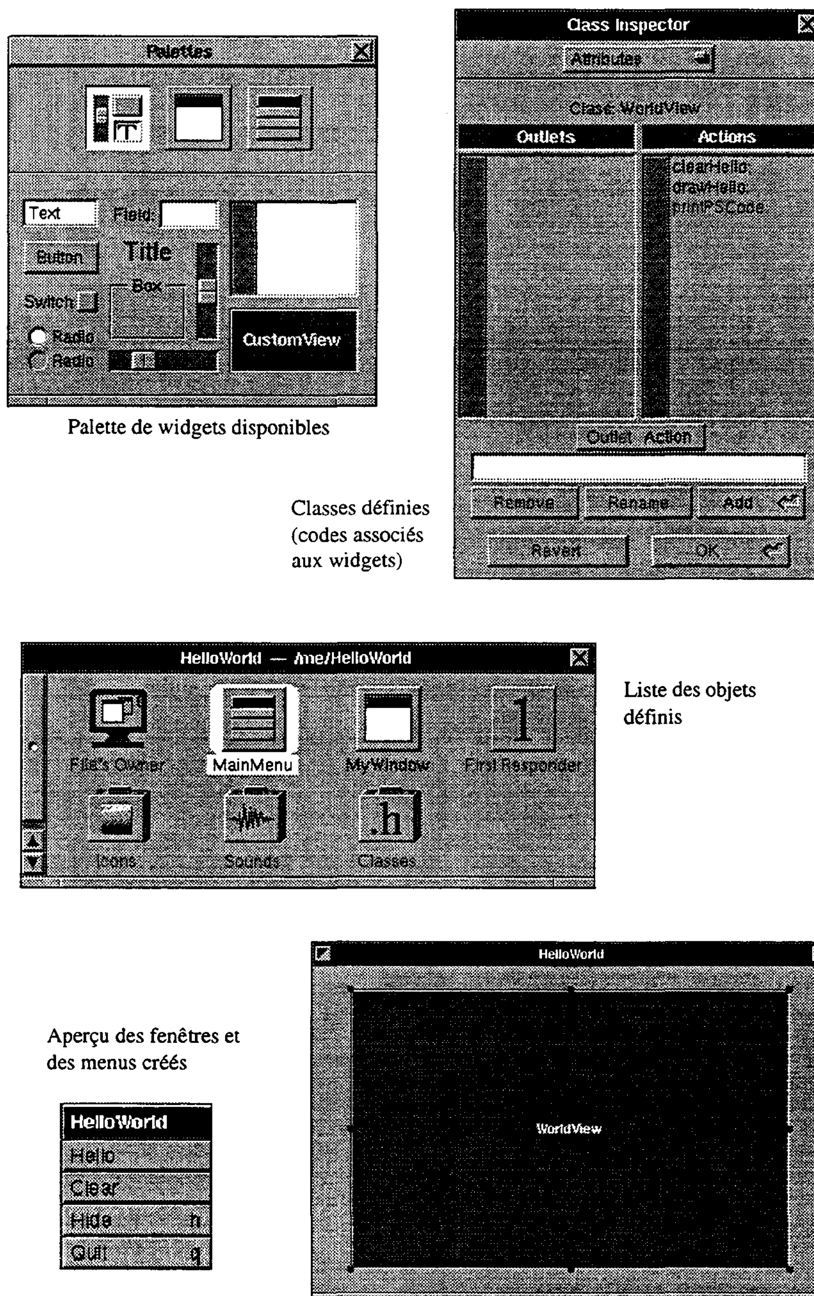


FIG. 2.2: *Construction d'application avec* INTERFACE BUILDER.

widgets, avant d'avoir écrit les codes de traitement associés.

idée 4 | *Cette approche, qui permet au programmeur de tester une partie de son programme alors qu'il n'est pas encore compilé est également un bon moyen de diminuer considérablement le temps de développement d'une application.*

Il est important de ne pas confondre les générateurs d'interfaces graphique avec la programmation visuelle. Un outil comme INTERFACE BUILDER permet de définir graphiquement l'interface d'un programme, c'est-à-dire quelque chose dont la conception est graphique en soi. Dans le cas d'un langage visuel, c'est l'algorithme lui-même qu'on cherche à exprimer de façon non textuelle.

2.2.5 L'intégration des outils annexes

Une fois que le programme a été écrit, le travail est loin d'être terminé. Il faut tout d'abord le compiler. À ce niveau, les messages d'erreurs de compilation renvoient à des numéro de lignes.

Si ces messages peuvent être interprétés par l'éditeur de texte, celui-ci a moyen de guider le programmeur dans ses corrections en le renvoyant automatiquement vers les lignes incriminées.

Une fois compilé, le programme doit être testé. Si des erreurs apparaissent dans son comportement, il faut en déterminer l'origine.

Pour cela, le programmeur doit pouvoir consulter les valeurs des données du programme à différentes étapes de l'exécution.

C'est dans ce but que des dévermineurs (ou *debugers*) ont été écrits. Ils permettent de placer des points d'arrêt dans le programme. Il devient alors possible de consulter les valeurs des variables et l'état de la pile d'appels de fonctions, pendant l'exécution.

Là encore, une intégration dans l'éditeur permet de suivre cette exécution directement dans le programme.

Il apparaît donc que l'intégration d'outils de compilation et de déverminage, dans l'éditeur de programmes, permet de faciliter les aller-retour nécessaires entre ces outils et l'éditeur.

De la même façon que les suites bureautiques intègrent tableur, traitement de textes et moteur de bases de données, des environnements de programmation intégrant éditeur de source, compilateur, dévermineur et analyseur de performances sont arrivés sur le marché.

Exemple : RHIDE

L'environnement de programmation RHIDE [Höh] (directement et ouvertement inspiré de celui de BORLAND) se présente comme un éditeur de texte spécialisé dans l'édition de programmes (voir figure 2.3 page ci-contre). Il est possible de le configurer pour différents langages de programmation. Ainsi, les mots clés, les noms de fonctions et d'autres éléments syntaxiques importants sont mis en valeur par des couleurs différentes.

De plus, à partir de l'éditeur, il est possible d'accéder à de la documentation en ligne, de lancer une compilation et d'exécuter le programme édité à travers un dévermineur.

Barre de menus
Vue du source
Pile d'appel de fonctions
Evaluation d'une expression pendant l'exécution
Traçage de variables
Fichiers constituant le projet courant

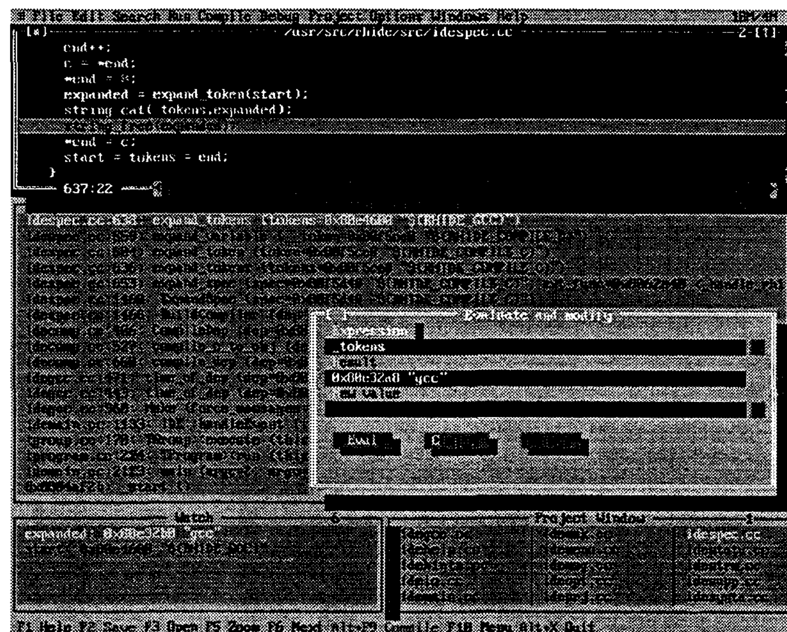


FIG. 2.3: *L'environnement* RHIDE.

L'intégration permet de renvoyer directement dans l'éditeur. Ainsi, lors de la compilation, il suffit de sélectionner un message d'erreur pour être renvoyé automatiquement sur la ligne de code erronée. De même, l'exécution pas à pas déplace un curseur dans la fenêtre d'édition.

idée 5

Un tel environnement intégré n'apporte pas grand-chose au niveau de chacun de ses composants, mais il permet de travailler avec plusieurs outils dont l'utilisation est alors relativement uniformisée.
L'utilisateur n'a plus à exécuter des programmes différents, ayant chacun leur interface propre. Il a tout « à portée de souris ».

Une fois le déverminage terminé, l'optimisation des performances du programme passe par l'analyse du temps passé dans les différentes parties du code.

Là encore l'intégration d'outils d'analyse de performances dans l'environnement de programmation facilite les va-et-vient entre les statistiques générées et le code.

2.2.6 Plus loin dans l'intégration

Au début de ce chapitre, les ateliers de génie logiciel ont été cités comme moyen de spécifier toute l'analyse d'un projet et de générer automatiquement le code de l'application.

Dans le cadre de l'informatique scientifique, il est également possible d'arriver à un tel niveau d'intégration des outils.

Un programme de calcul peut être spécifié par :

- un ensemble de données d'entrées,

- des procédures de saisie de ces données,
- une suite de traitements à effectuer sur ces données, en utilisant un ensemble de données intermédiaires,
- un ensemble de données de sortie,
- des procédures de visualisation de ces données.

Les différents ensembles de données peuvent être decrits graphiquement. Les traitements sont soit des algorithmes standard déjà définis dans des bibliothèques, soit des algorithmes spécifiques, pour lesquels des modules peuvent être développés.

Les interactions entre données et modules peuvent être organisées graphiquement par un organigramme, ou un graphe de type *dataflow*.

Le principe général d'un « atelier de génie logiciel dédié au calcul scientifique » semble donc assez simple [5].

Exemple : AVS/EXPRESS

AVS/EXPRESS [6] [AVS] constitue un exemple d'un tel environnement.

Ce logiciel permet de construire graphiquement des applications de calcul scientifique à partir de briques de bases, sur un modèle proche du *dataflow* (voir figure 2.4 page suivante).

Chaque module de sa bibliothèque prend des flux de données en entrée, effectue des traitements dessus et sort d'autres flux. De plus, certains modules sont des interfaces de visualisation d'ensembles de données. Enfin, les flux d'entrées scalaires peuvent être associés à des *widgets* que le programmeur peut placer dans une fenêtre à la manière d'un éditeur d'interfaces graphiques.

Bien que sa bibliothèque de traitement et de visualisation de données et d'images soit très complète, l'utilisateur a aussi la possibilité d'écrire ses propres modules en C ou en FORTRAN.

Il s'agit donc d'un outil qui regroupe à la fois les notions de langage visuel et de constructeur d'interfaces graphiques.

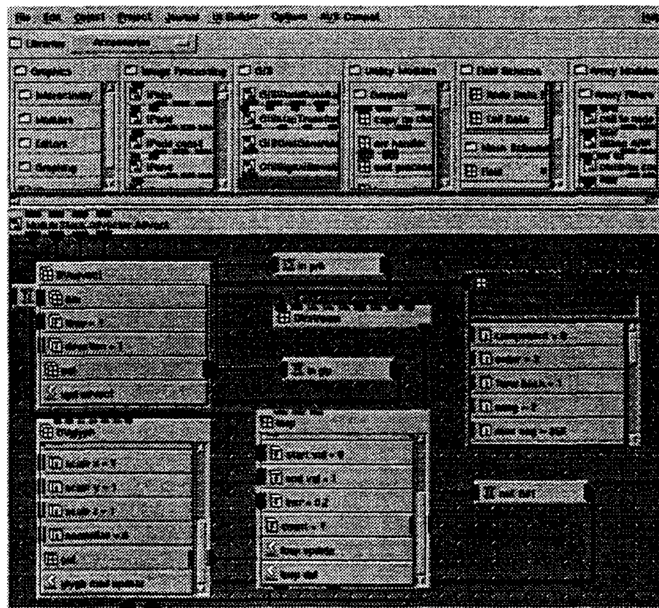
idée 6 AVS/EXPRESS *constitue donc un environnement de programmation complet : de l'algorithme général à l'application graphique finale, tout peut être développé à partir du même outil.*

2.3 La programmation à parallélisme de tâches

Le dernier exemple cité dans la section précédente nous montre un environnement de programmation complet très puissant. Cependant, celui-ci n'est pas disponible pour des architectures parallèles.

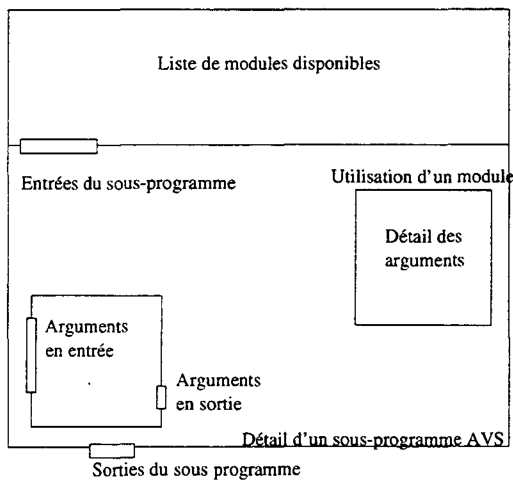
5. Je ne dis pas qu'il est facile à réaliser ! Je dis juste que ça ne doit pas être pire que dans le cas de l'informatique de gestion.

6. Merci à Charlene Glatkowski, d'AVS Inc., pour m'avoir fourni gracieusement les copies d'écrans.



L'interface de programmation

Exemple d'application réalisée avec AVS



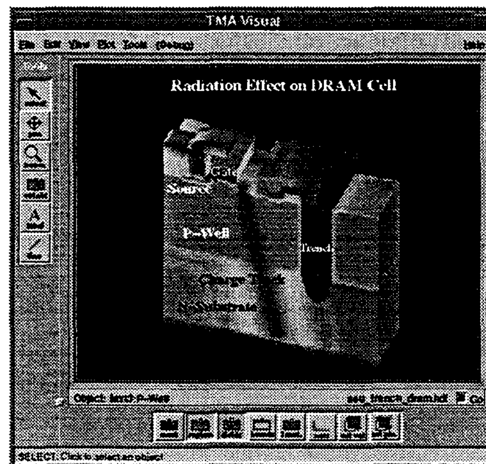


FIG. 2.4: *Application créée avec* AVS/EXPRESS *et exemple d'utilisation de l'interface de programmation.*

La programmation séquentielle existe depuis beaucoup plus longtemps que la programmation parallèle et les machines parallèles sont bien plus rares que les machines séquentielles. L'existence de langages de programmation « grand public » comme BASIC ou PASCAL a permis l'émergence de nombreux outils de développement.

Au contraire, le parallélisme est longtemps resté un domaine de spécialistes. Les développeurs d'applications parallèles sont souvent des personnes spécialisées dans ce domaine. La demande en outils de vulgarisation est donc moins forte.

D'un autre côté, le déverminage d'applications parallèles est très ardu, surtout dans le cas du parallélisme de tâches, où les problèmes d'ordonnancement et de synchronisation sont difficiles à résoudre.

De plus, quel que soit le modèle de programmation utilisé, il n'est pas facile de suivre la valeur des données manipulées quand elles sont « éparpillées » parmi plusieurs nœuds de calcul.

C'est pourquoi c'est surtout dans la partie post-compilation qu'on trouve des outils d'aide au développement d'applications parallèles.

2.3.1 Le découpage des tâches

Dans le cas du parallélisme de tâches, les premières étapes du développement consistent à organiser le code en tâches communicantes. Cette partie constitue un travail de conception pour lequel il est difficile de construire des outils.

Une fois le découpage défini, il est possible d'organiser de façon graphique les différentes tâches et les liens de communication à établir entre eux. Tout comme **AVS/EXPRESS** propose de relier des modules par des liens symbolisant les flux de données, il est possible de « dessiner » les liens de communication entre les nœuds d'un graphe représentant les différentes tâches.

Exemple : **TRAPPER**

TRAPPER [Tra] est un environnement de programmation parallèle utilisant ce principe. Le logiciel est découpé en quatre outils distincts :

- Design Tool permet de définir une application parallèle comme un ensemble de modules échangeant des informations à travers des ports de communication. Les ports de communication sont implémentés par des appels à PVM.

  L'utilisateur définit graphiquement un ensemble de nœuds. Chaque nœud contient un certain nombre de ports de communication, que l'utilisateur peut relier entre eux.

- Configuration Tool génère les fichiers de configuration nécessaires à PVM à partir d'une interface graphique de description de l'architecture de la machine cible et d'un fichier de description du placement généré par l'outil précédent.

  On obtient donc automatiquement un fichier de configuration à partir d'une représentation visuelle de son contenu, de la même manière qu'un générateur d'interfaces graphiques génère un code à partir du dessin des fenêtres.

- Visualization Tool aide au déverminage en proposant des graphiques représentant les activités, les échanges de messages ou les valeurs des variables.

- Performance Tool utilise des fichiers de données collectées par des processus de traçage pour afficher des graphes des temps d'exécution des différentes tâches et isoler les chemins critiques.

idée 7 — *Le Design Tool propose donc une aide directe lors du développement lui-même et pas seulement une source d'informations sur un programme existant. Cette étape est donc très proche de la programmation visuelle, même si chaque module doit être écrit en langage de programmation classique.*

2.3.2 Le déverminage

Une fois le programme écrit et compilé, la phase de déverminage est plus délicate que dans le cas séquentiel.

En effet, on exécute non plus un programme, mais un ensemble de programmes communiquant entre eux.

Les erreurs de programmation ou de conception peuvent alors apparaître à deux niveaux : un des composants peut contenir une erreur, ou les passages de messages peuvent être mal organisés.

Dans le premier cas, le traçage du composant soupçonné est possible. Par contre le traçage des messages est assez complexe et il n'est plus possible d'utiliser un dévermineur classique.

Il est possible de générer une trace de tous les envois/réceptions de messages et de les analyser après coup, mais de tels fichiers de traces sont très difficilement lisibles.

C'est pourquoi des outils de visualisation des échanges de messages sont nécessaires.

Exemple : XPVM

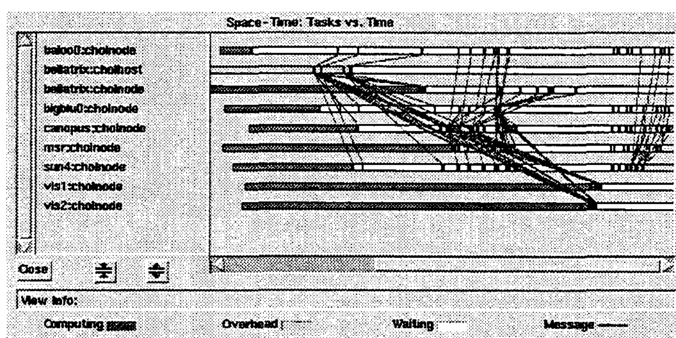


FIG. 2.5: *Visualisation de passages de messages avec* XPVM.

Dans le cas de la programmation avec la librairie de passages de messages PVM, XPVM constitue un tel outil [XPV]. Il s'agit d'une interface graphique à la console PVM, à partir de laquelle il est possible de contrôler interactivement la mise en place de l'architecture virtuelle, l'exécution des différentes tâches et les échanges de messages.

L'activité de chaque nœud de calcul est visualisée par une barre dont les différentes couleurs représentent les états (actif, inactif, bloqué ...). Entre ces barres, les liens représentent les passages de messages (voir figure 2.5 page précédente).

idée 8 — XPVM *permet donc de visualiser des activités et des passages de message de façon visuelle et évite donc à l'utilisateur de se perdre dans les fichiers de traces souvent incompréhensibles pour des non spécialistes.*

Exemple *:* VAMPIR *et* DINEMAS

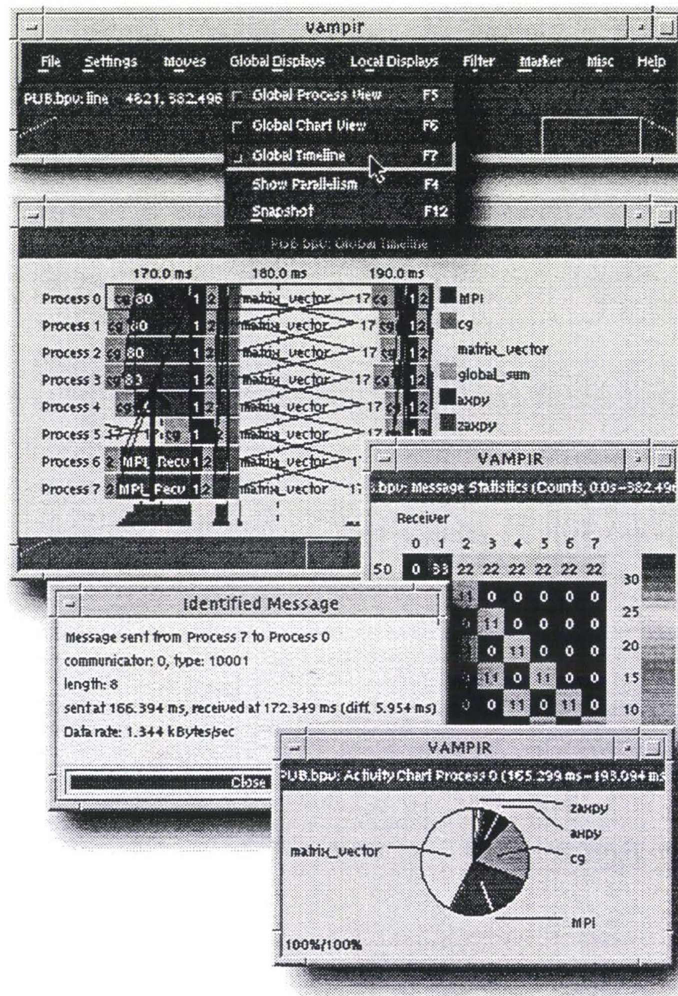


FIG. 2.6: *Visualisation d'activités et de passages de messages avec* VAMPIR.

VAMPIR [Vam] est un autre outil de visualisation du déroulement d'un programme à passage de messages, mais sous MPI. Il permet de représenter graphiquement des fichiers d'analyse post-mortem (voir figure 2.6).

Il fournit des filtres et des outils statistiques qui permettent d'isoler les informations pertinentes selon les critères demandés par l'utilisateur.

DINEMAS [Din] vient compléter l'utilisation de VAMPIR. Il effectue lui aussi un traçage des communications, mais il permet surtout d'en déduire le comportement du programme tracé relativement à une autre machine. L'utilisateur fourni un ensemble de spécifications d'une machine donnée et DINEMAS transcrit les fichiers de traces pour ce contexte. La transcription est alors utilisable par VAMPIR.

Ainsi, à partir d'une exécution sur une machine quelconque, il est possible d'obtenir une prévision des performances du programme en cas d'exécution sur une autre machine parallèle. De plus, ces prévisions sont visualisables graphiquement par VAMPIR.

idée 9 *Le couple* VAMPIR/DIMENAS *fournit un moyen d'évaluer les performances d'un programme sans être obligé de l'exécuter sur la machine parallèle cible, dont le coût d'utilisation est parfois prohibitif.*

En conclusion, le déverminage et l'optimisation de programmes écrits selon le modèle à parallélisme de tâches peuvent être facilités par des outils graphiques aussi conviviaux que ceux disponibles en programmation séquentielle.

2.4 L'utilisation de Fortran et HPF

Dans le cadre du calcul scientifique, on a vu que la plupart du développement consiste à paralléliser des codes FORTRAN existants.

Il est bien sûr possible de passer à un code à parallélisme de tâches, en utilisant des bibliothèques comme PVM ou MPI. On doit alors reprendre toute l'organisation du programme, comme on l'a vu dans la section précédente.

Une autre solution consiste à utiliser HPF. HPF utilise le parallélisme de données comme concept de programmation. Le développement et l'optimisation des programmes ne nécessitent donc plus les mêmes besoins que pour les outils dédiés au parallélisme de tâches.

Certains compilateurs permettent de générer du code à passages de messages. Les outils vus précédemment sont alors utilisables pour observer en détail le code généré. Cependant, comme l'utilisateur n'a pas moyen d'influer directement sur la façon dont ce code est généré, ces observations restent d'une utilité limitée.

Dans le chapitre précédent, on a vu que le parallélisme de données permet de paralléliser petit à petit un code, sans revoir son organisation en profondeur.

En effet, un portage vers HPF peut se faire de façon incrémentale. Chaque insertion de directive apporte un peu plus de parallélisme, sans qu'il n'y ait aucune intervention sur l'algorithme lui-même.

De même, il est possible de repérer des parties de code dont une réorganisation locale permet de faire ressortir des traitements réguliers qu'un compilateur saura optimiser pour une exécution vectorielle ou SPMD.

2.4.1 Le passage de Fortran 77 à Fortran 95

Comme il a été vu dans la section 1.6.6 page 42, le portage de FORTRAN vers HPF nécessite une remise au propre du code. Une part importante du travail consiste à supprimer les constructions obscures qui empêchent de faire ressortir les éléments parallélisables, jusqu'à l'obtention d'un programme en FORTRAN 95 « propre ».

Exemple : FORESYS

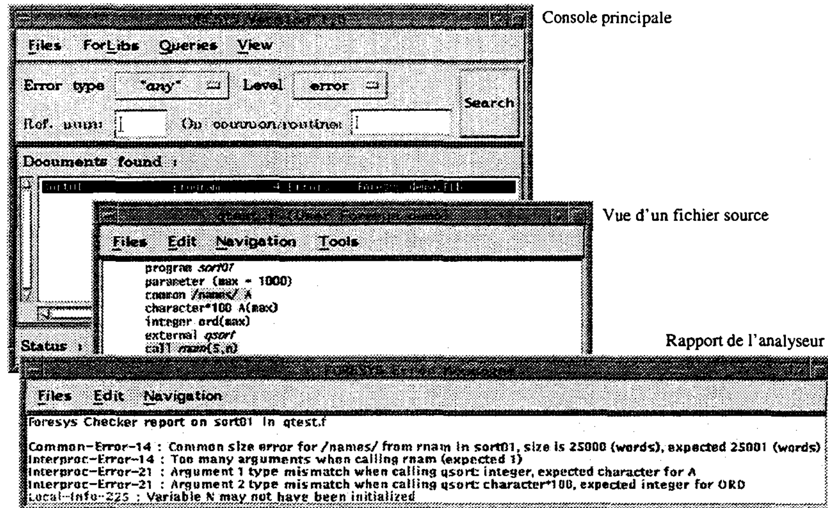


FIG. 2.7: *Navigation dans un programme avec* FORESYS.

FORESYS [For] est un exemple d'outil qui aide à la transcription de tels programmes et qui devient de plus en plus reconnu dans la communauté scientifique et industrielle.

FORESYS et composé de plusieurs modules, dont :

- Un analyseur de FORTRAN 77 qui permet de vérifier la cohérence d'un programme FORTRAN dans son ensemble. Il permet de valider la qualité numérique et de suivre les appels de procédures et les utilisations de constructions `common` afin de repérer les redimensionnements implicites de données et les inconsistances de types dans les arguments.

  Les informations recueillies par l'analyseur sont réunies dans un ensemble de fichiers spécifiques, afin d'être réutilisables par les autres modules de FORESYS.

- Une interface spécialisée pour FORTRAN, qui permet de naviguer à travers un programme. À partir de cette interface, il est possible d'accéder aux informations réunies par l'analyseur et de corriger interactivement les erreurs qu'il a détectées (voir figure 2.7).

- Un outil de restructuration qui permet de transformer les constructions obsolètes du programme en une forme FORTRAN 95.

- Un outil de parallélisation qui permet de vectoriser des boucles et de guider l'utilisateur dans des transformations utiles au passage vers HPF.

FORESYS est donc un outil idéal pour passer d'un code FORTRAN 77 à un code FORTRAN 95 « propre » dans lequel il est ensuite possible d'insérer des directives HPF.

idée 10 *FORESYS propose donc un moyen de modifier un code existant de façon interactive. Les modifications sont effectuées par l'utilisateur, mais celui-ci est conseillé dans ses choix par des outils d'analyse et de parallélisation automatique.*

2.4.2 La parallélisation automatique

Une fois « remis au propre », le repérage des constructions parallélisables peut être fait automatiquement dans les cas peu complexes.

Exemple : PAF, CARAVAN

Ainsi, PAF (Paralléliseur Automatique de FORTRAN), développé au PRiSM [PRi], essaie d'automatiser la détection de code optimisable par des traitements parallèles.

Il intervient principalement dans la restructuration des boucles. Une analyse des dépendances entre les indices de boucles et les expressions dans les corps de boucles permet de déduire un ordonnancement optimal.

Cet ordonnancement est alors mis en application dans la génération d'un code CM-FORTRAN.

Le projet CARAVAN, le successeur de PAF, reprend le même principe, mais en y insérant des notions d'analyse floue de données qui permettent de traiter des cas plus complexes. Cette fois, le langage cible visé est HPF.

idée 11 *PAF travaille comme un compilateur : on lui fournit un programme et il travaille dessus pour en générer un autre, sans aucune intervention de l'utilisateur.*
Dans les cas qu'il sait traiter, c'est bien sûr idéal, mais ce manque d'interactivité empêche toute entraide entre l'utilisateur et le paralléliseur.
Si l'outil possédait un moyen d'expliquer pourquoi il ne peut pas traiter un cas donné, l'utilisateur pourrait apporter les modifications nécessaires pour supprimer les causes de blocage et l'outil pourrait reprendre la parallélisation.

2.4.3 Le placement de données

L'efficacité d'un programme HPF dépend fortement de l'ordonnancement des boucles et du placement des données. L'ordonnancement des boucles peut être optimisé à l'aide des outils vus précédemment. Nous allons donc nous concentrer maintenant sur le problème du placement.

Le placement de données consiste à décider ce que contiendra la mémoire locale de chaque processeur. Dans le cas d'HPF, on a vu que ce placement se fait sur les variables de type tableau et qu'il est décomposé en deux étapes : l'alignement et la distribution. Globalement, l'alignement cherche à optimiser la localité des données et la distribution cherche à équilibrer les charges de calcul.

Pour l'alignement, il faut donc repérer les interactions entre données et en déduire les parties qui doivent être réunies. Pour cela, il faut donc observer les accès aux tableaux.

Ces opérations ont clairement un aspect géométrique. Un tableau est un pavé dans un espace multidimensionnel et une opération régulière entre tableaux peut être représentée par une projection. Il semble donc possible de représenter graphiquement ces objets.

De même, l'alignement et la distribution représentent des projections d'un objet vers un autre. Là encore, une interprétation graphique est donc immédiate.

Spécifier des placements de la même manière qu'on dessine des interfaces graphiques ou des liens de communication entre tâches semble donc réalisable.

Exemple : `GDDT`

`GDDT` [GDD] (Graphical Data Distribution Tool) est un outil permettant de travailler sur des programmes écrits en VIENNA FORTRAN.

Il intervient au niveau de la définition des données et de leur distribution. Il permet non seulement de visualiser leur comportement, mais apporte aussi un moyen de les spécifier et de les modifier graphiquement.

Pour cela, les différents composants de `GDDT` interviennent au niveau du programme source pour la définition et la visualisation des distributions et aussi au niveau de l'exécution pour la visualisation des données.

Ainsi, en lançant l'exécution d'un code à travers un dévermineur, il est possible de suivre graphiquement l'évolution des données lors des redistributions. L'effet des distributions avec recouvrement peut également être affiché, ainsi que les mouvements de données lors d'une opération de calcul.

De plus, ces visualisations sont possibles au niveau des processeurs abstraits ou au niveau des processeurs physiques sur lesquels ils sont placés.

Au niveau de la définition des placements, `GDDT` propose un éditeur graphique permettant de définir la distribution d'un tableau donné. La directive de distribution correspondante est alors automatiquement générée.

`GDDT` met donc en place l'idée de définition visuelle des placements. Cependant, il est dédié à VIENNA FORTRAN et son compilateur. C'est grâce à cela qu'il peut intervenir aussi bien au niveau du source que du code généré.

Il semble qu'actuellement, `GDDT` n'ait pas d'équivalent fonctionnel dédié à HPF [7].

7. Mis à part HPF-BUILDER, bien sûr. Mais gardons un peu de suspense pour la suite ...

idée 12

On trouve dans cet outil un moyen d'éditer des directives de placement et de les visualiser aussi bien à partir du code source qu'à partir des informations recueillies à l'exécution par le dévermineur.
Ainsi, il est possible d'avoir une vue générale des placements ainsi qu'une vue détaillée de la charge effective en calcul et en communication.

2.4.4 La visualisation de données

Une fois les placements définis et le programme compilé, une étape de déverminage est généralement nécessaire.

Dans un programme utilisant le parallélisme de données, on garde un flot d'instruction unique. Il est donc possible de suivre son exécution de la même façon qu'avec un programme séquentiel.

Par contre, les données du programme sont réparties sur les différentes mémoires des processeurs et d'une façon dépendante du placement.

Consulter la valeur des données pendant l'exécution est donc plus délicat.

Exemple : DAQV

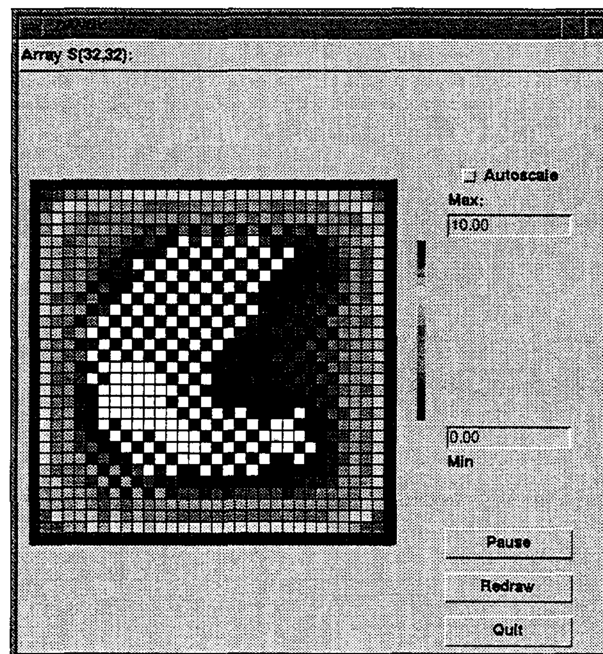


FIG. 2.8: *Visualisation de données avec* DAQV.

Le but de DAQV est de permettre à l'utilisateur de visualiser la valeur des données d'un programme HPF, de façon transparente, c'est-à-dire sans tenir compte de leur distribution.

Pour cela, DAQV contient une bibliothèque de fonctions que l'utilisateur peut insérer dans son code pour envoyer des données vers une interface graphique de visualisation (voir figure 2.8).

Les données sont alors affichées en utilisant des jeux de couleurs.

L'utilisateur appelle donc des fonctions demandant l'affichage d'une variable et cet appel provoque l'envoi des données locales à chaque processeur. L'interface graphique s'occupe de façon transparente de la remise en ordre de ces données.

idée 13

Cet outil permet donc de visualiser des données, quelle que soit leur distribution.
Cependant, c'est à l'utilisateur de « tout faire » : il doit placer des appels à des fonctions initialisant des données internes à DAQV, *pour chaque variable qu'il veut visualiser. Ensuite, chaque visualisation est décidée par un autre appel explicite.*
Il est donc nécessaire de modifier le programme à la main, de le recompiler et de l'exécuter à nouveau dès qu'on veut afficher une autre donnée, ou la même à un autre point de l'exécution.

Exemple *:* ANNAI/DDV

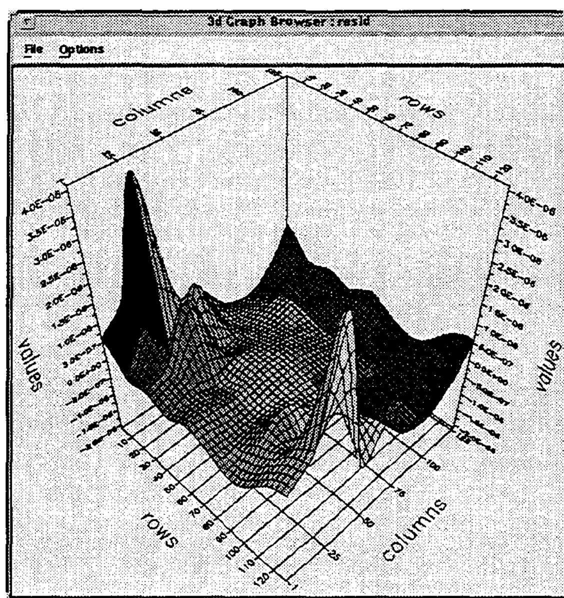


FIG. 2.9: *Visualisation de données avec* ANNAI/DDV.

ANNAI [Ann] est un ensemble d'outils d'aide au développement d'applications en HPF et/ou MPI.

Parmi ses fonctionnalités, cet ensemble permet l'instrumentation de codes HPF. Ainsi, l'insertion automatique d'appels à des fonctions de traçage permet ensuite de suivre l'exécution du programme avec les divers composants inclus dans ANNAI.

Contrairement à DAQV, cette instrumentation est faite automatiquement : des appels sont générés et insérés dans le programme source, avant la compilation. Ainsi, pendant l'exécution, toutes les informations nécessaires sont envoyées automatiquement à un processus intermédiaire que les autres outils vont consulter.

Un des composants, DDV (Distributed Data Visualizer), utilise cette instrumentation pour visualiser les tableaux du programme pendant l'exécution.

Cette visualisation est faite à trois niveaux :

- Le niveau le plus général visualise la distribution : un graphique représente par des couleurs la position des différentes parties d'un tableau le long des processeurs.
- Le niveau suivant affiche la valeur numérique de chaque élément du tableau.
- Le dernier niveau affiche ces valeurs sous la forme d'une courbe 3D (voir figure 2.9).

idée 14

ANNAI *permet de visualiser la valeur des données du programme indépendamment de leur distribution.*
La valeur des variables est affichée numériquement ou graphiquement. Par contre la visualisation de distribution utilise une méthode de jeux de couleurs (une couleur par processeur cible) qui nous semble limitée dès que la grille de processeurs cibles est de taille importante.
De plus, visualiser une distribution ne nécessite pas forcément l'exécution du code. Il est donc dommage de devoir instrumenter et exécuter un programme quand on cherche seulement à visualiser les placements.

2.4.5 Évaluation des performances

Une fois au point, le programme HPF obtenu peut sûrement être optimisé. Quelques modifications des placements permettent souvent d'améliorer la localité sur certaines sections critiques de l'algorithme.

Pour cela, il est nécessaire de déterminer les coûts induits par les communications générées et par l'équilibrage des charges de calcul.

Souvent, les compilateurs génèrent un code à passages de messages et il est parfois possible de le tracer (en utilisant un des outils vus à propos du parallélisme de tâches, par exemple). Cependant, HPF utilise un modèle de haut niveau et le compilateur apporte de nombreuses optimisations. De ce fait, il est difficile de comprendre à quelle ligne du programme initial correspond un message donné.

Il est donc nécessaire de connaître précisément le comportement du compilateur pour pouvoir remonter les traces de messages au niveau d'HPF.

Exemple *:* SVPABLO

SVPABLO est un autre outil d'instrumentation de code dont le but est cette fois d'évaluer les performances d'un programme HPF. L'instrumentation génère des statistiques sur les performances du code au niveau le plus bas, c'est-à-dire sur le code généré par le compilateur.

Le point fort de SVPABLO consiste à analyser après exécution ces statistiques, afin de remonter les informations au niveau d'HPF. Il est alors possible de déduire quelle partie du

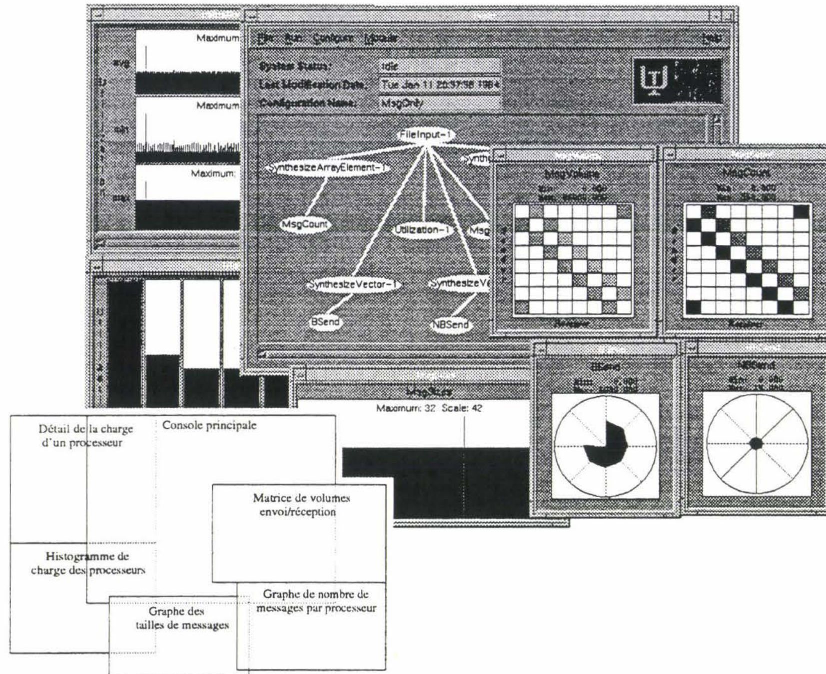


FIG. 2.10: *Statistiques sur les messages avec* PABLO.

code HPF est responsable de quel résultat sur le code généré. Une interface graphique visualise ces statistiques, en les rapprochant de la partie de code dont ils proviennent (voir figure 2.10).

SvPABLO fait partie d'un projet plus vaste, PABLO, dont le but est de fournir des outils d'analyse de performance, indépendamment du langage, du compilateur et de la machine. Ce projet va jusqu'à l'utilisation de la réalité virtuelle pour effectuer la visualisation. SvPABLO est une mise en œuvre de PABLO sur HPF, avec le compilateur de PGI.

idée 15 SvPABLO *permet donc d'observer finement les performances obtenues et d'en déterminer les causes au niveau HPF, sans être dépendant du modèle d'exécution, du compilateur ou de la machine cible.*

2.4.6 Intégration d'outils

Tout comme on rencontre des environnements de programmation complets en programmation séquentielle, des projets d'intégration d'outils intervenant à toutes les étapes du développement de programmes parallèles commencent à apparaître.

On a vu que le portage d'un programme FORTRAN 77 vers HPF est une tâche ardue mais souvent nécessaire pour les scientifiques. Cette chaîne de portage peut se résumer ainsi :

- mise au propre du code (suppression des `common`, suppression des déclarations implicites ...),

– adaptation au contraintes de parallélisation (suppression des *storage associations*, suppressions des goto ...),

– réordonnancement des boucles,

– placement des données,

– tests et optimisations des performances par affinement progressif dans les deux étapes précédentes.

A priori, le regroupement d'un compilateur et d'outils dédiés à chacune de ces étapes autour d'un éditeur commun semble donc être l'environnement idéal.

Exemple : HPFIT

HPFIT (HPF Integrated Tools) [HPFd] est un premier pas vers ce but. Il est basé sur un noyau, nommé TRANSTOOL, qui contient lui-même un éditeur (XEMACS) un analyseur syntaxique (celui d'ADAPTOR) et un analyseur de dépendances (PETIT).

HPFIT doit également intégrer HPFIZE, un outil d'insertion de directives HPF, ainsi que des outils de conversion des appels à des bibliothèques séquentielles (comme BLAS et LAPACK) en appels à leurs équivalents parallèles (PBLAS et SCALAPACK).

Enfin, pour qu'il puisse être utilisable d'un bout à l'autre de la chaîne de développement, il doit également être interfacé avec des outils de déverminage et d'analyse de performances.

Exemple : FITS

Le projet FITS (FORTRAN Integrated Tool Set) est une boîte à outils de développement regroupant FORESYS, VAMPIR, ANALYST et IDA.

Les deux premiers outils ont déjà été présentés dans les sections précédentes.

ANALYST effectue l'analyse d'un programme afin d'en tirer des informations comme le graphe d'appels de procédures et un organigramme résumé de chaque procédure (voir figure 2.11 page suivante).

Des informations sur les dépendances de données sont également disponibles et la prochaine version devrait intégrer des analyses sur les accès aux tableaux afin d'aider au placement de directives HPF.

Enfin, IDA (Interprocedural Dependency Analyser) permet l'analyse interprocédurale détaillée. Il génère les graphes d'appels complétés par les associations entre variables et arguments ainsi que les traces d'utilisation des variables et des common.

On peut remarquer que cet ensemble couvre donc toutes les étapes énumérées au début de cette section.

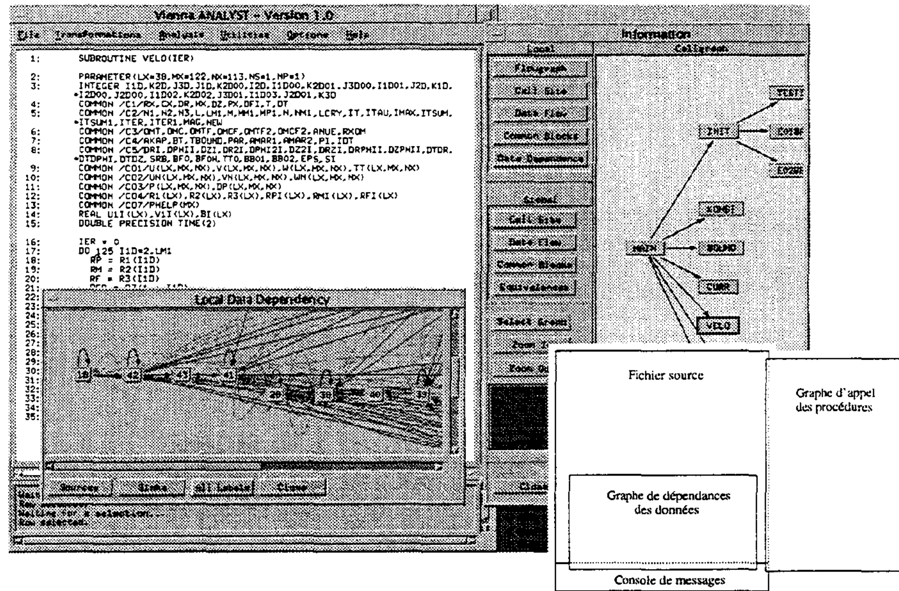


FIG. 2.11: *Graphes de dépendances de données et d'appels avec* **ANALYST**.

2.5 Conclusion : Vers un environnement graphique d'aide au développement en HPF

Tout au long de ce chapitre, nous avons montré la démarche suivie pour le développement d'une application. Pour chaque étape, nous avons passé en revue les divers problèmes auxquels est confronté l'auteur d'un programme de calcul scientifique, aussi bien en séquentiel qu'en parallèle.

Nous avons vu qu'il existe des outils pouvant aider le programmeur pour chacun de ces points.

En conclusion de ce chapitre, nous allons essayer de résumer ce qu'il en est dans le cas de la mise en œuvre d'un algorithme, sur une machine parallèle, en HPF.

On a vu que dans ce contexte, le développement peut se résumer à :

- définition d'un algorithme et traduction en FORTRAN 95, ou mise au propre d'un code FORTRAN 77 existant,
- parallélisation des boucles,
- placement des données,
- tests et optimisations.

Les spécificités des différents outils exposés pour chacun de ces points peuvent être réunies dans une suite logicielle commune (idées 5 et 6) dont le « cahier des charges » est alors le suivant :

Exemple : *L'outil idéal ?*

- Dès la première phase de la programmation, il faut disposer d'un éditeur de texte.

  Celui-ci doit mettre en place une *interface de navigation* à travers les différentes parties du programme. Il faut notamment pouvoir appréhender visuellement (à l'aide de couleurs et d'indentation automatique) la structure du programme et retrouver rapidement la déclaration d'un objet donné (idée 2).

- La restructuration des boucles et des constructions parallèles doit être en partie automatisée (idée 11) et, au moins, guidée par des outils d'analyse du code (idée 10).

  La remise en forme d'une boucle peut être faite sans manipulation directe du texte : une opération de type *drag and drop* peut suffire à échanger des bornes de boucles, tout en diminuant le risque d'erreurs dans les positions des bornes.

- Le placement des données doit être fait visuellement (idées 3 et 7). La représentation graphique des placements, mais aussi des accès aux tableaux aide dans le choix des placements. Les directives HPF doivent être générées à partir d'une édition visuelle (idée 12).

  On a vu qu'il est intéressant de garder une référence textuelle (idée 1), mais qu'il est possible d'effectuer des opérations d'édition et d'insertion de façon visuelle. C'est ensuite à l'outil utilisé de générer le texte correspondant. Un tel outil doit également savoir interpréter le contenu d'un programme afin d'être capable d'effectuer des modifications sur un code existant, à partir de la même interface que celle utilisée pour la création.

- Avant l'exécution et même avant la compilation, certaines vérifications peuvent être effectuées (idée 4). Les accès aux tableaux, selon les placements choisis, entraînent des charges de calculs et des communications dont le volume peut être estimé *a priori*.

  Pour cela, il faut un outil de pré-mesure, à partir d'une analyse du code source, des performances obtenues sur une ligne de code donnée, ou toute une partie du programme.

  Cette fois, c'est à l'utilisateur de guider l'outil, afin de lui donner des indications sur les valeurs inconnues à ce stade (contenu des variables, taille des tableaux passés en arguments...).

- Il faut ensuite compiler chaque module du projet et en faire un exécutable.

  À ce niveau, la compilation séparée peut être faite « intelligemment » (génération automatique de *makefiles*) à partir de l'éditeur.

  Pour cela, la notion de *projet* comme on la trouve dans les interfaces de BORLAND permet de simplifier la compilation. Un tel projet peut passer par l'utilisation d'une structure propre à l'éditeur comme le fait l'analyseur de **FORESYS**.

  De plus, l'éditeur doit permettre d'établir rapidement les liens entre les messages d'erreurs et les lignes erronées.

  Enfin, la gestion de versions et d'historique des modifications doit aussi être intégrée à ce niveau.

- Lors de l'exécution du programme, le traçage de son déroulement et la visualisation du contenu des variables distribuées doivent être effectués à partir de l'interface de navigation (idées 8 et 15).

  L'exécution sur une machine donnée doit permettre d'extrapoler les résultats sur une autre machine, à partir de la description de son architecture (idée 9).

  Lorsque c'est possible, l'instrumentation du code doit être réduite au minimum et doit être transparente à l'utilisateur (par une option de compilation par exemple) (idées 13 et 14).

Il est clair que l'élaboration d'un tel environnement dépasse largement le cadre d'une thèse. De plus, comme on l'a vu dans la section 2.4.6, certains projets ont déjà pour but de mettre en place un tel atelier de développement.

Cependant, on peut remarquer qu'au niveau de l'édition interactive de directives de placement, il n'y a pas encore d'outils disponibles (c'est seulement au stade de projet dans FITS et HPFIT).

Le placement des données est pourtant un problème clé dans un programme HPF, puisqu'il a une influence directe sur les performances. C'est donc sur ce point que nous allons nous pencher dans les chapitres suivants.

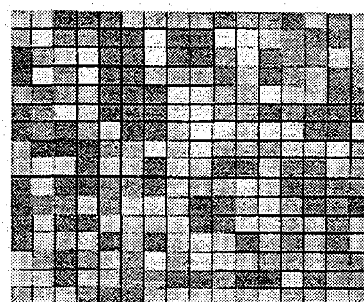
On a vu qu'il faut non seulement aider à l'insertion des directives, mais qu'il faut aussi donner une idée à l'utilisateur des conséquences de ces directives.

Le prochain chapitre va donc présenter un prototype d'interface graphique d'aide à l'édition de directives HPF. Ensuite, dans le chapitre 4 nous proposeront quelques solutions d'évaluation *a priori* des performances résultant d'un placement donné.

Chapitre 3

HPF-Builder

Une distribution hétéroclite :



Checkerboard with Light Colors,
Piet Mondrian, 1919

3.1 Introduction

Les chapitres précédents ont montré qu'une grande partie des utilisateurs de l'informatique scientifique s'est orientée vers l'utilisation de machines parallèles, mais continue à vouloir utiliser FORTRAN comme langage de programmation.

Transformer des programmes FORTRAN existants afin de les adapter à ces nouvelles machines n'est pas une tâche facile. Les efforts d'automatisation de ce travail ont abouti à des outils qui ne sont efficaces, voire utilisables, que dans des cas restreints.

Le langage FORTRAN a donc évolué dans ce sens et introduit dans sa dernière normalisation des notions de parallélisme de données, modèle de programmation bien adapté à de nombreux algorithmes fréquemment rencontrés dans le domaine de l'informatique scientifique.

Le développement en HPF

Porter un programme vers le modèle à parallélisme de données consiste principalement en deux étapes : l'ordonnancement des boucles sous des formes parallélisables et le placement des données de façon à minimiser les communications induites.

L'ordre dans lequel ces deux étapes doivent être réalisées est encore un problème ouvert. En effet, ces deux tâches sont fortement dépendantes l'une de l'autre sur certains points.

De plus, une fois les boucles ordonnancées, le compilateur doit être capable de détecter qu'il est en présence d'une situation favorable, qu'il peut optimiser.

HPF semble donc promis à un bel avenir, puisqu'il ajoute aux notions de parallélisme de FORTRAN la possibilité d'optimiser les programmes, indépendamment de l'architecture de la machine cible, grâce à des directives permettant d'indiquer au compilateur des possibilités d'optimisation des boucles, ainsi que des instructions de placement des données.

L'aide au développement en HPF

Le réordonnancement automatique de nids de boucles est une voie déjà bien explorée [PRi, PIP], mais dont les mises en application sont peu satisfaisantes. C'est pourquoi des logiciels interactifs sur ce thème commencent à apparaître [For]. Ils permettent à l'utilisateur d'effectuer ces ordonnancements "à la main", mais intègrent des outils automatiques afin de l'aider dans ses choix.

En revanche, du côté du placement des données, les méthodes automatiques commencent à peine à apparaître et les interfaces interactives sont quasiment inexistantes ou dédiées à un compilateur ou une machine particulière [GDD]. Pourtant, le chapitre précédent a montré que les placements ont un aspect géométrique qui facilite la mise en œuvre d'une approche graphique d'un tel outil.

Notre but est donc de commencer à combler ce vide en proposant un outil d'aide à l'insertion de directives de placement de données dans des programmes FORTRAN.

Aspect visuel du placement de données

Lorsqu'un programmeur commence à étudier les placements qu'il va mettre en place, il commence généralement par faire des croquis de ses structures de tableaux pour y dessiner les accès.

À partir de ces dessins, il en déduit les parties qui interagissent le plus et qui ont donc intérêt à être placées au même endroit. Il décide ainsi des alignements. Une fois cette partie effectuée, il peut évaluer les quantités de calcul à effectuer autour de chaque élément de `template` et en déduire une distribution.

Cette démarche pourrait être utilisée non plus via un crayon et du papier, mais à travers une interface graphique, c'est-à-dire via une souris et un écran.

L'intérêt de cette approche est qu'il est alors possible d'interpréter automatiquement le sens du dessin pour générer les directives HPF correspondantes. C'est cette idée que nous avons cherché à développer et qui est à l'origine du projet qui va être décrit dans ce chapitre.

L'aide au placement des données

Une telle interface graphique doit constituer un premier pas vers le « placement assisté par ordinateur » et peut servir de plateforme pour y greffer d'autres outils ou, au contraire, venir s'intégrer dans un environnement existant.

L'idée de base de cet outil est donc de proposer au programmeur un moyen de spécifier des placements de données autrement qu'en écrivant les directives HPF une par une :

À partir d'un programme FORTRAN 95, une interface graphique lui propose de visualiser les tableaux et de spécifier interactivement leur placement. Les directives HPF correspondantes sont alors insérées automatiquement dans le programme par l'outil.

Une fois la première ébauche de placement effectuée, des tests permettent éventuellement de décider des modifications dans ces placements. Ces tests sont effectués en traçant une exécution, ou en évaluant les coûts induits par le placement, à partir de la même interface graphique.

L'outil permet donc de reprendre un code source HPF existant, afin de visualiser les directives qu'il contient et de les modifier interactivement et toujours de façon visuelle.

Nous allons donc détailler les fonctionnalités intéressantes à intégrer à un tel outil et présenter une façon de les organiser, après quoi nous présenterons l'implémentation qui a été développée à partir de ce cahier des charges.

Avant de continuer, nous allons détailler quelques points de vocabulaire :
Nous appellerons *contexte*, une partie de programme telle qu'une fonction, une procédure, un programme ou un module.
Nous appellerons *objet* une structure telle qu'un tableau, une variable, un `template` ou un `processors`. De plus, nous appellerons un `template` ou un `processors` un *objet HPF*.
Enfin, nous appellerons *lien* une directive reliant deux objets, c'est-à-dire une spécification d'alignement ou de distribution.

3.2 Concept, fonctionnalités

Comme le montre la figure 3.1, un source HPF[1] contient un certain nombre d'informations sous forme textuelle. Cette forme est compacte, mais n'est pas aisée à appréhender globalement. Certaines informations sont difficiles à retrouver rapidement, surtout au-delà d'une certaine taille de programme.

1. Dans la suite du document, nous considérerons la version 1.0 d'HPF, c'est-à-dire la version 2.0, hors extensions, avec en plus les directives dynamiques. Nous ne prendrons donc pas en compte les sous-ensembles de `processors`, les distributions généralisée ...

En effet, quand le projet a été lancé, la version 2.0 d'HPF n'était pas encore parue, nous sommes donc partis sur la base d'HPF 1.0.

Quand la version 2.0 a été publiée, les directives dynamiques ont été renvoyées avec les autres extensions approuvées. Comme il était alors difficile de savoir quelles extensions seraient mises en œuvre, nous avons préféré attendre une certaine stabilisation des compilateurs avant d'intégrer des visualisations qui risquaient de n'être jamais utilisées. Par contre, les directives dynamiques étant déjà mises en place dans le prototype, nous les avons laissées.

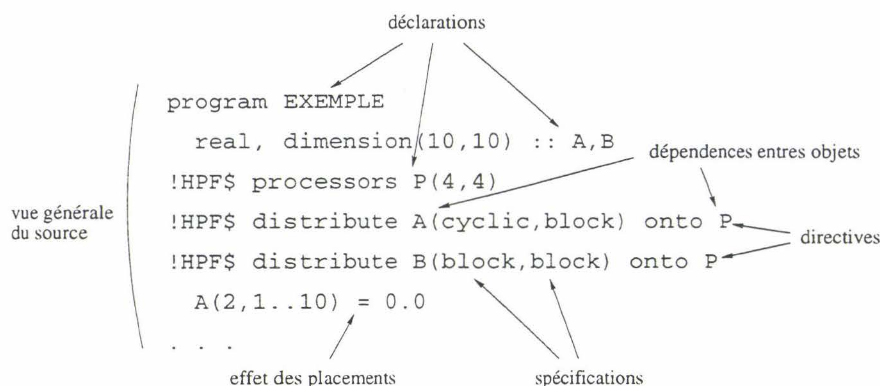


FIG. 3.1: *Informations à visualiser.*

Ainsi, dans les six lignes de code montrées dans la figure, on trouve :

- des déclarations de programme et de variables,
- des directives HPF,
- des liens entre objets, du fait de ces directives,
- des spécifications dans les directives,
- une affectation dont l'efficacité dépend des directives.

Toutes ces informations ont une influence à des niveaux différents, alors qu'elles apparaissent toutes « en vrac » dans la même partie du programme.

De plus, quand le point de départ est un programme FORTRAN 95, l'insertion des directives HPF n'est pas immédiate. La syntaxe des directives est assez lourde et il est facile de se tromper dans les bornes des spécifications. Ainsi, des données dont on croit avoir amélioré la localité se retrouvent systématiquement distantes et il devient difficile de retrouver l'origine du problème.

Un outil aidant à visualiser de manière plus confortable ces informations et permettant de les manipuler interactivement doit donc aider à isoler une partie de ces informations selon des critères différents.

Il est donc intéressant de mettre en place différents points de vue, chacun ayant pour rôle de mettre en valeur de façon visuelle une partie de l'information[Lef98] :

Vue générale du source : bien que notre but soit de s'écarter de la forme textuelle du langage, il est nécessaire de donner à l'utilisateur la possibilité de voir les fichiers sources.

En effet, on a vu au chapitre précédent que le tout visuel a plutôt tendance à perdre l'utilisateur. Il faut donc lui permettre de modifier son programme de façon graphique, mais en lui laissant un moyen d'observer « dans le texte » l'évolution du programme source.

Un programme est généralement constitué de plusieurs fichiers sources. Il doit donc être possible de naviguer entre les différents fichiers. De plus, dans chaque fichier, les parties directement liées au placement (déclarations et directives) doivent être mises en valeur, afin que le programmeur puisse les repérer plus facilement.

Vue résumée du source : comme certaines parties du code ne sont pas intéressantes dans cette phase de développement, il devient difficile de retrouver les objets dont l'utilisateur a besoin au milieu d'un programme volumineux, même si les déclarations sont mises en valeur.

Quand on observe une instruction, la déclaration de ses opérandes peut être située bien plus haut dans le texte, voire dans un autre module. Un résumé des informations pertinentes est donc intéressant pour guider l'utilisateur.

Bien sur, on doit pouvoir garder le lien entre les déclarations et le code où elles interviennent. Il faut donc garder la possibilité d'être renvoyé à la ligne de code correspondant à une entrée dans le résumé.

Vue de l'organisation générale des directives : si les objets à manipuler n'apparaissent que dans leur ordre de déclaration, il est difficile de retrouver à quel objet est associée une directive donnée.

Il faut donc disposer d'une vue représentant les dépendances entre objets et directives. Ainsi, lorsqu'il observe une instruction, l'utilisateur peut retrouver facilement les directives de placement qui interviennent sur les opérandes. De même, quand il édite une directive de placement, la déclaration des objets source et cible est repérable immédiatement.

Là encore, le renvoi vers le résumé ou le source est nécessaire. Ainsi, on a accès à la structure globale aussi bien dans l'ordre syntaxique que dans l'ordre de l'architecture des directives HPF. On obtient donc une interface de navigation évoluée à travers le source.

Vue globale de chaque objet et directive : à partir de chacune des vues précédentes, le programmeur doit pouvoir accéder à une vue générale des caractéristiques de l'objet qu'il sélectionne. Ainsi, pour une directive de placement, il doit pouvoir obtenir ses spécifications ainsi que les objets source et destination.

Vue détaillée de chaque objet et directive : cette vue globale doit être accompagnée de la possibilité de savoir précisément à quel endroit est placé un élément d'un objet donné, ou, au contraire, quels éléments sont placés à une position donnée.

À partir de l'instruction qu'il observe, le programmeur peut ainsi suivre les directives qui spécifient le placement des opérandes et observer précisément l'emplacement où est projeté chaque élément de ces opérandes.

Création/destruction de directives : le premier but du projet étant de partir de programmes FORTRAN et d'y ajouter des directives HPF, il faut bien sûr être capable d'insérer des directives de placement dans le source, et de façon visuelle.

Pour cela, le programmeur doit pouvoir sélectionner les éléments à placer et spécifier graphiquement la manière dont ils doivent être alignés ou distribués. Ainsi, il peut définir les directives de placement sans avoir de problèmes de traduction en syntaxe HPF.

Le risque d'erreurs de syntaxe, ou de décalages dans les indices est donc réduit et l'insertion automatique de certains points (emplacement des directives, spécifications par défaut) accélère le développement.

Modification de directives : une fois créées, ces directives doivent pouvoir être modifiées afin d'affiner le placement.

Là encore, l'édition graphique permet de contrôler la cohérence des directives (dépassement de bornes, précédence de déclarations ...) et permet de mettre à jour automatiquement les différentes vues.

Évaluation des coûts : c'est du placement que dépend l'efficacité du programme, mais visualiser ces placements ne permet pas directement d'en déduire les performances du code. Il est donc nécessaire d'offrir à l'utilisateur des moyens d'évaluer les coûts induits par ce qu'il a défini, afin de pouvoir améliorer ses placements.

Le projet HPF-BUILDER a été développé dans le but de remplir ce cahier des charges. Pour répondre à ses premiers points, nous avons opté pour une visualisation à quatre niveaux qui seront détaillés dans les quatre sections suivantes. Pour l'évaluation des placements, le problème sera traité dans le chapitre suivant.

La description de chaque niveau sera accompagnée d'une figure d'exemple représentant l'extrait de code montré dans la figure 3.1 page 74.

3.2.1 Niveau global

```
— program EXEMPLE
  ├ array A
  ├ array B
  ├ processors P
  ├ distribute A → P
  └ distribute B → P
```

FIG. 3.2: *Niveau global.*

Ce premier niveau est le plus général. Il s'agit de représenter dans son ensemble le programme HPF considéré.

La vue du source se fait de façon textuelle, comme avec un éditeur de texte classique. Pour améliorer sa lisibilité et faciliter la reconnaissance d'éléments syntaxiques importants, ceux ci peuvent être indiqués par des couleurs ou polices différentes.

En plus de sa forme textuelle, un résumé des éléments syntaxiques qui serviront à définir des directives doit être présent. Pour garder l'ordre des déclarations, une solution simple consiste à représenter ce résumé sous la forme d'une liste dans laquelle chaque entrée correspond à un contexte (un fichier, un programme ou une procédure). Les déclarations d'objets et de liens contenues dans ce contexte sont alors réunies dans une sous-liste (voir figure 3.2). De même, les sous-contextes (procédures d'un module, `contains` d'une procédure ...) sont inclus dans un niveau de liste inférieur.

Ce niveau correspond donc aux deux premiers points du cahier des charges.

C'est à partir de ce niveau que doit se faire l'insertion de déclarations d'objets HPF, puisque l'utilisateur doit quand même décider du contexte auquel appartient une directive.

En effet, des contraintes sur les précédences de déclaration (dans une déclaration, on ne peut faire référence qu'à des objets déjà déclarés) imposent en partie la position des directives. L'utilisateur n'a qu'à préciser le contexte dans lequel il veut insérer une directive et celle ci peut être placée automatiquement à la fin de la section des déclarations de ce contexte. Cependant, dans le cas des directives dynamiques, l'utilisateur doit spécifier à quel endroit du code il veut effectuer le replacement. Il est impossible de décider cette position à sa place.

3.2.2 Niveau hiérarchique

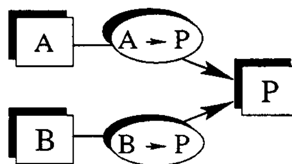


FIG. 3.3: *Niveau hiérarchique.*

Le but de ce niveau et de mettre en place le troisième point du cahier des charges. Il s'agit donc de montrer l'ensemble des objets et liens du programme, non plus dans l'ordre de leur déclaration, mais d'une façon qui permet de comprendre visuellement leurs interactions.

Ces interactions présentent naturellement la forme d'un graphe dont les nœuds sont les objets du programme et chaque arc une directive les reliant (voir figure 3.3).

Ainsi, une directive d'alignement peut être déclarée dans une procédure d'un module et avoir comme cible un `template` déclaré dans l'entête du module. Dans ce cas, ce `template` ne figure pas dans la même partie de l'arbre des déclarations décrit précédemment, alors qu'il est voisin de la directive dans le graphe.

À ce niveau, on peut donc suivre rapidement le cheminement de directives qui amène un tableau donné vers le `processors` sur lequel il est placé, indépendamment de l'endroit où est déclaré chaque maillon de cette chaîne de placements.

À partir de ce niveau, l'insertion de liens peut être effectuée en désignant simplement les objets source et destination. Le contexte de chaque objet permet de déterminer automatiquement le contexte dans lequel la nouvelle directive doit être insérée. S'il s'agit d'une directive dynamique, un renvoi au source doit demander la position précise de la nouvelle directive.

3.2.3 Niveau directive

Chaque nœud ou lien de ce graphe constitue donc une déclaration de variable ou de directive HPF. Pour chacun d'eux, l'utilisateur doit avoir accès à des informations générales telles que la géométrie des objets, ou les spécifications des directives.

Pour cela nous proposons de rendre « sélectionnable » chacun de ces éléments du graphe.

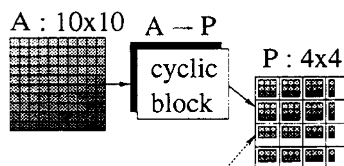


FIG. 3.4: *Niveau directive.*

Cette sélection fait alors apparaître l'élément à un niveau plus précis, dans lequel ses caractéristiques sont représentées graphiquement.

Un placement constitue une projection des éléments d'un nœud dans un autre. Quand un lien est sélectionné, la projection de sa source peut donc être affichée dans la représentation de sa cible.

À partir de ces sélections, il devient possible d'accéder aux informations détaillées sur les objets et liens (géométrie, taille, spécification l'alignement ou de distribution ...). C'est donc à ce niveau que ces informations pourront être éditées. Toute modification des caractéristiques d'un objet est alors répercutée automatiquement dans le fichier source et l'objet est mis à jour dans les autres vues. De plus, la cohérence des directives est vérifiée afin de signaler toute erreur à l'utilisateur.

Il est important de n'autoriser que la modification des objets spécifiques à HPF. Notre but n'étant pas d'intervenir sur l'algorithme lui même, il est interdit de changer la spécification d'un tableau.

Enfin, pour faciliter la navigation entre les différentes vues, il doit être possible de sélectionner un objet ou un lien à partir des autres vues. De plus, il faut pouvoir sélectionner plusieurs objets, afin de pouvoir observer simultanément plusieurs maillons d'une chaîne de placements. Chaque sélection peut être repérée par une couleur.

3.2.4 Niveau localité

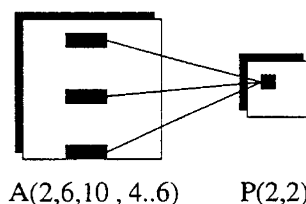


FIG. 3.5: *Niveau localité.*

Ce dernier niveau permet à l'utilisateur de visualiser en détail le résultat de ses placements. En effet, le principe de la projection entre les vues montre globalement l'aspect d'un placement, mais ne permet pas de déterminer précisément sur quel processeur abstrait est projeté un élément de tableau donné.

Pour cela, le niveau détaillé fournit un moyen de suivre la projection d'un élément donné. L'utilisateur spécifie une position dans un objet en déplaçant un curseur dans la vue d'une

sélection (`P(2,2)` dans la figure). La projection de cette position apparaît alors dans les autres sélections, sous la forme de pavés à l'intérieur des autres vues (dans l'exemple, `A(2,6,10 , 4..6)`). Les coordonnées exactes de cette position doivent également être affichées, afin que l'utilisateur puisse comparer les projections de différents objets. Le curseur peut ensuite être déplacé vers une cellule voisine, par un simple clic.

Ainsi, le programmeur a la possibilité d'observer la position exacte de différents éléments de tableaux, à travers les divers placements qu'il a spécifié. Pour une instruction donnée, il peut alors vérifier précisément la localité des opérandes.

C'est donc à ce niveau que l'utilisateur peut observer l'effet du placement qu'il a défini et donc son efficacité.

3.3 Mise en œuvre

Nous allons maintenant décrire HPF-BUILDER, l'implémentation qui a été développée à partir du cahier des charges décrit précédemment et des premières versions du projet[DL97, 1.195].

Nous allons commencer par expliquer l'organisation générale du projet, avant de détailler chacune de ses parties.

La description de l'interface graphique sera relativement succincte. En effet, il est assez lourd d'expliquer sur papier le fonctionnement d'une telle interface. C'est pourquoi nous conseillons au lecteur de parcourir la démonstration sur CD-ROM parallèlement à la lecture de cette partie.

3.3.1 Organisation générale

Nous avons vu au chapitre précédent le cahier des charges général d'un atelier d'aide au développement en HPF. Le projet HPF-BUILDER a pour but de constituer une ébauche d'un tel environnement.

L'idée générale consiste à disposer d'un éditeur associé à un ensemble d'outils aidant le programmeur aux divers stades du développement.

Tous ces outils ont besoin d'informations sur le programme courant et plusieurs outils peuvent avoir besoin des mêmes informations. De plus, la plupart des données à recueillir passent par une analyse syntaxique du source. Il semble donc intéressant de disposer d'un analyseur commun, qui servira de *serveur d'informations* pour les autres composants, comme montré dans la figure 3.6.

Nous avons vu dans l'introduction que le projet HPF-BUILDER s'intéresse au placement des données et qu'il doit permettre d'éditer des directives existantes. Il a donc besoin d'accéder à ce serveur.

Le développement du projet HPF-BUILDER est donc séparé en deux parties : le serveur d'informations et l'interface graphique.

La version actuelle d'HPF-BUILDER fait une entorse à cette organisation : il n'y a pas d'éditeur de texte proprement dit. C'est donc une fenêtre de visualisation de texte qui en fait

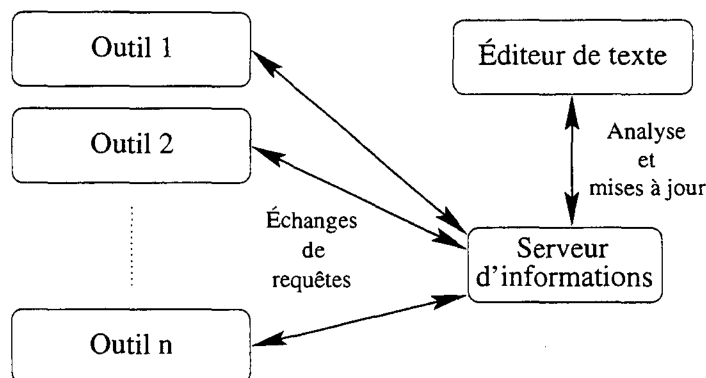


FIG. 3.6: *Organisation autour d'un serveur d'informations.*

office et qui est intégrée à l'interface graphique.

Cette organisation est utilisée pour des raisons de facilité : si on utilise un éditeur de texte, cela autorise l'utilisateur à modifier son programme à tout moment, comme il le veut. Il est alors nécessaire de mettre à jour, en direct, toutes les structures internes du serveur d'informations, à chaque modification.

Il en résulterait une lourdeur inacceptable. Il semble donc plus adapté de bloquer l'éditeur de texte dans un mode *read only* lorsqu'un outil annexe est utilisé. Dans ce mode dégradé, seul les outils peuvent effectuer des modifications dans le source, par l'intermédiaire du serveur.

Comme la version actuelle d'HPF-BUILDER n'implémente qu'un seul outil, il est apparu plus simple de n'utiliser qu'une simple fenêtre de visualisation à la place d'un éditeur.

Dans la version actuelle, le serveur s'adresse donc à un des éléments de l'interface graphique pour effectuer ses analyses. C'est la seule modification à apporter pour l'adapter à une utilisation avec un éditeur indépendant.

L'interface graphique et le serveur d'informations sont mis en œuvre par deux exécutables séparés, que nous allons maintenant présenter.

3.3.2 L'interface graphique

L'interface graphique constitue donc la partie visible de l'iceberg. Elle met en place les quatre niveaux décrits dans la section précédente, par l'intermédiaire de quelques modules organisés comme montré dans la figure 3.7

On voit qu'une certain nombre de modules communiquent entre eux et vont chercher des informations sur le serveur, via un module d'interface. Les quatre niveaux de visualisation sont respectivement mis en place par les modules `source` et `tree`, `skel`, `icons` et `zoom`. La notion de sélection est implémentée par le module `selection`, qui assure la cohérence entre les modules. Enfin, un ensemble de modules d'édition communique avec les autres modules pour effectuer les modifications demandées par l'utilisateur.

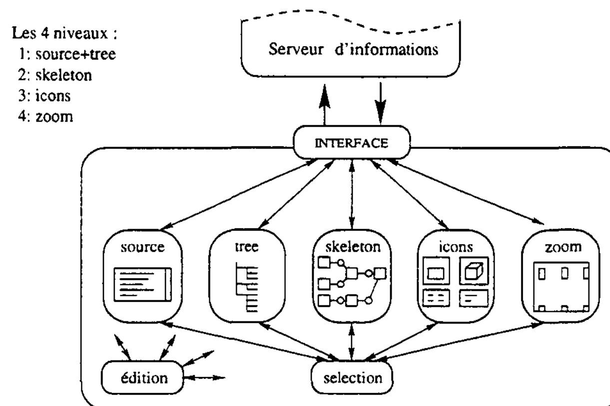


FIG. 3.7: *Organisation de l'interface graphique.*

Choix du langage de programmation

Une fois que l'organisation générale du projet est établie, il faut décider du ou des langages de programmation à utiliser.

Pour cela, un certain nombre de contraintes doivent être prises en compte :

- Il s'agit d'une interface graphique, ce qui implique l'utilisation d'une bibliothèque dédiée aux interfaces graphiques. Le langage utilisé doit donc être capable d'utiliser des bibliothèques telles que X11 ou le SDK de WINDOWS.
- Il est intéressant de disposer d'un générateur d'interfaces, ou d'un langage à partir duquel la description d'interfaces est facile.
- L'interface ne doit pas seulement gérer des boutons, des menus et d'autres *widgets* « classiques ». Il est également nécessaire de faire des dessins assez complexes (vus en fils de fer, arborescences ...). Le langage doit donc permettre d'accéder à des fonctions de dessins et de gérer l'interactivité autour de ces dessins.
- HPF-BUILDER n'a pas de vocation commerciale. Il est prévu de le distribuer librement. Il n'est donc pas possible de le distribuer avec des bibliothèques payantes.
- Enfin, il n'y a pas de contraintes de performances. Il faut aller à la vitesse d'une utilisation interactive, c'est-à-dire relativement lentement, par rapport à des applications de calcul.

Pour toutes ces raisons, notre choix s'est porté sur TCL/TK[Ous].

TCL est un langage interprété, à la syntaxe proche de C-SHELL. C'est plus précisément un langage pseudo-interprété : un mécanisme de compilation au vol permet de transformer le source lu en un pseudo-code. Ainsi, dans une boucle, par exemple, dès le deuxième tour, le pseudo-code est exécuté à la place du source original, ce qui améliore considérablement les performances.

Son principal intérêt réside dans la possibilité de lui ajouter très facilement de nouvelles fonctionnalités. Il est en effet possible d'ajouter de nouvelles fonctions et même de nouvelles structures de contrôle, en le liant à des bibliothèques dynamiques écrites en C. Il est donc possible d'adapter le langage aux besoins particuliers d'une application donnée.

Un autre atout d'importance dans TCL réside dans une de ses extensions : TK.

Cette bibliothèque met en place toutes les fonctionnalités nécessaires pour gérer des interfaces graphiques. Elle permet de créer des *widgets* classiques et de les gérer de façon événementielle.

TK effectue la gestion des événements de façon automatique et fournit des méthodes de placement semi-automatique des *widgets*. La programmation est donc grandement simplifiée.

Enfin, des fenêtres de dessins peuvent être utilisées. Dans ces fenêtres, il est possible de placer des polygones, des textes et même des sous-fenêtres. La puissance de ces fenêtres de dessin réside dans la gestion de ces objets : on les utilise comme des *widgets*. On peut leur associer des actions à effectuer en réactions à certains événements.

Afin de simplifier le développement, la bibliothèque TIX a également été utilisée. C'est une extension de TK qui définit des *widgets* complexes comme des fenêtres associées automatiquement à des ascenseurs, des « boites à onglets » ...

Fonctionnement général

Une fois le langage de programmation et les bibliothèques principales choisies, une définition plus détaillée, suivie d'une (longue !) étape de programmation ont conduit à la version actuelle, que nous allons maintenant décrire.

Après avoir lancé HPF-BUILDER, l'utilisateur peut demander à travailler sur un ensemble de fichiers sources constituant un programme FORTRAN (voir figure 3.8). On appelle cet ensemble de fichiers un *projet*.

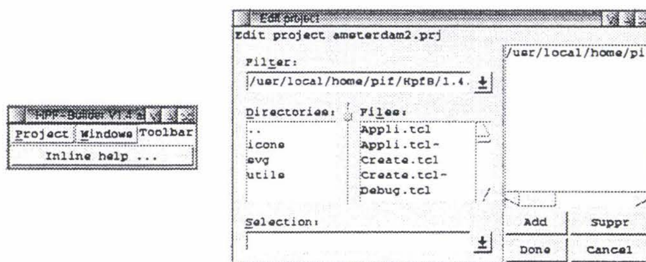


FIG. 3.8: *Insertion de fichiers dans un projet.*

L'interface graphique s'adresse alors au serveur pour effectuer l'analyse des fichiers. Elle lit ensuite les informations dont elle a besoin pour effectuer la visualisation.

Par la suite, toutes les modifications faites par l'utilisateur sont répercutées au serveur afin qu'il mette à jour ses structures de données et le programme source.

Le module `selection`

Ce module gère la sélection d'objets et de liens dans les différentes fenêtres de l'interface.

Dès que l'utilisateur sélectionne ou désélectionne un objet dans une des fenêtres, le module qui gère cette fenêtre fait suivre cette requête au module `selection`, qui répercute cette action dans les autres fenêtres.

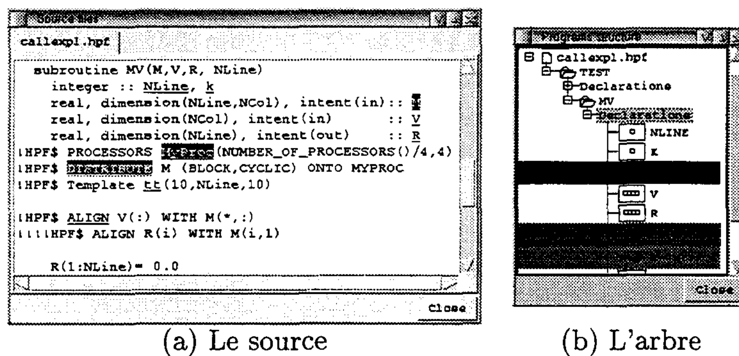
De plus, le module stocke quelques informations sur les sélections courantes, afin de pouvoir renseigner les autres modules. Il fait donc office de serveur à propos de l'état actuel des constituants de l'interface graphique.

À tout moment, l'utilisateur peut effectuer jusqu'à cinq sélections simultanées. Cette limite est généralement suffisante, puisqu'elle permet d'afficher les cinq éléments d'une suite tableau → alignement → `template` → distribution → `processors`, ou d'afficher deux objets, projetés par deux liens sur un même objet destination.

Ce module constitue donc le « chef d'orchestre » de l'application. C'est lui qui sert de serveur d'informations spécifiques à l'interface graphique.

Le module `source`

Comme le montre la figure 3.9(a), ce module remplace ce qui devrait être un éditeur de texte indépendant. Il gère tout ce qui est relatif à la visualisation des fichiers sources du projet courant.



(a) Le source (b) L'arbre

FIG. 3.9: *Niveau 1 de la visualisation.*

Il s'agit donc d'afficher chaque fichier source dans une sous-fenêtre, mais aussi d'effectuer la mise en valeur des éléments syntaxiques intéressants et le surlignage de ceux qui sont sélectionnés.

Pour cela, le module effectue une série de requêtes au serveur pour identifier chaque élément syntaxique à mettre en valeur. Chacun d'eux est alors souligné et peut être cliqué pour le sélectionner ou le désélectionner.

De plus, c'est à partir de ces fenêtres que l'utilisateur désigne les points d'insertion des nouvelles directives, lorsque c'est nécessaire. Pour cela, lors d'une insertion, le module reçoit un ordre lui demandant de mettre en valeur la partie du code où cette insertion est possible. L'utilisateur n'a alors qu'à cliquer la ligne ou il veut effectuer l'insertion et le module retourne

cette information au demandeur afin qu'il effectue l'insertion, par une série de requêtes au serveur.

Ce module gère donc la référence textuelle que l'utilisateur peut consulter pour voir l'évolution de son code tout au long des modifications.

Le module `tree`

Ce module gère l'affichage d'une liste arborescente résumant les déclarations contenues dans le source. Lors du chargement d'un projet, il récupère les informations obtenues par le module `source` afin de créer l'arbre montré figure 3.9(b).

Chaque entrée de cette hiérarchie de listes peut être cliquée pour la sélectionner ou désélectionner, comme dans le source.

Là aussi, les objets sélectionnés sont surlignés dans la couleur de la sélection.

Cette vision résumée du source permet de repérer plus rapidement une déclaration d'objet dont on connaît le contexte, alors que, dans le source du programme, une déclaration n'est pas plus visible qu'une autre. Pour améliorer le repérage, une icône précède chaque nom d'objet. Cette icône représente le type de l'objet et, pour un nœud, son nombre de dimensions.

Enfin, chaque sous-liste peut être ouverte ou refermée afin de réduire la taille de l'arbre. Par exemple, sur la figure 3.9(b), les déclarations du programme `TEST` sont refermées, alors que celles de la procédure `MV` sont affichées.

Ce module apporte donc une aide supplémentaire au repérage d'objets dans le code, qui s'ajoute à la vue du source.

Le module `skeleton`

Ce module affiche ce que nous avons appelé le squelette, c'est-à-dire l'arborescence des objets et des liens selon leurs dépendances.

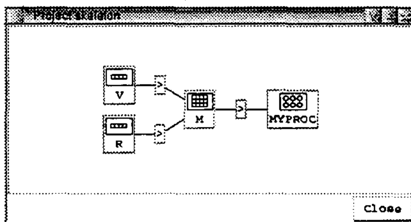


FIG. 3.10: *Niveau 2 de la visualisation.*

Chaque nœud de cet arbre est un objet et chaque branche est une directive dont le père et le fils sont respectivement la source et la cible.

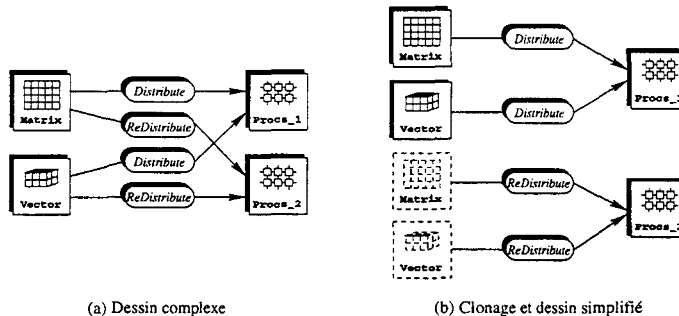
Comme le montre la figure 3.10, les nœuds sont représentés par des icônes contenant le nom de l'objet et une forme indiquant leur type. Les arcs sont marqués par une autre icône en forme de flèche.

Pour ne pas alourdir inutilement le graphique, les nœuds qui ne sont attachés à aucune directive ne sont pas affichés, tant qu'ils ne sont pas sélectionnés à partir d'une autre fenêtre.

Ainsi, dans la figure 3.10, on voit tous les objets de la procédure `MV`: les vecteurs `M` et `V` sont alignés sur `M`, lui-même distribué sur `MyProc`.

Les icônes peuvent être cliquées pour gérer les sélections comme dans les autres fenêtres.

La présence de directives dynamiques complique la représentation de cette arborescence (voir figure 3.11(a)).



(a) Dessin complexe (b) Clonage et dessin simplifié

FIG. 3.11: *Représentation des directives dynamiques.*

En effet, si un objet et placé sur un autre, puis replacé sur un troisième, on n'a plus un arbre, mais un graphe orienté. Or les graphes sont plus compliqués à représenter. De plus, si un objet est replacé sur la même cible, mais avec des spécifications différentes, on obtient deux arcs entre les mêmes nœuds.

Pour éviter ces complications qui alourdissent la représentation graphique, nous avons décidé de représenter les objets en autant d'exemplaires qu'ils apparaissent comme source dans une directive (voir figure 3.11(b)). Dès qu'un des nœuds est sélectionné, chacun de ses clones l'est aussi.

Avec ce module, on arrive donc à une interface complète de navigation dans le programme source HPF. L'utilisateur a la possibilité d'accéder aux déclarations des différents éléments qui interviennent dans les placements. Il peut passer de l'une à l'autre aussi bien par des critères syntaxiques (ordre des déclarations) que hiérarchiques (directives entre objets).

Le module `icons`

Lorsqu'une des icônes du squelette est sélectionnée, elle est transformée en sous-fenêtre présentant les caractéristiques de l'icône. C'est à ce niveau que les caractéristiques des objets vont pouvoir être modifiées.

La figure 3.12 présente l'exemple de la figure 3.10, avec les cinq icônes de la partie gauche sélectionnées.

Les deux sélections de gauche représentent des vecteurs dont on voit en en-tête le nom, le nombre de dimensions et la taille. Dans le cas d'un objet HPF (on a vu qu'il était interdit de modifier un tableau) chaque donnée de cet en-tête est modifiable.

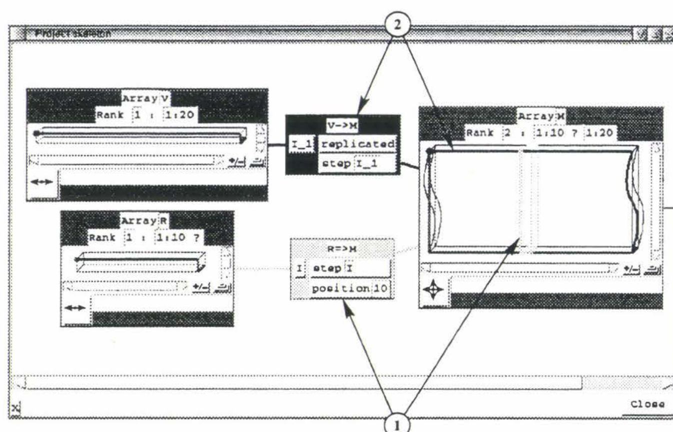


FIG. 3.12: *Niveau 3 de la visualisation.*

En-dessous, est affichée une représentation en « fil de fer ». On y voit un coin marqué par un cercle. En partant de cette marque, on peut suivre un bord plus épais. Ce bord marque les dimensions successives de l'objet. Ainsi, dans le cas d'un objet à 2 ou 3 dimensions, on peut repérer l'ordre des dimensions.

C'est dans cette vue que seront dessinées les informations relatives au placement. Cette représentation est pratique quelle que soit la taille de l'objet. En effet, le dessin peut être zoomé et redimensionné grâce aux deux boutons situés dans le coin du dessin, entre les ascenseurs.

Les deux sélections centrales montrent deux directives vers le tableau `M`. Dans celle du haut, l'en-tête indique `V`$\rightarrow$`M`, c'est-à-dire une directive d'alignement de `V`. Par contre, l'autre indique `R`$\Rightarrow$`M`, avec une double flèche qui indique une directive dynamique, c'est-à-dire un réalignement.

En dessous de ces en-têtes, deux colonnes représentent les deux parties de la directive, c'est-à-dire la liste des indices utilisés pour désigner les dimensions de la source et les spécifications d'alignement. Chacune de ces boîtes est éditable.

On peut remarquer dans les boîtes de la sélection du haut qu'un nom d'indice `I_1` a été pris automatiquement. En effet, la directive utilise un symbole : comme indice de spécification. Or, si l'utilisateur modifie la directive, ce symbole peut devenir invalide. L'éditeur force donc l'indice à un nom utilisable dans tous les types de spécifications.

La sélection de droite montre un tableau dans lequel sont représentés graphiquement les alignements des deux vecteurs.

Chaque vue en fil de fer est dans la couleur de la sélection qu'elle représente.

On voit donc que la sélection (1) représente l'alignement du vecteur `R`, par la directive `R`$\Rightarrow$`M` et qu'il s'agit d'un placement sur la dixième colonne de `M`.

Pour la sélection (2), on voit la marque du point d'origine en haut à gauche. Ensuite, le bord épais suis une ligne de `M` puis descend avec une forme courbée qui signale une réplication.

`V` est donc répliqué sur chaque ligne de `M`.

Cette représentation en fil de fer diffère volontairement des représentations « classiques » par des jeux de couleurs. En effet, beaucoup d'outils utilisent une couleur par élément de la destination et colorient les éléments de la source avec ces couleurs, selon la distribution (voir les exemples de GDDT, page 63 et ANNAI/DDV, page 65). Cette méthode est pratique pour des distributions sur des `processors` de petite taille, mais devient illisible sur des objets de grande taille ou à trois dimensions.

Par contre, la méthode en fil de fer permet d'avoir une idée du placement, quelle que soit la taille des objets en présence, puisque les lignes reste visibles de la même façon, à n'importe quel niveau de zoom[2].

Toute modification est répercutée aux autres fenêtres ainsi qu'au serveur qui met à jour le source. De plus, lors de ces modification, la cohérence des directives est vérifiée : si une directive n'a pas autant de spécifications qu'il y a de dimensions dans l'objet associé, si un alignement sort des bornes de sa cible ou si une distribution par blocs doit « cycler », un drapeau signale l'erreur près de la spécification erronée.

L'interface graphique n'interdit pas les directives erronées, elle ne fait que les signaler. Ainsi, un code contenant des erreurs peut être chargé afin de le corriger, sans « planter » l'interface graphique. Bien sûr, dans ce cas, les visualisations peuvent avoir des formes imprévisibles.

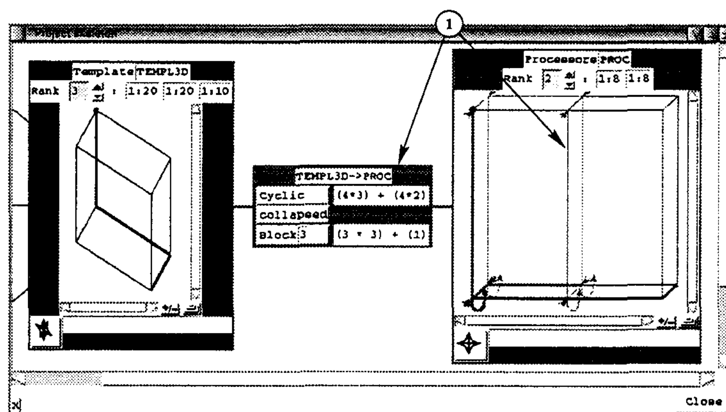


FIG. 3.13: *Niveau 3 de la visualisation (suite).*

Un autre exemple de visualisation de placement est présenté dans la figure 3.13. Il s'agit cette fois de la distribution d'un tableau 3D sur un `processors`.

Dans la sélection de droite, on voit la projection du `template` selon la sélection (1).

Si on suit le bord épais de la vue en fil de fer :

- on commence par une ligne terminée par une boucle signalant une distribution cyclique de la première dimension,

2. [NDLA] En plus, je suis un peu daltonien !

- puis une ligne encadrée par deux flèches en étau pour signaler un écrasement de la deuxième dimension,

- puis un pointillé, signalant une distribution de type `block(`n`)` pour la troisième dimension.

Dans la sélection centrale, la colonne de gauche rappelle ces spécifications, tandis que celle de droite donne des indications plus précises sur la charge du `processors`.

Ainsi, dans la première ligne, la formule `(4*3) + (4*2)` exprime le fait qu'il y a 4 colonnes du `processors` qui contiennent 3 « tranches » du `template`, tandis que les 4 suivantes en contiennent 2 chacune. Ceci traduit le fait que la distribution cyclique effectue 2 fois le tour de la cible, suivi de la moitié d'un troisième tour.

De même, la dernière ligne indique que la distribution par blocs de 3 entraîne le placement de 3 blocs de 3, suivi d'un bloc de 1.

Ce module est donc le point central de la visualisation et de l'édition des directives : À partir des sous-fenêtres, le programmeur peut obtenir et modifier les spécifications précises de chaque directive. La vue en fil de fer permet de donner une idée visuelle d'un placement. De plus, l'intégration de ces sous-fenêtres dans le squelette général permet de garder la vue générale de l'organisation des directives en même temps que leur spécifications.

La réunion de ces trois points dans une même partie de l'interface graphique permet donc à l'utilisateur d'obtenir une visualisation de l'ensemble du placement.

Le module `zoom`

Le troisième niveau permet de donner à l'utilisateur une idée du placement qu'il a défini, mais ne permet pas d'observer finement le placement des éléments d'un objet donné.

Le dernier niveau, implémenté par le module `zoom`, a pour but de suivre exactement la position d'un élément donné. Non seulement on veut savoir où l'élément est placé sur l'objet suivant dans la chaîne de directives, mais aussi sur les maillons suivants, jusqu'au `processors` final.

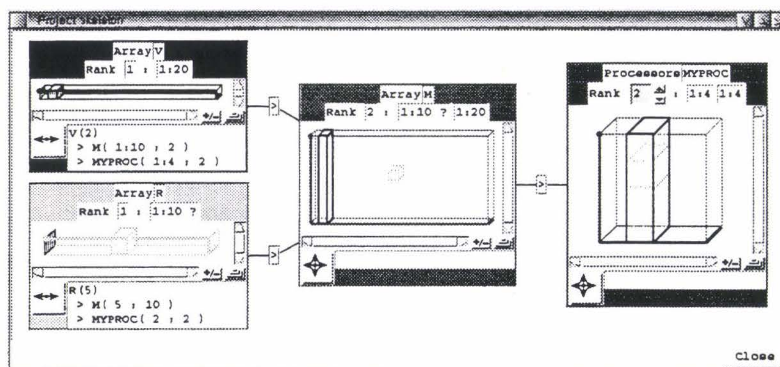


FIG. 3.14: *Niveau 4 de la visualisation.*

Vue des projections : Ainsi, dans la figure 3.14, un curseur est dessiné dans les deux sélections de gauche. Chacun d'eux est dessiné dans la couleur de sa sélection et sa projection est affichée dans les autres sélections.

En plus du dessin de cette projection, la version textuelle est affichée dans le bas de la sous-fenêtre.

On voit, par exemple, que `V(2)` est projeté sur toute la deuxième colonne de `M` (du fait de la réplication) et que cette colonne est à son tour projetée sur la deuxième colonne de `MyProc`. De même, `R(5)` est projeté sur `M(5,10)`, puis sur `MyProc(2,2)`.

Il apparaît donc que `V(2)` et `R(5)` sont tous les deux présents sur le même processeur que `M(5,10)` : `MyProc(2,2)`. Une opération du type de l'instruction suivante n'implique donc pas de communication :

```
R(5)= R(5) + V(2)*M(5,10)
```

Vue des projetés : Pour pouvoir comparer la localité des données, il est également intéressant de disposer directement de la liste des objets projetés sur un élément de **processors** donné. Pour cela, le mécanisme de zoom est également réversible, comme le montre la figure 3.15.

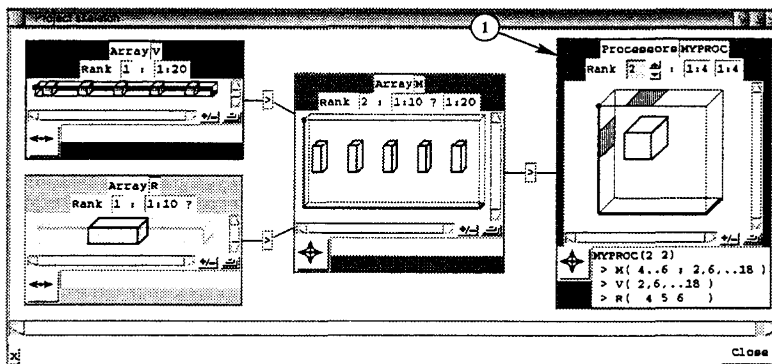


FIG. 3.15: *Niveau 4 de la visualisation (suite).*

Si on place le curseur de la sélection (1) à la position `MyProc(2,2)`, on voit toute la série de blocs de `M` qui s'y projettent, ainsi que les éléments de `V` et `R` qui y sont eux-même alignés. On y retrouve notamment `V(2)`, `R(5)` et `M(5,10)`, mais de façon plus directe, puisqu'ils sont tous les trois indiqués dans le texte en bas de la sous-fenêtre.

Ainsi, la projection permet de vérifier la position relative de certaines données et si elle n'est pas satisfaisante, « l'anti-projection » permet de retrouver rapidement les éventuels décalages. L'utilisateur possède alors les informations nécessaires pour corriger ses alignements.

Édition

L'insertion, la destruction et l'édition de directives ne sont pas liées à un module particulier, mais interviennent à tous les niveaux. Un ensemble de modules met en œuvre la création

de nouvelles directives, ou de nouvelles spécifications à greffer lors des modifications, ainsi que les mises à jour dans les autres directives (par exemple, quand on change le nom d'un `template`, toutes les directives où il est cité doivent être modifiées).

Pour créer un nouveau nœud, un menu peut être affiché en cliquant le nom d'un contexte dans le source ou dans l'arbre. Dans ce menu, deux entrées permettent la création d'un `template` ou d'un `processors`. La zone des déclarations de ce contexte est alors mise en valeur et l'utilisateur n'a qu'à cliquer la ligne où il veut placer la déclaration pour qu'elle soit insérée automatiquement.

De même, un *drag and drop* entre deux références d'objets (icône du *skeleton*, déclaration dans le source ou entrée dans l'arbre) entraîne la création d'une directive de placement entre ces objets. Le type des deux objets et les contraintes d'HPF permettent de décider des types de directives qu'il est possible de créer. S'il reste plusieurs possibilités, un menu demande à l'utilisateur de préciser son choix. Ensuite, les contextes où sont visibles les deux objets permettent de décider du contexte dans lequel cette directive peut être placée. S'il s'agit d'une directive dynamique, l'utilisateur n'a plus qu'à choisir sa position dans ce contexte.

Chaque création est effectuée avec des spécifications par défaut. L'utilisateur peut ensuite modifier chaque caractéristique des directives (nom et taille des objets, spécification d'alignement et de distribution) à partir des sous-fenêtres décrites page 85.

Toutes ces opérations sont illustrées de façon plus détaillée dans la démonstration sur CD-ROM.

Gestion des données dynamiques

Pour pouvoir visualiser les objets et leur placement, l'interface graphique doit récupérer des informations numériques. Si un tableau a une taille définie par une variable ou un argument, il est impossible de connaître sa valeur sans exécuter le programme.

On en voit un exemple dans la figure 3.9 page 83 : le vecteur `R` a une longueur définie par l'argument `NLine`.

Pour contourner ce problème, le serveur d'informations essaie d'évaluer numériquement toutes les expressions symboliques incluses dans les déclarations. Si l'une d'elle contient une référence à une valeur qui ne peut être déterminée qu'à l'exécution, l'analyseur lui fixe une valeur de 10 par défaut.

Ainsi, l'interface graphique ne reçoit que des valeur numériques. Cependant, lorsque le serveur retourne une expression évaluée avec des valeurs par défaut, il le signale. L'interface peut alors le signaler en faisant suivre la valeur d'un « ? ». Pour l'exemple du vecteur `R`, on voit dans la figure 3.12 page 86, qu'une valeur de 10 a été prise par défaut pour la variable et donc pour la taille du vecteur.

Les valeurs des variables et des spécifications de tableaux fixées par les arguments d'une procédure (type `deferred`) peuvent être modifiées par l'utilisateur à partir de l'interface graphique. Ces modifications sont automatiquement répercutées dans les évaluations d'expressions afin de mettre à jour la visualisation.

L'utilisateur peut ainsi suivre l'évolution d'un placement, par exemple au cours des itérations d'une boucle ou des appels successifs d'une fonction.

De même, on a vu dans les chapitres précédents qu'on est quasiment toujours obligé de définir les processors à partir d'un appel à une fonction retournant le nombre de processeurs physiques. Pour pouvoir travailler avec de telles déclarations, le serveur d'informations évalue automatiquement les appels à ces fonctions en utilisant un fichier de description de la machine cible.

Pour rendre cet aspect plus souple, il faudrait que cette description puisse être définie et modifiée pendant l'utilisation d'HPF-BUILDER. Ainsi l'utilisateur pourrait visualiser ses placements, relativement à différentes machines cibles. Pour l'instant, cette fonctionnalité n'est pas encore implémentée dans le prototype.

Limitations de la visualisation

Nous venons de voir comment visualiser des directives HPF de façon graphique. Cependant, certaines situations sont assez difficiles à exprimer graphiquement sans obtenir une visualisation surchargée. C'est notamment le cas quand on a un grand nombre d'objets, ou que certains d'entre eux ont plus de trois dimensions.

Nous présentons ici quelques idées permettant de contourner ces problèmes. Cependant, ces solutions n'ont pas encore été implémentées dans le prototype.

Ainsi, les tableaux FORTRAN peuvent comporter beaucoup plus de trois dimensions. Dans ce cas, la visualisation en fil de fer n'est plus possible.

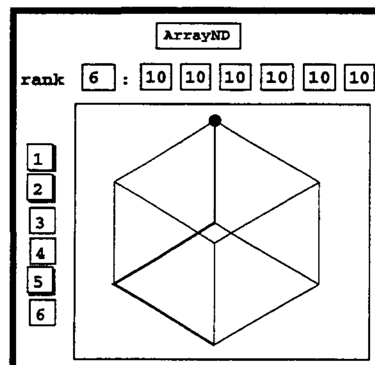


FIG. 3.16: *Visualisation au-delà de 3 dimensions.*

Pour contourner ce problème, on peut remarquer qu'en général, une structure ayant plus de trois dimensions représente généralement une structure à 2 ou 3 dimensions, dont chaque élément est lui-même un tableau. Dans ce cas, l'algorithme n'effectue de balayages de la structure que sur quelques dimensions à la fois et il n'est donc pas nécessaire d'avoir un placement complexe sur chaque dimension.

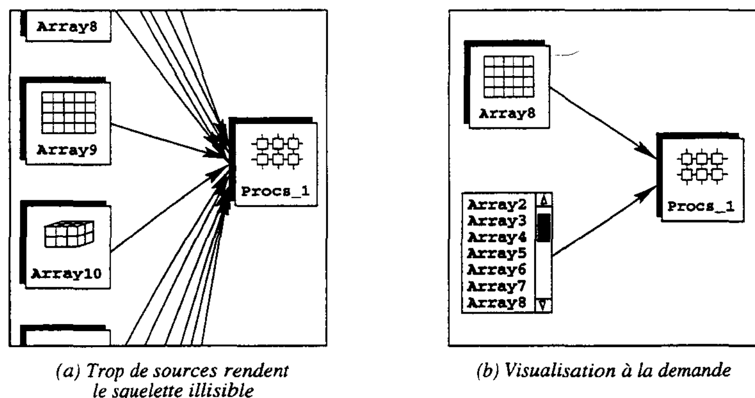
De plus, il semble que l'imagination ne soit pas capable de se représenter une notion de structure à plus de 3 dimensions et qu'il est donc inutile de chercher à la visualiser.

L'utilisateur peut donc se contenter de la vue de 2 ou 3 dimensions à la fois. Une liste de boutons peut permettre à l'utilisateur de sélectionner les dimensions qu'il veut visualiser, comme le montre la figure 3.16.

Un autre problème vient du nombre de tableaux que peut comporter un programme. Certains algorithmes nécessitent une grande quantité de tableaux qui interagissent et doivent donc être tous alignés sur la même destination.

Dans ce cas, le squelette devient illisible, car il comporte une longue liste de nœuds qui rejoignent la même cible (voir figure 3.17(a)).

Cet aspect « sac de nœuds » peut être résolu en permettant à l'utilisateur de n'afficher qu'une partie du squelette.



(a) Trop de sources rendent le squelette illisible

(b) Visualisation à la demande

FIG. 3.17: *Visualisation d'objets très nombreux.*

Nous avons vu que les objets qui ne font partie d'aucune directive ne sont pas affichés. De même, on peut n'afficher que les objets explicitement sélectionnés et n'afficher leurs voisins qu'à la demande, comme on le voit sur la figure 3.17(b).

Conclusion

L'interface graphique que nous venons de présenter met donc en place les différents points demandés dans le cahier des charges décrit en introduction.

Nous n'allons pas détailler plus précisément le fonctionnement de cette interface, ni son utilisation.

Le « mode d'emploi » est plus clairement montré dans la démonstration. Nous conseillons donc au lecteur de se reporter au dernier chapitre, qui présente le CD-ROM de démonstration joint au manuscrit.

3.3.3 Le serveur d'informations

Pour effectuer la visualisation des structures HPF d'un programme, nous avons vu au début de ce chapitre qu'un serveur d'information est utilisé.

Ce serveur fait office d'intermédiaire entre les programmes sources et l'interface graphique.

Pour définir ce serveur, il est nécessaire de définir l'ensemble des requêtes qui peuvent lui être adressées. Pour cela, remettons-nous rapidement en mémoire les informations que les

différents modules de l'interface graphique doivent pouvoir obtenir :

- Dès le premier niveau de visualisation, il faut pouvoir récupérer l'arborescence des contextes et de toutes les déclarations qu'ils contiennent.
- Inversement, il faut pouvoir mettre en valeur des éléments dans l'éditeur.
- Pour le deuxième niveau, il faut savoir reconnaître la source et la cible de chaque directive.
- Pour le niveau suivant, il faut également les spécifications précises des objets.
- Enfin, pour le dernier niveau, il faut pouvoir calculer la projection d'un élément à travers une suite de directives et effectuer l'opération inverse.
- Les expressions symboliques doivent être évaluées sous une forme numérique afin de pouvoir être manipulée par l'interface graphique.
- Pour cela, le serveur doit utiliser des valeurs par défauts. Il doit donc être possible de fixer ces valeurs.

De plus, l'interface graphique permet de modifier des spécifications et de créer de nouvelles directives. Il faut donc pouvoir :

- Créer/détruire/modifier une spécification de directive ou une directive complète.
- Mettre à jour le source, suite à ces changements.
- Changer la valeur par défaut d'une variable et réévaluer les expressions qui en dépendent
- Vérifier la validité des directives (cohérence entre nombres de dimensions, dépassement de bornes ...)

On trouve donc quatre familles d'actions à mettre en place :

- les interactions avec l'éditeur (mise en valeur, modifications ...),
- récupération d'informations dans le source (liste des contextes et déclarations, spécifications ...),
- création/modification de ces informations
- calcul de projections et vérification de cohérence.

Enfin, il faut rappeler que cette liste est ouverte, puisque le serveur doit être extensible à d'autres fonctionnalités, pour pouvoir l'utiliser à partir d'autres outils.

L'étude de cette liste de « travaux à faire » a mené à une organisation autour d'un analyseur syntaxique (le module `Parser`) générant un arbre syntaxique complet du projet (la structure `AST`). À partir de cet arbre syntaxique, une autre structure est générée (la structure `CTX`) par un analyseur (le module `Analyser`). Celle-ci résume les informations que l'interface graphique demande.

Enfin, les évaluations de projections et de cohérence sont gérés par un module à part (le module `Proj`) qui manipule ses propres structures internes (les **ENUMS**).

On arrive donc à l'architecture montrée dans la figure 3.18

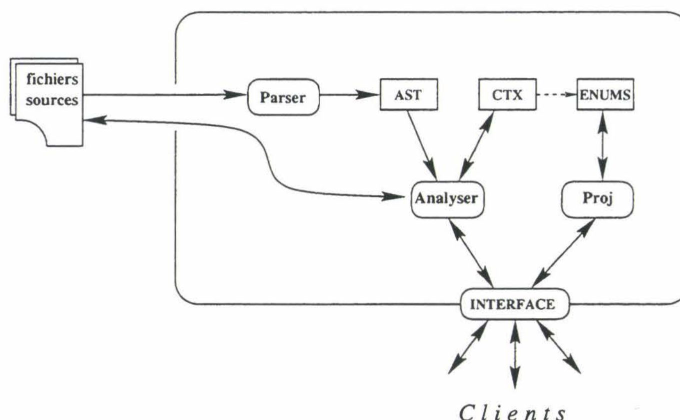


FIG. 3.18: *Organisation du serveur d'informations.*

3.3.4 Le module d'analyse de source

Le module `Parser` a pour but de lire une suite de fichiers source FORTRAN ou HPF, d'en effectuer les analyses lexicale et syntaxique et d'en déduire un arbre syntaxique complet, l'`ast`[3].

Pour construire ce *parser*, plusieurs solutions ont été envisagées :

Réécrire un *parser* de toutes pièces n'apporte pas grand chose par rapport à des outils existants. De plus, c'est un travail fastidieux, d'autant plus que la syntaxe de FORTRAN est assez délicate à analyser.

Au moment où le développement du projet a commencé, l'université de l'Indiana avait déjà publié ses premiers résultats sur **SAGE++** [Sag, BBG+94], une bibliothèque d'analyse syntaxique pour C, C++, FORTRAN, HPF et leur langage PC++. Cependant, à cette époque, leur produit n'en était qu'à ses débuts, et l'utilisation de cette librairie était rendue quasiment impossible du fait du nombre de fonctionnalités *not yet implemented* !

Le *parser* écrit par l'équipe de Thomas BRANDES pour développer le compilateur **ADAPTOR** [Bra96] se révéla beaucoup plus intéressant. Il utilise **COCKTAIL** [Coc], une bibliothèque d'outils de développement qui permet non seulement d'effectuer les analyses lexicale et syntaxique, à la façon des outils `lex` et `yacc`, mais qui contient aussi un outil de description de structures d'arbre et un langage de manipulation de ces arbres.

Il est ainsi possible de décrire un `ast`, de le consulter et de le manipuler de façon très souple.

Cet ensemble étant librement disponible, c'est ce *parser* qui a été choisi pour développer notre module. Le module `Parser` est donc constitué principalement de la partie du compilateur

3. *abstract syntactic tree*, arbre syntaxique abstrait

ADAPTOR dédiée à l'analyse du source.

Depuis, d'autres outils sont apparus, tels que le HPF FRONT END de l'université de Syracuse[HPFc], le *parser* de F LANGUAGE[F l] et FORTRAN2HTML[Sev]. Le travail avec COCKTAIL étant alors déjà bien avancé et semblant apporter toutes les fonctionnalités nécessaires, nous avons préféré le poursuivre que le reprendre à zéro. Ces outils n'ont donc pas été étudiés.

3.3.5 Le module `Analyser` et la structure `context`

Une fois l'`ast` créé, il est nécessaire d'être capable d'y récupérer les informations relatives aux directives HPF et aux différentes déclarations.

Pour cela, le module `Analyser` effectue un parcours de l'`ast` et construit en conséquence une autre structure de données, nommée `context`, qui rassemble toutes les informations présentes dans le source et nécessaires à HPF-BUILDER.

Le *parser* ne sait lire qu'un fichier source à la fois, c'est pourquoi il est nécessaire de disposer d'une structure englobante rassemblant tous les `ast` d'une série de fichiers. De plus, cette méthode permet de rassembler toutes les déclarations et directives sous une forme unifiée alors que l'`ast` contient différentes formes de déclarations, selon la syntaxe utilisée dans le source (dimensions spécifiées en dehors de la déclaration ...).

Enfin, cette structure contient des informations supplémentaires telles que les descriptions de projections (voir section suivante) mais aussi des informations pour l'évaluation des expressions.

3.3.6 Le module `Proj`

La visualisation détaillée nécessite un moyen d'évaluation des projections représentant les directives.

Calculer la projection d'un élément d'objet demande un moyen de décrire les domaines de définition des objets et les directives. Il faut également un moyen de calculer l'image d'une position à travers une directive et l'image réciproque d'une position à travers l'inverse d'une directive, que nous nommerons pré-image. Enfin, le résultat d'une image doit pouvoir être utilisé comme point de départ d'une autre projection.

Le modèle proposé pour effectuer ces opérations sera présenté dans la section 3.4. Il est implémenté par le module `Proj` (comme *projections*), qui permet de manipuler des structures de données relatives à ces modélisations de placements, les enums (comme *énumérations*).

Ce module permet :

- la définition de domaines (représentant les tableaux) et d'ensembles de points (représentant des éléments de tableaux);
- la définition de fonctions de projection représentant des directives de placement;
- le calcul de l'image d'un ensemble par une projection, c'est-à-dire l'endroit où est placé un élément de tableau;

- le calcul de la pré-image d'un ensemble par une projection, c'est-à-dire l'ensemble des éléments de tableaux placés sur un même élément d'objet;
- la mise en forme de domaines et de projections pour en faire des représentations en fil de fer;
- la mise en forme d'ensembles de points pour les décrire sous forme lisible dans la visualisation détaillée;
- la détection de directives incohérentes.

Chaque structure est liée à son objet associé, dans la structure `context`.

3.4 Modélisation des placements

Pour visualiser et montrer en détail le placement d'un objet, il est nécessaire de savoir interpréter l'ensemble des directives de placement qui interviennent entre cet objet et son `processors` cible.

En effet, le module `zoom` doit être capable de déterminer à quelle position un élément de tableau est placé selon une directive donnée. Le résultat de cette projection doit lui-même pouvoir être projeté à travers une autre directive.

Inversement, il faut être capable de déterminer quelles parties d'un objet sont placées sur une position donnée de la destination d'une directive. Là encore, le calcul doit pouvoir être effectué à travers plusieurs directives consécutives.

Cette section présente la façon dont nous avons modélisé la notion de placement HPF et comment nous effectuons les calculs nécessaires à chacune de ces opérations.

Exemple

Tout au long de cette section, nous nous référerons à l'exemple décrit dans l'extrait de code suivant :

```
real, dimension(10,15,20) :: A
!HPF$ template T(40,30,20)
!HPF$ align A(i,j,k) with T(2*k,*,i+5)
!HPF$ processors P(4,4)
!HPF$ distribute T(cyclic,*,block) onto P
```

3.4.1 Introduction

Nous avons vu précédemment que les placements en HPF ont un aspect géométrique. Un tableau est un ensemble de positions qu'on peut modéliser par un pavé dans un espace discret multidimensionnel $\mathbb{N}^n$.

Un tel tableau est placé dans un `processors` à la suite d'une série de directives d'alignement et de distribution. Chaque directive correspond à une projection dans un autre objet

(tableau, `template` ou `processors`). Ces autres objets sont eux aussi modélisables par des pavés de $\mathbb{N}^n$.

On notera $\mathcal{D}_x$ le pavé de $\mathbb{N}^n$ décrivant l'objet x et $\mathcal{D}_{x_i}$ l'intervalle couvert par la i^e dimension de x. Le nombre de dimensions de x sera noté R_x.

$$\mathcal{D}_x = \{\vec{I} = \begin{pmatrix} i_1 \\ .. \\ i_{R_x} \end{pmatrix} \mid i_n \in [x_n^{min} : x_n^{max}], \forall n\} = \begin{bmatrix} x_1^{min}:x_1^{max} \\ .. : .. \\ x_{R_x}^{min}:x_{R_x}^{max} \end{bmatrix}$$

Chaque alignement et distribution correspond donc à une application de projection des éléments de la source vers la cible.

Cette application n'est pas du tout bijective : s'il y a réplication, l'image d'un point peut être tout un intervalle et s'il y a écrasement d'une dimension, plusieurs points auront la même image.

Nous auront donc à manipuler des points, mais aussi des ensembles de points. Nous les noterons $\{\vec{I}\}$.

Pour le tableau `A`, aligné sur le `template T`, distribué sur le `processors P`, le placement peut être modélisé comme suit :

Pour une position $\vec{I} = \begin{pmatrix} i_1 \\ .. \\ i_{R_A} \end{pmatrix}$, l'élément $A(\vec{I})$ est aligné sur $T(\{\vec{J}\})$ qui est une section de T. $\{\vec{J}\}$ est une liste de points et constitue l'image de $\vec{I}$ par une application de projection que nous noterons $P^{A\to T}$. De même, chaque élément de $\{\vec{J}\}$ est projeté sur P par $P^{T\to P}$.

Une restriction sur la validité des directives implique que les projections ne doivent pas sortir du domaine cible. Il faut donc vérifier que $\{\vec{J}\}$ est inclus dans $\mathcal{D}_T$.

Exemple

`A`, `T` et `P` sont définis par leurs domaines $\mathcal{D}_A = \begin{bmatrix} 1{:}10 \\ 1{:}15 \\ 1{:}20 \end{bmatrix}$, $\mathcal{D}_T = \begin{bmatrix} 1{:}40 \\ 1{:}30 \\ 1{:}20 \end{bmatrix}$ et $\mathcal{D}_P = \begin{bmatrix} 1{:}4 \\ 1{:}4 \end{bmatrix}$.

On voit qu'un point $A(\vec{I}) = A \begin{pmatrix} i \\ j \\ k \end{pmatrix}$ se projette sur $T(P^{A\to T}(\vec{I})) = T \begin{pmatrix} 2*k \\ \mathcal{D}_{T_2} \\ i+5 \end{pmatrix}$

On peut remarquer que comme l'indice j n'est pas spécifié dans l'alignement, cela signifie que toute la deuxième dimension de `A` est écrasée sur chaque point projeté de `T`.

3.4.2 Modélisation

Si on essaie de modéliser les placements HPF de façon très générale, on arrive à des constructions mathématiques assez lourdes à manipuler. En effet, les enchaînements de di-

rectives impliquent des composées de projections qui donnent comme images des sections de domaines irrégulières.

Cependant, HPF impose certaines restrictions qui simplifient ces constructions.

En effet, le modèle doit permettre de calculer l'image d'une position dans un objet à travers un alignement ou une distribution.

Pour chacune des dimensions, après un alignement, on obtient un point, ou, dans le cas d'une réplication, un intervalle couvrant la totalité de l'objet cible.
De tels intervalles ne peuvent plus être projetés par d'autres alignements, car une telle transformation ne serait pas représentable directement par un seul alignement et entrerait donc en conflit avec les notions d'alignement ultime définies dans [For97, pp20–21]
Par contre, les points peuvent encore être projetés à travers un autre alignement et restent des points ou deviennent des intervalles.

Ensuite, les différents types de distributions sont l'écrasement et les `block` et `cyclic` qui peuvent tous se ramener au cas `cyclic(`k`)`.
La projection d'un point par une telle fonction reste un point et transforme les intervalles en intervalles. Enfin, une fois distribué, un objet ne peut plus être aligné ou distribué.

En conclusion, les structures à manipuler en entrée d'une projection sont des points et des intervalles, comme l'indique la figure 3.19.

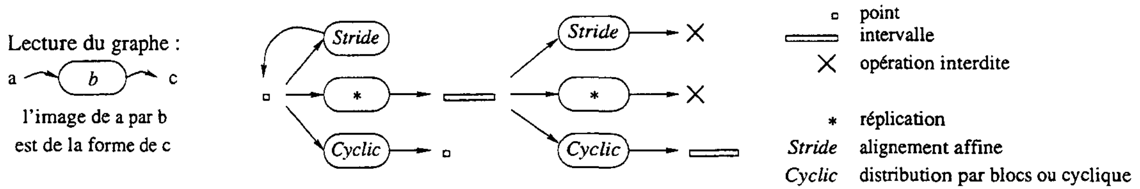


FIG. 3.19: *Ensembles de points manipulés par les calculs d'images.*

Nous allons donc maintenant présenter une modélisation possible des placements HPF, qui prend en compte ces contraintes et simplifications.

Alignements

Nous avons vu que l'alignement de A sur T peut être modélisé par une application de projection d'un point vers un ensemble de points

$$P^{A\to T}: \left| \begin{array}{l} \mathcal{D}_A \to \{\mathcal{D}_T\} \\ \vec{I} \mapsto \{\vec{J}\} \end{array} \right.$$

Cette fonction est caractérisée par les domaines de départ et d'arrivée $\mathcal{D}_A$ et $\mathcal{D}_T$ et par $p^{A\to T}$, un ensemble de sous-fonctions donnant l'image des indices.

Chaque sous-fonction $p_i^{A\to T}$ prend la valeur d'un des indices en entrée et la projette selon une des formes d'alignement. HPF accepte des alignements sous forme de fonction affine (un *stride*), de réplication et d'effondrement d'une dimension:

– On a vu précédemment que l'effondrement se traduit par l'absence de référence à un

des indices.

- La réplication transforme toute position en un intervalle parcourant toute la dimension d'arrivée. Cette projection sera notée $*$.
- Enfin, le *stride* transforme un indice i en une position $ai+b$. Cette projection sera notée $a \times (i) + b$.

Exemple

La fonction $p^{A \to T}$ peut être noté sous la forme

$$\begin{pmatrix} 2 \times (3) + 0 \\ * \\ 1 \times (1) + 5 \end{pmatrix}$$

ce qui signifie que la première dimension contient 2 fois le 3e indice plus 0, la 2e dimension contient une réplication et la 3e dimension contient le premier indice plus 5

On obtient donc la définition suivante :

$$P^{A \to T} = (\mathcal{D}_A, \mathcal{D}_T, p^{A \to T}) = \left(\mathcal{D}_A, \mathcal{D}_T, \begin{pmatrix} 2 \times (3) + 0 \\ * \\ 1 \times (1) + 5 \end{pmatrix} \right)$$

Distributions

De la même façon, la distribution de T sur P peut se modéliser par une fonction $P^{T \to P}$. On a vu précédemment que toutes les formes de distributions autres que l'effondrement peuvent se ramener à la forme `cyclic(`k`)`.

L'image d'un indice i_m d'une dimension de T parcourant $[T_m^{min} : T_m^{max}]$ vers la dimension n de P est donc

$$p_m^{T \to P}(i_m) = ((i_m - T_m^{min}) \div k) \mod \delta_{P_n} + P_n^{min}$$

où δ_{P_n} représente la taille de la dimension n de P, c'est-à-dire $P_n^{max} - P_n^{min} + 1$ (l'opérateur $a \div b$ représente $\lfloor \frac{a}{b} \rfloor$, c'est-à-dire la division entière de a par b).

Il suffit donc d'ajouter au modèle précédent un nouveau type de description de sous-fonction $(i)[k]$ où i est le numéro de la dimension source concernée et k la taille de bloc.

Exemple

$p^{T \to P}$ peut être noté sous la forme

$$\begin{pmatrix} (1)[1] \\ (3)[5] \end{pmatrix}$$

ce qui signifie que la première dimension de T est projetée cycliquement sur la première dimension de P et la troisième est projetée selon une distribution $cyclic(\lceil \frac{\mathcal{D}_{T_3}}{\mathcal{D}_{P_2}} \rceil)$, c'est-à-dire par blocs.

Cas général

Dans le cas général, le placement d'un objet X sur un objet Y est donc représentable par une fonction

$$P^{X\to Y} = \left(\mathcal{D}_X, \mathcal{D}_Y, \begin{pmatrix} p_1^{X\to Y} \\ p_2^{X\to Y} \\ .. \\ p_{R_Y}^{X\to Y} \end{pmatrix} \right)$$

avec $p_j^{X\to Y}$ de la forme $a \times (i) + b$, $*$ ou $(i)[k]$.

Cette modélisation permet donc de calculer l'ensemble des points où est projeté un élément d'objet à travers une directive de placement donnée.

Composées de projections

La visualisation oblige à calculer non seulement des images de points, mais aussi des composées de projections. Il faut donc savoir calculer l'image du résultat d'une première projection, c'est-à-dire d'un point ou d'un ensemble de points.

Le modèle précédent peut être immédiatement étendu de façon à accepter des ensembles en entrée.

La forme de ces ensembles est régulière : on commence par demander l'image d'un point. Après projection dans un alignement, certaines dimensions peuvent devenir des intervalles couvrant la totalité de l'objet destination (par une réplication).

Ensuite, une étape de distribution laisse les points en points et transforme les intervalles en intervalles. En effet, un intervalle parcourt toute la largeur de l'objet source. Son image couvre donc toute une « tranche » du `processors`, jusqu'à l'image du dernier point de l'intervalle, ou, si on a bouclé dans le `processors`, jusqu'au bout de celui-ci. Cette image est donc de la forme $[1 : u]$ ou $\mathcal{D}_{P_i}$.

De façon plus formelle, l'intervalle de départ est de la forme $[T_m^{min} : T_m^{max}]$. Les points i_m de cet intervalle se projettent en $((i_m - T_m^{min}) \div k) \mod \delta_{P_n} + P_n^{min}$.

La partie $(i - T_m^{min}) \div k$ de l'expression transforme l'intervalle en

$$\left[(T_m^{min} - T_m^{min}) \div k \,:\, (T_m^{max} - T_m^{min}) \div k\right] = \left[0 \,:\, (\delta_{T_m} - 1) \div k\right]$$

Si $(\delta_{T_m} - 1) \div k < \delta_{P_n}$ le modulo n'a pas d'effet, sinon, l'intervalle parcourt tout $[0 : \delta_{P_n} - 1]$. Dans les deux cas, on obtient donc l'intervalle

$$\left[P_n^{min} \,:\, \min(P_n^{max}, ((\delta_{T_m} - 1) \div k) + P_n^{min})\right]$$

Résumé des calculs d'images

En résumé, les calculs de projection ne nécessitent que la manipulation de points et d'intervalles dans chaque dimension des objets concernés (comme on l'a vu dans la figure 3.19 page 98). On va de la dimension m de X à la dimension n de Y par les calculs suivants :

	$a \times (m) + b$	$*$	$(m)[k]$
i_m	$ai_m + b$	$[X_m^{min} : X_m^{max}]$	$((i_m - X_m^{min}) \div k) \mod \delta_{Y_n} + Y_n^{min}$
$\mathcal{D}_{X_m}$	interdit	interdit	$[Y_n^{min} :$ $\min(Y_n^{max}, ((\delta_{X_m} - 1) \div k + Y_n^{min} - 1)]$

Ces calculs permettent donc de mettre en place le zoom dans le sens des projections. Il faut maintenant voir comment effectuer les calculs inverses.

3.4.3 Calcul des pré-images

La visualisation nécessite également de pouvoir déterminer quelle section d'une source X est projetée sur un élément donné d'une cible Y.

Ceci équivaut à calculer l'image d'un point par la fonction inverse d'une fonction de projection. Or, on peut obtenir par cette fonction inverse des ensembles plus complexes qu'auparavant (voir figure 3.20) :

- la pré-image d'un *stride* est un point, ou est vide;
- la pré-image d'une réplication fait disparaître une dimension, comme dans le cas de l'image d'un effondrement;
- au contraire, à la suite d'un effondrement la pré-image peut faire apparaître des intervalles parcourant toute une dimension de l'objet source;
- enfin, la pré-image d'un *cyclic* est beaucoup moins évidente : il s'agit d'une suite d'intervalles de taille k commençant tous les $k\delta_{Y_n}$ bornée par $\mathcal{D}_{X_m}$.

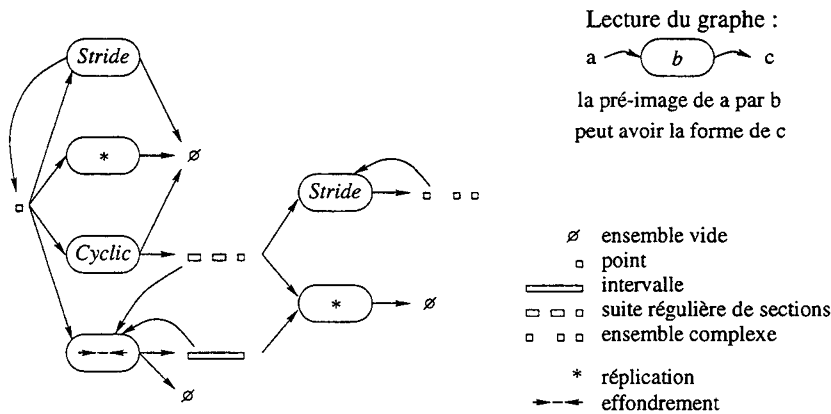


FIG. 3.20: *Ensembles de points manipulés par les calculs de pré-images.*

Les composées de projections entraînent différentes formes de calcul des pré-images :

- Nous venons de voir que la pré-image d'un *stride* donne un point ou est vide. Ce résultat peut donc être repris directement pour un calcul de pré-image au niveau précédent.

- Dans le cas où on obtient un intervalle (pré-image d'un effondrement), il couvre tout l'ensemble d'arrivée. La pré-image d'un tel ensemble est donc également un intervalle couvrant toute la source.
- L'ensemble résultant de la pré-image d'un *cyclic* est plus complexe et sa pré-image par un *stride* donne un ensemble irrégulier qui sera décrit dans la dernière section.

On note j_n la coordonnée dans la n^{e} dimension d'un point j de Y. La pré-image de j_n est notée $p_n^{X \leftarrow Y}(j_n)$.

Pré-image d'un *stride*

Dans le cas d'un *stride*, la pré-image d'une coordonnée j_n de Y_n est issue d'une fonction de projection dont la m^{e} spécification était de la forme $a \times (n) + b$

La pré-image est donc l'ensemble des coordonnées de X_m dont l'image par $p_m^{X \rightarrow Y}$ est j_n.

$$\begin{aligned} p_n^{X \leftarrow Y}(j_n) &= \{i_m \mid p_m^{X \rightarrow Y}(i_m) = j_n\} \cap \mathcal{D}_{X_m} \\ &= \{i_m \mid a i_m + b = j_n\} \cap \mathcal{D}_{X_m} \\ &= \{i_m \mid a i_m = j_n - b\} \cap \mathcal{D}_{X_m} \end{aligned}$$

Donc, si $j_n - b$ est divisible par a et que $(j_n - b)/a$ est dans $\mathcal{D}_{X_m}$, on obtient $p_n^{X \leftarrow Y}(j_n) = \{(j_n - b) \div a\}$ sinon, $p_n^{X \leftarrow Y}(j_n) = \emptyset$.

Cette méthode donne donc un moyen de calculer une pré-image à travers un *stride* et, à partir de là, à travers tout alignement.

Exemple

Le calcul de la pré-image de `T(2,3,4)` correspond au calcul de $p_1^{A \leftarrow T}(2)$, $p_2^{A \leftarrow T}(3)$ et $p_3^{A \leftarrow T}(4)$. $p_1^{A \leftarrow T}(2)$ implique le calcul de la pré-image de 2 par $2 \times (3) + 0$, c'est-à-dire

$$p_1^{A \leftarrow T}(2) = \{k \mid 2k = 2 - 0\} \cap [1:20]$$

2 est divisible par 2 et $2 \div 2 = 1$ est dans $[1:10]$. Donc $p_1^{A \leftarrow T}(2) = 1$.

De même $p_3^{A \leftarrow T}(4) = \{i \mid i = 4 - 5\} \cap [1:10]$. Cette fois, la solution $i = -1$ n'est pas dans l'intervalle, donc $p_3^{A \leftarrow T}(4) = \emptyset$ et de ce fait, la pré-image de `T(2,3,4)` est vide.

Par contre, si on cherche la pré-image de `T(2,3,6)`, on obtient $i = 1$. On a toujours $k = 1$ et j couvre tout $\mathcal{D}_{T_2}$.

Donc, la pré-image de `T(2,3,6)` est `A(1,1..15,1)`

Pré-image d'un *cyclic*

Dans le cas d'un *cyclic*, j_n est issue d'une fonction de projection dont la m^{e} spécification était de la forme $(n)[k]$:

$$\begin{aligned} p_n^{X \leftarrow Y}(j_n) &= \{i_m \mid p_m^{X \rightarrow Y}(i_m) = j_n\} \cap \mathcal{D}_{X_m} \\ &= \{i_m \mid ((i_m - X_m^{min}) \div k) \mod \delta_{Y_n} + Y_n^{min} = j_n\} \cap \mathcal{D}_{X_m} \end{aligned}$$

Or, cette équation peut être transformée sous la forme suivante :

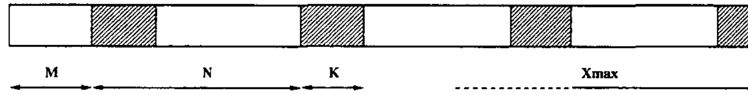
$$
\begin{aligned}
&((i_m - X_m^{min}) \div k) \mod \delta_{Y_n} = j_n - Y_n^{min} \\
\Leftrightarrow &(i_m - X_m^{min}) \div k) = j_n - Y_n^{min} + \alpha\delta_{Y_n} \\
\Leftrightarrow &(i_m - X_m^{min}) = k(j_n - Y_n^{min} + \alpha\delta_{Y_n}) + \beta \text{ avec } 0 \leq \beta < k \\
\Leftrightarrow &i_m = \alpha k \delta_{Y_n} + \beta + X_m^{min} + kj_n - kY_n^{min}
\end{aligned}
$$

On obtient donc la forme générale

$$p_n^{X \leftarrow Y}(j_n) = \{i_m \mid i_m = \alpha N + \beta + M\} \cap \mathcal{D}_{X_m}$$

avec $N = k\delta_{Y_n}, M = kj_n + X_m^{min} - kY_n^{min}, 0 \leq \alpha < \lceil \frac{\delta_{X_m}}{k\delta_{Y_n}} \rceil, 0 \leq \beta < k$ (α est borné par ces valeurs, car au-delà, aucune valeur de β ne permet d'obtenir de résultat compris dans $\mathcal{D}_{X_m}$).

Un tel ensemble a la forme d'une suite régulière d'intervalles de même taille, comme le montre le schéma suivant :



Comme toutes les distributions autres que les effondrements se ramènent à des *cyclic* et que l'effondrement est déjà traité, une pré-image par une distribution peut être calculée.

Exemple

La pré-image dans T de P(2,3) correspond au calcul des fonctions $p_1^{T \leftarrow P}(2)$ et $p_2^{T \leftarrow P}(3)$.

La première implique le calcul de la pré-image de 2 par (1)[1], c'est-à-dire

$$p_1^{T \leftarrow P}(2) = \{i \mid i = \alpha N + \beta + M\} \cap [1:40]$$

avec $N = 1\delta_{P_1} = 4$ et $M = 1 \times 2 + 1 - 1 \times 1 = 2$. β est bloqué à zéro et α va de 0 à $\lceil \frac{40}{1 \times 4} \rceil = 10$ non compris.

On obtient donc l'ensemble $\{2, 6, 10, 14, ..38\}$.

De la même manière,

$$p_2^{T \leftarrow P}(3) = \{i \mid i = \alpha N + \beta + M\} \cap [1:40],$$

avec cette fois $N = 5\delta_{P_1} = 20$ et $M = 5 \times 3 + 1 - 5 \times 1 = 11$. On a β compris entre 0 et 5 exclu et α entre 0 et $\lceil \frac{20}{5 \times 4} \rceil = 1$ exclu, c'est-à-dire que α est bloqué à zéro.

On obtient donc l'ensemble $\{11, 12, 13, 14, 15\}$.

En conclusion, la pré-image de P(2,3) est donc l'ensemble $\begin{pmatrix} 2, 6, 10, 14, ..38 \\ 1..30 \\ 11, 12, 13, 14, 15 \end{pmatrix}$.

Pré-image d'une composée *stride* ∘ *cyclic*

Cette fois, il s'agit de trouver les points h_l de la source W_l dont une image i_m dans X_m est incluse dans la pré-image de j_n par la projection de X_m vers Y_n :

$$\begin{aligned} p_n^{W \leftarrow Y}(j_n) &= p_l^{W \leftarrow X}(p_m^{X \leftarrow Y}(j_n)) \\ &= \{h_l \mid \exists i_m \in \mathcal{D}_{X_m} \text{ avec } p_l^{W \rightarrow X}(h_l) = i_m \text{ et } p_m^{X \rightarrow Y}(i_m) = j_n\} \\ &= \{h_l \mid \exists i_m \in \mathcal{D}_{X_m} \text{ avec } ah_l + b = i_m \text{ et } i_m = \alpha N + \beta + M\} \end{aligned}$$

On est donc ramené à la résolution de l'équation $ah_l = \alpha N + \beta + M - b$.

Or, pour chaque valeur de α, l'équation admet des solutions $\frac{\alpha N + \beta + M - b}{a}$ pour les valeurs de β où le numérateur est divisible par a, c'est-à-dire quand β est de la forme :

$$a - ((\alpha N + M - b) \mod a) + na$$

On peut donc énumérer les éléments de $p_l^{W \leftarrow Y}(j_n)$ en énumérant les valeurs de α de 0 à $\lceil \frac{\delta_{X_m}}{k\delta_{Y_n}} \rceil$ et, pour chacune, en énumérant les valeurs de β, de $|a| - (\alpha N + M - b) \mod |a|$ à k, par pas de $|a|$.

La forme générale d'un tel ensemble correspond à la répétition régulière d'un motif dont la taille correspond au PPCM de a et $k\delta_{Y_n}$, comme le montre l'exemple de la figure 3.21.

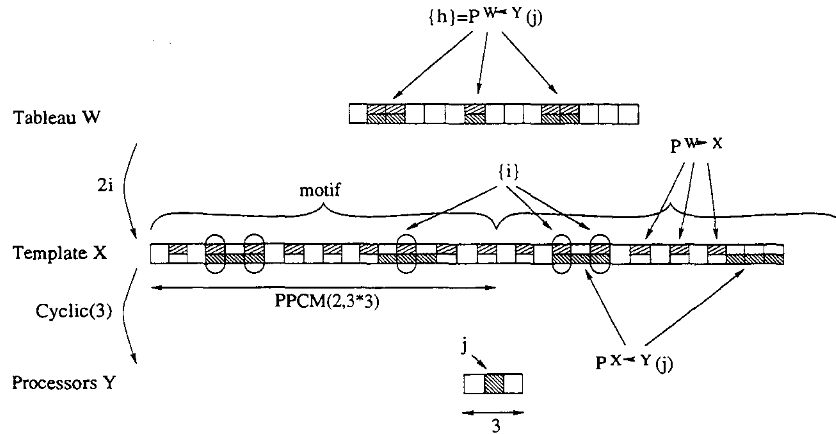


FIG. 3.21: *Exemple de pré-image stride ∘ cyclic.*

C'est une forme difficile à manipuler directement, mais que l'on sait énumérer. Il est donc possible de l'utiliser pour obtenir la liste des éléments projetés par un *stride* suivi d'un *cyclic*.

Exemple

La pré-image dans `A` de `P(2,3)` correspond, pour la première dimension, à $p_1^{A \leftarrow T}(p_1^{T \leftarrow P}(2))$, c'est-à-dire la pré-image de 2 à travers $2 \times (3) + 0 \circ [1](1)$

Il faut donc énumérer les $\frac{\alpha N + \beta + M - b}{a}$, avec $N = 4$ et $M = 2$ pour α de 0 à 9 et β bloqué à zéro.

L'ensemble obtenu est donc $\{2, 6, 10, ..38\}$. Chacune de ces valeurs est un indice j dans T où un élément de A est projeté par $2 \times (3) + 0$. Comme on sait qu'un élément de A est présent à cet indice, on sait qu'on peut calculer $(j - 0) \div 2$.

On obtient donc finalement l'ensemble d'indices $\{1, 3, 5, ..19\}$.

Le cas particulier où $a = 0$ est différent. Il correspond à un effondrement d'une dimension sur un élément de l'intervalle. Il s'agit alors de savoir si cet élément est compris dans la pré-image du *cyclic*.

Il faut donc déterminer si il existe un couple α, β tel que $\alpha N + \beta + M = b$.

Cette équation n'a de solutions que pour $\alpha = \frac{b-M-\beta}{N}$. Comme on sait que $0 \leq \beta < k \leq N$ (c'est-à-dire que les intervalles successifs sont disjoints), on en déduit que

$$b - M - N < b - M - \beta \leq b - M$$
$$\Leftrightarrow \frac{b - M - N}{N} < \alpha \leq \frac{b - M}{N}$$

C'est-à-dire qu'il ne peut y avoir de solution que pour $\alpha = (b - M) \div N$. On a alors $\beta = b - M - \alpha N = (b - M) \mod N$.

Il suffit de vérifier que α et β sont dans les intervalles imposés pour savoir si la pré-image est vide, ou vaut tout l'intervalle $\mathcal{D}_{W_l}$.

Pour β, il doit être entre zéro inclus et k exclu.

Pour α, on a

$$b - M = b - kj_n - X_m^{min} + kY_n^{min} = b - X_m^{min} - k(j_n - Y_n^{min})$$

Or, une directive valide interdit d'avoir un effondrement en-dehors de la cible, c'est-à-dire qu'on a $b \in \mathcal{D}_{X_m}$. Donc on sait que $0 \leq b - X_m^{min} < \delta_{X_m}$. De plus, $0 \leq j_n - Y_n^{min} < \delta_{Y_n}$, donc $-k\delta_{Y_n} < -k(j_n - Y_n^{min}) \leq 0$, c'est-à-dire que

$$-N < b - M < \delta_{X_m}$$
$$\Rightarrow -1 < \frac{b - M}{N} < \frac{\delta_{X_m}}{N}$$
$$\Rightarrow 0 \leq \alpha < \frac{\delta_{X_m}}{k\delta_{Y_n}}$$

La valeur calculée d'α est donc forcément valide et il suffit de vérifier que celle de β est strictement inférieure à k.

Cette méthode permet donc d'effectuer le calcul dans le cas d'un effondrement sur un objet distribué.

3.4.4 Conclusion

Nous venons donc de voir une modélisation formelle des objets HPF et des directives qui les relient. Cette modélisation permet d'effectuer les calculs de projections d'éléments d'objets et leurs réciproques.

La visualisation détaillée peut donc utiliser cette méthode pour mettre en œuvre l'effet de zoom.

3.5 HPF-Builder : Conclusion

Dans ce chapitre, nous avons mis en place le cahier des charges d'un outil de placement de directives HPF. Cet outil a pour but de venir s'intégrer dans un projet d'environnement graphique d'aide à la programmation en HPF.

On a vu ce qu'un tel outil doit apporter à l'utilisateur pour l'aider dans cette phase du développement qui consiste à ajouter dans le programme source les informations nécessaires au compilateur pour qu'il puisse générer un code efficace.

Un prototype de cet outil a été implémenté sous la forme d'une interface graphique, HPF-BUILDER, liée à un serveur d'informations qui centralise les travaux d'analyse et de mise à jour des sources. Cette implémentation en deux parties permet de venir y greffer d'autres outils d'aide au développement.

L'utilisation d'HPF-BUILDER a été décrite de façon succincte et une démonstration est disponible sur le CD-ROM qui accompagne ce document. Cette démonstration montre l'insertion de directives HPF dans un programme FORTRAN 90, étape par étape, jusqu'à l'obtention d'un placement de données complet.

Il a été vu dans le chapitre précédent qu'un tel outil tirerait avantage d'une évaluation des effets des placements que l'utilisateur définit. Cette évaluation, même approximative, serait surtout intéressante s'il n'est pas nécessaire d'exécuter le code pour la calculer. En effet, sans cela, l'aspect interactif est fortement réduit.

C'est dans ce but qu'un projet d'évaluation des coûts de communications induits par les placements à été initié. C'est ce projet que nous allons présenter dans le chapitre suivant.

Chapitre 4

Pré-évaluation de performances

Problèmes de communications :

Allô, New York?
Passez-moi le 22 à Asnières

Fernand Raynaud

4.1 Introduction

Tout au long des chapitres précédents, nous avons vu que le langage HPF permet de mettre en place le parallélisme de données en FORTRAN.

Nous avons remarqué que les performances d'un programme HPF dépendent fortement du placement des données. En effet, ce placement va influer directement sur la charge de chaque processeur (c'est-à-dire sur l'équilibre dans la répartition des calculs) et sur les communications nécessaires à la réunion des opérandes de chaque instruction (c'est-à-dire sur les délais d'exécution de ces instructions).

En HPF, ce placement est spécifié par l'intermédiaire de directives d'alignement et de distribution. Nous avons présenté l'outil HPF-BUILDER, qui permet au programmeur d'éditer de façon graphique les directives d'un programme HPF.

Cet outil permet de visualiser les placements que le programmeur met en œuvre. Il fournit une vision globale de ces placements, mais aussi un moyen d'observer élément par élément leurs effets sur la localité des données.

Cette vue détaillée des placements permet au programmeur d'évaluer l'efficacité de cette localité : il peut comparer ce qu'il voit avec les instructions de son programme pour en déduire approximativement les coûts en charge de calculs et en communications.

Ce travail d'évaluation est assez difficile. C'est pourquoi il semble intéressant de chercher

à l'automatiser. Il s'agit donc de visualiser non seulement la localité des données, mais aussi leurs conséquences relativement à l'algorithme, c'est-à-dire qu'on va chercher à déterminer le coût d'exécution des instructions du programme, en fonction des placements qui ont été définis.

Nous avons vu qu'il n'existe que très peu d'outils qui soient capables d'évaluer l'influence d'un placement de données. Chronométrer l'exécution du programme ne suffit pas, car cette information est trop imprécise.

Observer les messages générés n'est pas suffisant non plus, puisqu'il est difficile de rapprocher ces messages des instructions HPF correspondantes. De plus, cette méthode nécessite une instrumentation du code qui a elle-même une influence sur les performances.

Enfin, ces deux solutions sont coûteuses, puisqu'elles nécessitent d'exécuter le programme avec des jeux d'essais en grandeur réelle et sur la machine parallèle cible, dont le coût d'utilisation est généralement prohibitif.

Il apparaît donc qu'une méthode d'évaluation des performances *avant compilation* aiderait grandement les programmeurs. Même si une telle évaluation ne peut pas fournir une mesure exacte des performances, il est possible d'évaluer le coût d'un groupe d'instructions selon un modèle approximatif.

Ce chapitre va développer les travaux que nous avons effectués sur ce sujet. Nous allons tout d'abord essayer de cerner les besoins de l'utilisateur, pour en déduire la forme que devrait avoir un tel outil. Ensuite, après avoir exposé les principes d'une telle pré-évaluation, nous présenterons les mises en œuvre possibles qui ont été entreprises [LD98].

4.2 Idée générale

Avant de chercher à savoir comment évaluer les performances d'un programme, il est intéressant de chercher à savoir ce que l'utilisateur attend d'une telle évaluation.

Si on se place dans le cadre de l'utilisation d'un ensemble d'outils tels qu'HPF-BUILDER, on doit chercher à mettre au point une interface graphique à partir de laquelle on peut demander à évaluer les performances d'une partie du code.

Si on part d'un programme très simple, on peut considérer qu'il n'y a qu'à exécuter le code en le chronométrant pour savoir s'il est efficace. Après une modification, il suffit de voir si le temps diminue ou augmente.

Dans le cas général, le problème vient du fait qu'une modification des placements peut améliorer l'efficacité sur une partie du code, mais la réduire sur une autre partie. Il est alors impossible de détailler l'origine des gains ou pertes de performances.

C'est pourquoi il est nécessaire de pouvoir mesurer les performances du programme sur des extraits de code. De plus, on a vu en introduction qu'on voulait éviter d'avoir à exécuter le code.

L'idéal serait donc de disposer d'un outil grâce auquel il suffirait d'un simple clic sur une ligne de code dans l'éditeur de programmes pour obtenir une évaluation des performances sur cette ligne.

Cette évaluation doit prendre en compte le placement courant défini dans le programme.

Ainsi, après une modification de ce placement, un autre clic permettrait de comparer l'efficacité des deux versions.

Déjà à ce niveau, certains problèmes apparaissent. Nous allons les énumérer sous forme de questions auxquelles les sections suivantes vont tenter de répondre :

- Les restrictions dues à HPF limitent le spectre d'informations qu'on peut obtenir : quelles informations sait-on obtenir ?
- L'utilisateur a besoin d'un type d'informations dont il peut tirer parti pour optimiser ses placements : quelles informations doit-on visualiser ?
- Il faut donc déterminer les informations que l'on peut obtenir et qui peuvent aider l'utilisateur : que faut-il afficher ?
- Enfin, la présentation même de ces informations doit être compréhensible et organisée de façon visuelle et intuitive : comment afficher ?
- Une fois les informations à afficher définies, il faut définir l'ensemble des calculs à effectuer pour les obtenir : que doit-on calculer ?
- Enfin, il faut déterminer un moyen d'effectuer ces calculs : comment calculer ?

4.2.1 Quelle information sait-on obtenir ?

Si on cherche à se placer avant la compilation, on ne peut obtenir d'informations qu'à partir du programme source HPF. Or, à ce niveau, on est indépendant du compilateur et de la machine. Il faut donc déterminer quelles informations sont possibles à obtenir à ce stade.

Abstraction de la machine physique et du compilateur

En HPF, le placement physique des données est inconnu avant l'exécution, puisque c'est au compilateur de décider du placement des données à ce niveau [For97, p19,l26].

On ne peut décider du placement qu'au niveau des éléments des `processors`. Ces éléments sont des processeurs abstraits : leur puissance et les caractéristiques des liens de communications entre eux sont considérés homogènes.

Cela signifie que le coût du déplacement d'une donnée est considéré comme invariable, quels que soient les processeurs source et destination.

Il apparaît donc que le coût de communication, au niveau HPF, ne peut se mesurer qu'en volume de déplacements de données. Il n'est pas possible de tenir compte des distances entre les processeurs physiques ou des effets de vectorisation des communications que le compilateur peut mettre en place.

De plus, c'est également au compilateur de décider quel processeur devra effectuer chaque partie des instructions. Généralement, les compilateurs utilisent la *owner compute rule*, c'est-à-dire que c'est le processeur où est placée la donnée à écrire qui effectue le travail.

Par la suite, nous utiliserons cette règle. Le processeur qui effectue le calcul sera appelé *processeur hôte*.

Restrictions dues à HPF

En plus de cette impossibilité de tenir compte de la machine cible, la norme HPF impose certaines restrictions sur les `processors` :

- Un compilateur HPF doit au moins savoir accepter des `processors` ayant la même taille que la machine physique cible et des `processors` d'un seul élément [For97, p38,l14].
- Les éléments de deux `processors` différents, mais de même géométrie doivent se correspondre deux à deux [For97, p37,l23], c'est-à-dire que pour deux `processors` `P1` et `P2`, `P1(5,7)` et `P2(5,7)` sont supposés désigner le même processeur abstrait et doivent donc être placés sur le même processeur physique.
- Aucune spécification n'est définie pour deux `processors` de géométries différentes.
- Enfin, les données scalaires peuvent être alignées et distribuées de la même façon que des tableaux.

Pour l'évaluation des performances, ces restrictions ont des bons et des mauvais côtés :

- La taille d'un `processors` n'est généralement décidée qu'à l'exécution, par des appels aux fonctions intrinsèques qui permettent de déterminer la taille de la machine cible.

  Comme nous nous plaçons avant l'exécution, il faudra demander cette taille à l'utilisateur, comme c'est déjà le cas dans HPF-BUILDER.
- Tous les `processors` d'un programme qui ont la même géométrie peuvent être considérés comme un seul.

  Pour les calculs faisant intervenir des données distribuées sur ces `processors` on peut donc déterminer les communications comme s'il n'y avait qu'un seul `processors`.
- Par contre, pour les `processors` de géométries différentes, on ne peut pas prévoir le comportement du compilateur.

  Entre deux données placées sur deux `processors` de géométries différentes, on ne peut donc rien calculer. Le principe d'HPF conseille plutôt de n'utiliser qu'un seul `processors` à la fois, mais entre deux phases d'un algorithme, il peut être intéressant de changer la topologie sur laquelle repose le placement. Pour les redistributions qui sont alors à mettre en œuvre, on ne sait donc pas faire d'évaluation des coûts.
- Nous ne considérons ici que des accès à des tableaux, les scalaires étant interprétés comme des tableaux de taille 1.

À partir de ce point, on commence à avoir une idée de ce qu'on sera capable de calculer. Il faut maintenant déterminer quelles informations il est intéressant de montrer à l'utilisateur :

4.2.2 Quelle information doit-on visualiser ?

Le point fondamental est celui du temps d'exécution de l'instruction. Comme on ne veut pas exécuter le code, il faut évaluer ce temps d'exécution instruction par instruction : pour chaque instruction, il faut définir les communications nécessaires à sa préparation, déterminer combien d'instructions seront effectuées par chaque processeur et en déduire le temps global.

Cependant, ce calcul dépend complètement de la machine cible et de la méthode de compilation. Il faut connaître exactement les caractéristiques de la machine, le placement effectif choisi par le compilateur et les techniques d'optimisation utilisées. Comme on vient de le voir, si on veut rester au niveau d'HPF, ce résultat n'est pas possible à obtenir.

De plus, obtenir un temps d'exécution global, même pour une instruction donnée, ne permet pas de savoir comment le programmeur peut l'améliorer. Pour cela, il faut pouvoir déterminer comment on est arrivé à ce temps de calcul, c'est-à-dire qu'il faut pouvoir visualiser le détail de l'exécution : combien de calculs ont été effectués sur chaque processeur et combien de communications a-t-il fallu effectuer ?

On voit donc que non seulement on ne saura évaluer les coûts qu'en termes de mouvements de données et d'occupation des processeurs abstraits, mais que cette vision est la plus utile pour le programmeur, puisque c'est la seule sur laquelle il peut agir par l'intermédiaire des directives HPF.

En conclusion, la visualisation de l'efficacité d'un programme peut se limiter à la représentation du nombre de déplacements de données nécessaires et de la charge des processeurs.

On peut supposer que si l'utilisateur réussit à améliorer ces aspects au niveau des placements HPF, on obtiendra aussi une amélioration au niveau physique.

En effet, si on réduit le nombre de mouvements de données entre les éléments des `processors` HPF, le nombre de mouvements entre processeurs physiques sera également réduit. De même, la charge sur les `processors` est globalement proportionnelle à celle des processeurs physiques.

De ce fait, on peut se contenter, dans une première étape, d'une représentation de ces deux points, en considérant les `processors` comme une machine cible virtuelle.

Il faut maintenant savoir ce que doit contenir une telle représentation.

4.2.3 Que faut-il afficher ?

Il a été vu précédemment qu'une instruction HPF se décompose en trois étapes : récupération des données distantes, calculs, écriture des résultats. Pour chacune d'elle, l'analyse des directives de placement permet de déterminer quels processeurs vont avoir à travailler. Ce sont les résultats de cette analyse qu'on doit visualiser.

Ces trois étapes doivent donc correspondre à une visualisation de la répartition des calculs et du nombre de données à envoyer et recevoir.

De plus, pour pouvoir observer en détail l'effet des placements, il faut que ces visualisations se fassent au niveau de chaque processeur. Donc, pour chacun d'eux et pour chaque

instruction, il faut afficher :

- le nombre de données qu'il doit envoyer à d'autres processeurs,
- le nombre de données qu'il doit recevoir des autres,
- le nombre de calculs qu'il doit effectuer.

Enfin, pour corriger les problèmes de surplus de communications, le programmeur doit pouvoir comprendre d'où viennent (resp. où partent) les informations qu'un processeur doit recevoir (resp. envoyer). La visualisation doit donc permettre de suivre les origines et destinations des mouvements de données.

Maintenant qu'on a défini plus précisément les informations à afficher, il faut déterminer un moyen de les visualiser.

4.2.4 Comment afficher ?

Les informations à afficher sont clairement de deux types : des nombres (quantités de calculs et de communications) et des ensembles de positions (origines et destinations des communications).

La représentation d'ensembles de valeurs numériques est assez répandue : on peut utiliser des tableaux, des graphiques ou des jeux de couleurs comme on l'a vu dans les exemples d'interfaces graphiques de DAQV (section 2.4.4 page 63) et ANNAI (section 2.4.4 page 64).

Par contre, les sources et destinations de communications ont une structure de graphe assez lourde à représenter. Si on représente l'ensemble de ce graphe, comme le fait XPVM (section 2.3.2 page 57) on obtient un dessin difficile à utiliser et peu intuitif.

Il semble donc plus intéressant de chercher à représenter les communications autour d'un seul processeur à la fois. Pour cela, la méthode du curseur, utilisée pour la visualisation détaillée, peut être reprise : on utilise un curseur pour sélectionner un processeur abstrait et les processeurs contenant des données qui sont à recevoir ou à envoyer sont mis en valeur.

Cette méthode permet en plus de visualiser les quantités d'information à transmettre en plus de leur position : en effet, on peut mettre en valeur les autres processeurs dans des couleurs correspondant à une échelle de quantités de transferts.

Exemple : *Une interface graphique possible*

Considérons un exemple où on part de l'extrait de code suivant :

```
program eval
 integer, dimension(100,100) :: A,B
!HPF$ processors, dimension(16,16) :: P
!HPF$ distribute A(cyclic,block) onto P
!HPF$ distribute B(block,cyclic(2)) onto P
...
 A=B
end
```

On cherche à évaluer le volume de communications générées par l'instruction A=B. Cette instruction correspond à une boucle parallèle sur les 100×100 indices de A et B.

L'idée d'un zoom sur les communications est montrée dans la figure 4.1. Elle présente un exemple de ce que pourrait être l'interface graphique d'un module d'évaluation de coûts.

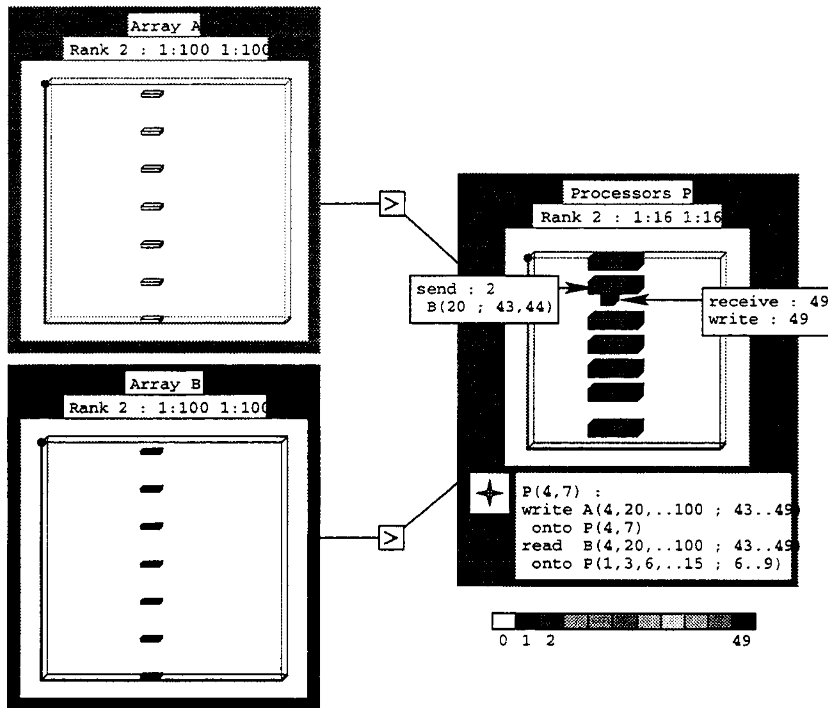


FIG. 4.1: *Exemple de visualisation des communications.*

La figure reprend le principe de la vue en squelette des directives HPF, avec une vue en fil de fer de chaque nœud. On voit dans la sélection de droite, qu'un curseur est affiché en `P(4,7)`. L'utilisateur peut le déplacer de la même façon que pour le mécanisme de zoom, mais cette fois-ci, les pavés affichés représentent les processeurs qui interviennent dans une communication. Ici, on voit que les processeurs des lignes 1, 3, 6 ... 15 et des colonnes 6 à 9 sont concernés.

Chacun de ces pavés a une couleur dépendant du nombre de communications dans lequel il intervient. Cette couleur correspond à un niveau dans l'échelle affichée en dessous de la sélection. Ainsi, `P(3,6)` est utilisé pour deux communications, alors que `P(3,9)` n'intervient que dans une seule.

Pour obtenir plus de détails, l'utilisateur peut cliquer un de ces pavés pour voir dans combien de lectures, écritures et calculs il intervient et pour en avoir le détail. Ainsi, la figure montre une sous-fenêtre dans laquelle on voit que `P(3,6)` intervient dans 2 lectures, mais dans aucune écriture. Une autre sous-fenêtre montre que `P(4,7)` effectue 49 lectures distantes et 49 écritures (et donc autant de calculs).

Si l'instruction sélectionnée fait partie d'un nid de boucles, ce sont les valeurs courantes des variables qui sont utilisées pour représenter une itération donnée. Ainsi, l'utilisateur peut visualiser les communications nécessaires autour d'un processeur donné, pour une itération donnée. Il lui suffit de déplacer le curseur et de modifier les valeurs par défaut des variables pour observer l'ensemble des communications.

Dans une autre partie de l'interface, on peut envisager d'afficher le total de calculs et de communications pour l'ensemble de l'exécution d'un nid de boucle. Dans ce cas, une représentation en niveaux de couleurs ou en histogrammes est suffisante.

De plus, on voit sur la figure que dans les sélections de gauche, les éléments de tableaux à transmettre sont mis en valeur en même temps que leur processeur hôte. Ainsi, on voit les parties de A représentées par des pavés. Ces pavés sont blancs pour indiquer qu'ils n'interviennent pas dans une communication. On comprend donc que c'est à cet endroit qu'il y a des données à écrire. Par contre, les parties de B indiquées sont remplies dans une couleur correspondant à la valeur 1 dans l'échelle. Ceci indique que ces données sont envoyées vers le processeur sélectionné.

L'utilisateur peut donc observer les communications au niveau des processeurs, mais aussi savoir à quelles données correspondent ces communications. De cette façon, il obtient directement les informations nécessaires pour corriger les directives de placement.

En effet, si l'utilisateur compare cet ensemble de pavés à ce qu'il peut obtenir avec le zoom pour observer la distribution, il verra immédiatement qu'il n'y a pas de correspondance entre eux et qu'il y a donc certainement un mauvais placement.

Un dernier problème vient de l'existence de directives dynamiques: à cause de ces directives, une même ligne de code peut être exécutée à des moments différents de l'exécution, dans des contextes où le placement est différent. Il est donc nécessaire de permettre à l'utilisateur de préciser un placement courant sur les données qui entrent en jeu dans l'évaluation.

Pour cela, on peut faire apparaître la liste de ces données et pour chacune, afficher la liste des placements possibles présents dans le programme. L'utilisateur n'a alors qu'à sélectionner les placements pour lesquels il veut visualiser l'efficacité, de la même façon qu'il fixe les valeurs des variables dans HPF-BUILDER.

D'un autre côté, le coût d'exécution d'une de ces directives dynamiques doit être lui-même pris en compte. Pour cela, il doit être possible de sélectionner une directive dynamique de la même façon qu'une ligne de code « classique ». Le placement courant défini par l'utilisateur correspond alors au placement avant l'exécution de la directive.

Cet exemple montre donc qu'il est possible de visualiser les informations sur les communications induites par le placement courant, instruction par instruction, voire même pour un groupe d'instructions.

Maintenant que le cahier des charges d'un tel outil a été défini, il faut déterminer de quelle façon on peut calculer les informations qui y sont présentées.

4.2.5 Que doit-on calculer?

En résumé, on sait maintenant sur quelles bases on peut effectuer des calculs et quelles informations il faut aller chercher. Il faut donc ensuite déterminer un modèle de calcul qui permette d'obtenir les informations demandées.

La proposition de zoom part d'un placement courant et d'une instruction donnée. Il faut donc commencer par modéliser la notion d'instruction et de placement courant autour de

cette instruction.

Ensuite, l'utilisateur sélectionne un processeur abstrait p, dans un **processors** P. L'interface graphique lui fournit alors les informations suivantes :

- Le nombre d'opérations à y effectuer. Nous noterons cette valeur O_p.
- L'ensemble des données à écrire sur ce processeur, que nous noterons $\mathcal{W}_p$.
- L'ensemble des données à rapatrier sur ce processeur, que nous noterons $\mathcal{C}_p$ et leur nombre, noté C_p.
- L'ensemble des processeurs de P contenant des éléments à communiquer à p, que nous noterons $\mathcal{R}_p^P$
- Le nombre d'éléments à communiquer pour chacun de ces processeurs p', que nous noterons $R_p^{p'}$
- Dans les autres sélections :

  L'ensemble des éléments de chaque tableau T qui interviennent dans les opérations à effectuer sur p, que nous noterons $\mathcal{R}_p^T$.

  Pour chacun de ces éléments t, s'il s'agit d'une donnée à écrire, du fait de la *owner compute rule*, il est local, sinon il est à lire et peut être alors distant ou local. Cet état sera indiqué par une valeur de R_p^t valant zéro ou un.

Ensuite, lorsque l'utilisateur clique un des processeurs p', la valeur de $R_p^{p'}$ est rappelée et le détail des éléments à communiquer est ajouté. Nous noterons cet ensemble $\mathcal{R}_p^{p'}$. Chaque élément de cet ensemble est une référence à un élément de tableau t pour lequel la valeur R_p^t nous dit s'il y a communication ou non.

Exemple

Si on reprend l'exemple de l'extrait de programme précédent, les variables qui viennent d'être définies prennent les valeurs suivantes :

- Il y a 49 opérations effectuées sur `P(4,7)` :

$$O_{P(4,7)} = 49$$

- Il y a 49 éléments à y écrire :

$$\mathcal{W}_{P(4,7)} = \{A(4, 20, 36, .., 100; 43..49)\}$$

- Ce processeur reçoit des valeurs des processeurs suivants :

$$\mathcal{R}_{P(4,7)}^P = \{P(1, 3, 6, 8, 10, 12, 15; 6, 7, 8, 9)\}$$

- Pour `P(3,6)`, il y a deux valeurs à communiquer :

$$R_{P(4,7)}^{P(3,6)} = 2$$

- Pour la sélection sur B, il faut communiquer les valeurs suivantes :

$$\mathcal{R}^{B}_{P(4,7)} = \{B(4,20,..,100;43..49)\}$$

- Ces valeurs sont à lire et sont toutes distantes, tous les $R^{B(i,j)}_{P(4,7)}$ correspondants valent donc 1.
- Le nombre total de communications vers $P(4,7)$ est $C_{P(4,7)} = 49$.

Nous avons donc défini les variables à calculer pour disposer des informations nécessaires à la visualisation. Il reste à déterminer comment elles peuvent être calculées.

4.2.6 Comment calculer ?

Une instruction engendre du calcul parallèle et des communications lorsqu'elle entraîne une série de lectures sur des données distribuées, un calcul sur ces données, puis le rangement du résultat.

Nous allons observer le comportement d'une telle instruction au niveau du calcul d'un élément donné du résultat, pour voir ensuite le cas de l'ensemble d'une opération parallèle.

Calcul autour d'un élément unique

Pour l'évaluation d'un élément, il s'agit donc une instruction d'affectation du type

$$G(\vec{I_G}) \leftarrow f(D_1(\vec{I_{D_1}}), D_2(\vec{I_{D_2}}), ...)$$

C'est-à-dire que le tableau G (comme « partie *g*auche de l'affectation »), à l'indice $\vec{I_G}$ reçoit le résultat de l'évaluation de la fonction f sur les données de la partie droite, $D_i(\vec{I_{D_i}})$.

Pour exécuter une telle instruction, il faut exécuter un certain nombre d'étapes :

- décider quel processeur effectuera le travail,
- rapatrier toutes les données nécessaires à ce calcul,
- effectuer l'opération,
- ranger le résultat.

Charge de calcul : Comme on considère que le compilateur utilise la *owner compute rule*, c'est le processeur qui contient $G(\vec{I_G})$ qui effectue le calcul. Dans ce cas, la charge de calcul des processeurs est nulle partout, sauf sur le processeur hôte, que nous noterons p_h.

Il faut donc déterminer sur quel processeur abstrait est placée la donnée à écrire. Pour cela, on peut utiliser les fonctions de projection vues dans le chapitre précédent (section 3.4 page 96). Si la donnée G est placée sur le **processors** P, sa position est donnée par $P^{G \to P}$:

$$p_h = P^{G \to P}(\vec{I_G})$$

On obtient alors $\mathcal{W}_{p_h} = G(\vec{I_G})$ ainsi que $O_{p_h} = 1$ et $O_p = 0$ partout ailleurs.

Données à lire : Pour les communications, il faut déterminer où sont placées les données à lire et toutes celles qui ne sont pas locales engendrent une communication. Cette liste de données distantes permet alors de déterminer les $\mathcal{R}_{p_h}^{D_i}$.

Comme HPF autorise la réplication des tableaux, une donnée peut être présente sur plusieurs processeurs. Cela implique deux contraintes : si une donnée source est répliquée, il faut vérifier qu'aucun des exemplaires n'est sur le processeur qui doit faire l'opération.

Si une donnée destination est répliquée, le compilateur peut agir de différentes façons. Il peut effectuer le calcul sur tous les processeurs hôtes, ou au contraire, ne l'effectuer que sur un seul processeur, puis mettre à jour le résultat sur les autres, ou encore choisir n'importe quelle solution intermédiaire.

Comme il est impossible de savoir ce que fera le compilateur, nous écarterons pour l'instant cette situation. Nous considérerons donc que $P^{G \rightarrow P}(\vec{I_G})$ ne contient qu'un seul élément. Par contre, les $P^{D_i \rightarrow P}(\vec{I_{D_i}})$ peuvent contenir plusieurs éléments.

Enfin, nous ne considérerons que le cas où les D_i sont tous projetés sur le même **processors** que G, ou sur un **processors** de même géométrie auquel il peut être assimilé. En effet, nous avons vu au début de la section qu'on ne peut rien évaluer sinon.

Il faut donc déterminer les ensembles d'éléments de tableaux qui interviennent dans l'instruction et dont le placement n'a aucune projection commune à celle de $G(\vec{I_G})$:

$$\forall i, \mathcal{R}_{p_h}^{D_i} = \{D_i(\vec{I_{D_i}}) \mid p_h \notin P^{D_i \rightarrow P}(\vec{I_{D_i}})\}$$

Pour les autres processeurs, ces ensembles sont vides, puisqu'il n'y a aucune donnée à y rapatrier :

$$\forall i, \forall p \neq p_h, \mathcal{R}_p^{D_i} = \emptyset$$

Pour chaque élément de chaque D_i, on a $R_{p_h}^{D_i(\vec{I_{D_i}})} = 1$ s'il est dans $\mathcal{R}_{p_h}^{D_i}$, sinon, il vaut zéro.

Processeurs concernés : Pour chacune de ces données, sa projection sur le **processors** ajoute un élément à l'ensemble des processeurs concernés par les communications. L'ensemble $\mathcal{R}_{P_h}^{P}$ peut donc être défini comme l'ensemble des processeurs contenant un des éléments des $\mathcal{R}_{p_h}^{D_i}$:

$$\mathcal{R}_{P_h}^{P} = \{p \mid \exists i \exists D_i(\vec{I_{D_i}}) \in \mathcal{R}_{p_h}^{D_i}, P^{D_i \rightarrow P}(\vec{I_{D_i}}) = p\}$$

Là encore, pour les autres processeurs, cet ensemble est vide.

Comme on l'a vu précédemment, les données à lire peuvent être répliquées. Dans ce cas, la définition qui vient d'être proposée n'est plus valide, puisque $P^{D_i \rightarrow P}(\vec{I_{D_i}})$ contient plusieurs éléments.

Dans cette situation, seul le compilateur décide du processeur sur lequel la donnée sera lue. On ne peut alors pas calculer $\mathcal{R}_{p_h}^{P}$.

Chaque processeur p' de cet ensemble peut être concerné par plusieurs communications. Leur nombre correspond à $R_{p_h}^{p'}$:

$$\forall p', R_{p_h}^{p'} = \text{Card}(\{D_i(\vec{I_{D_i}}) \in \mathcal{R}_{p_h}^{D_i} \mid P^{D_i \rightarrow P}(\vec{I_{D_i}}) = p'\})$$

Là encore, en cas de réplication d'un des D_i, ces valeurs ne sont pas calculables.

Nombre de communications : Pour p_h, l'ensemble des communications à effectuer est donc l'union des $\mathcal{R}_{p_h}^{D_i}$

$$\mathcal{C}_{p_h} = \bigcup_{\forall i} \mathcal{R}_{p_h}^{D_i}$$

Le nombre de communications en réception est égal au cardinal de cet ensemble. C'est aussi la somme des communications à effectuer sur les autres processeurs.

$$C_{p_h} = \text{Card}(\mathcal{C}_{p_h}) = \sum_{\forall p \in \mathcal{R}_{p_h}^{P}} R_{p_h}^{p}$$

Généralisation à l'ensemble d'une opération parallèle

Pour l'instant, nous n'avons parlé que du cas d'une instruction séquentielle. Bien sûr, il est beaucoup plus intéressant de savoir effectuer cette évaluation sur des constructions parallèles comme les opérations de tableaux ou les constructions `forall` et `do..enddo`.

Dans ce cas, on arrive à une instruction effectuant un ensemble d'affectations sur l'espace d'itération d'un ensemble d'indices. De plus, chaque lecture/écriture se fait sur un tableau, à une position dépendant de ces indices.

De même, le cas d'une directive dynamique, peut être assimilé à une affectation d'une variable à elle même, avec un placement différent entre la variable lue et celle écrite.

On arrive donc aux formules suivantes :

- L'ensemble des indices de boucles parcourus est noté $\{\vec{I}\} = \{i_1\} \times \{i_2\} ... \times \{i_n\}$. Une itération est repérée par un élément $\vec{I}$ de cet ensemble.

  Dans l'exemple, cet ensemble est $\{1..100\} \times \{1..100\}$.

- À l'itération $\vec{I}$, l'accès à un tableau T est effectué à travers une fonction d'accès $\phi_T(\vec{I})$. Dans l'exemple, l'accès est direct ; ϕ_B est donc la fonction identité.

- On définit $\mathcal{O}_{P(\vec{J})}$ comme étant l'ensemble des itérations pour lesquelles l'opération est effectuée sur le processeur abstrait `P(`$\vec{J}$`)` et $O_{P(\vec{J})}$ en est alors le cardinal. Cet ensemble contient les itérations pour lesquelles la donnée à écrire est placée sur ce processeur :

$$\mathcal{O}_{P(\vec{J})} = \{\vec{I} \mid P^{G \rightarrow P}(\phi_G(\vec{I})) = \vec{J}\}$$

  Dans l'exemple, $\mathcal{O}_{P(4,7)} = \{4, 20, 36, ..100\} \times \{43..49\}$

- Sur un processeur $P(\vec{J})$, les données à recevoir sont toutes celles qui sont distantes et dont on a besoin pour calculer une des itérations de $\mathcal{O}_{P(\vec{J})}$. Pour chaque partie droite D_i, on a donc :

$$\mathcal{R}^{D_i}_{P(\vec{J})} = \{D_i(\phi_{D_i}(\vec{I})), \vec{I} \in \mathcal{O}_{P(\vec{J})} \mid \vec{J} \notin P^{D_i \to P}(\phi_{D_i}(\vec{I}))\}$$

  Dans l'exemple, $\mathcal{R}^{B}_{P(4,7)} = \{B(4, 20, 36..100; 43..49)\}$

- Les processeurs devant envoyer des données à un processeur $P(\vec{J})$ sont donc ceux qui contiennent des éléments des différents $\mathcal{R}^{D_i}_{P(\vec{J})}$:

$$\mathcal{R}^{P}_{P(\vec{J})} = \{p \mid \exists D_i(\phi_{D_i}(\vec{I})) \in \mathcal{R}^{D_i}_{P(\vec{J})}, P^{D_i \to P}(\phi_{D_i}(\vec{I})) = p\}$$

  Dans l'exemple, on retrouve $\mathcal{R}^{P}_{P(4,7)} = \{P(1, 3, 6..15; 6..9)\}$.

- Pour chacun de ces processeurs, le nombre d'éléments à envoyer s'exprime de la même façon que dans le cas séquentiel, à la fonction d'accès près :

$$\forall p', R^{p'}_{P(\vec{J})} = \mathrm{Card}(\{D_i(\phi_{D_i}(\vec{I})) \in \mathcal{R}^{D_i}_{P(\vec{J})} \mid P^{D_i \to P}(\phi_{D_i}(\vec{I})) = p'\})$$

- L'ensemble des communications à effectuer reste donc l'union des $\mathcal{R}^{D_i}_{P(\vec{J})}$

$$\mathcal{C}_{P(\vec{J})} = \bigcup_{\forall i} \mathcal{R}^{D_i}_{P(\vec{J})}$$

$$C_{P(\vec{J})} = \mathrm{Card}(\mathcal{C}_{P(\vec{J})}) = \sum_{\forall p \in \mathcal{R}^{P}_{P(\vec{J})}} R^{p}_{P(\vec{J})}$$

Conclusion

Cette modélisation permet de calculer les valeurs des informations nécessaires à l'interface graphique définie précédemment. On a donc un point de départ solide pour sa réalisation. Il reste à mettre au point une méthode permettant d'effectuer ces calculs en temps raisonnable.

4.3 Mise en œuvre

La section précédente a fait le tour d'un cahier des charges de ce que devrait être une interface graphique d'évaluation des coûts dans un programme HPF.

Un exemple de ce que pourrait être l'apparence de cette interface a été montré ; c'est maintenant à l'aspect calcul que nous allons nous intéresser.

Dans un tel programme, il n'est pas nécessaire d'effectuer les traitement à des vitesses importantes. La seule contrainte est de pouvoir travailler en *temps interactif*, c'est-à-dire d'avoir des temps de réponse raisonnables pour une utilisation interactive.

Cependant, même sans contraintes fortes, les calculs qui viennent d'être exposés risquent fortement d'exploser en temps de calcul, dans des cas où les tableaux sont de taille importante.

C'est pourquoi nous allons étudier quelques approches qui ont été envisagées pour mettre en œuvre ces calculs.

4.3.1 Calcul direct

Si on cherche à calculer les communications induites par une seule instruction séquentielle, nous avons vu qu'on est réduit à quelques calculs de projections. Ces calculs sont identiques à ceux utilisés pour l'affichage de l'effet de zoom. Leur implémentation est donc directe.

Par contre, dans le cas d'une instruction parallèle, il faut itérer ce calcul sur tout l'espace d'itération de l'instruction.

Dans l'exemple utilisé dans les sections précédentes, l'instruction `A=B`, correspond à une boucle parallèle sur les 100×100 indices de `A` et `B`. Le calcul direct du coût de communication implique donc l'évaluation de $P^{A \rightarrow P}(i, j)$, celle de $P^{B \rightarrow P}(i, j)$ puis la comparaison des deux pour les 10000 itérations de i et j.

Il est clair qu'un tel calcul ne sera pas réalisable en temps interactif et encore moins pour des boucles imbriquées, sur des expressions complexes et avec des tableaux de grande taille.

Elle peut donc être utilisée pour un premier prototypage de l'interface graphique, mais il sera nécessaire de trouver une méthode plus efficace.

C'est dans ce but que deux projets ont été lancés : le calcul par simulation et l'utilisation des méthodes polyhédriques.

4.3.2 Calcul par simulation

Principe

Lorsqu'un compilateur génère un code à partir d'un source HPF, il y insère le code nécessaire pour récupérer les données distantes avant chaque instruction.

Ainsi, dans l'exemple de programme présenté dans la section précédente, le code généré sur le processeur physique qui « héberge » `P(4,7)` devra stocker des parties de `A` et `B` spécifiées par la distribution.

Comme le montre la figure 4.2, l'utilisation de HPF-BUILDER nous permet de voir facilement que ce processeur stocke `A(4,20,...100 , 43..49)` et `B(22..28 , 13,14,45,46,77,78)`

Le code sur ce processeur devra donc effectuer les opérations suivantes [1] :

1. On a vu dans la section 1.5.2 page 32 que la sémantique d'HPF oblige à passer par un temporaire.

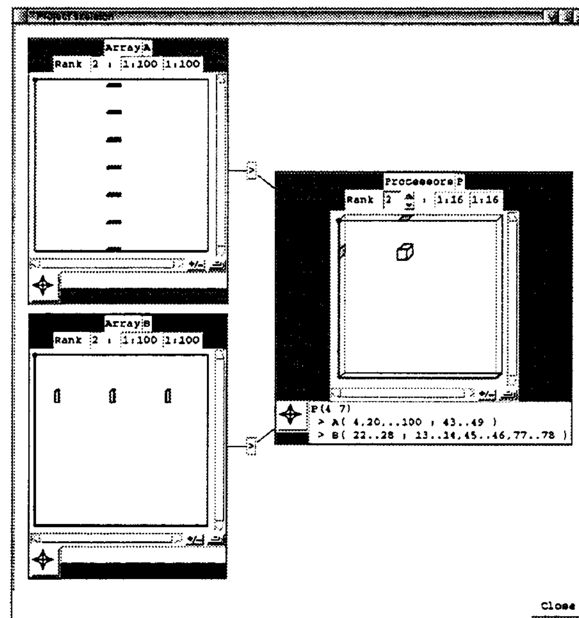


FIG. 4.2: *Tableaux stockés sur P(4,7).*

```
pour chaque i de 1 à 7                    l'indice de ligne local
   i' vaut i*7+3                          son équivalent global
   pour chaque j de 1 à 7                 idem pour les colonnes
      j' vaut j+42
      localiser B(i',j')
      le rapatrier dans un temporaire T(i,j)
   fin
fin
pour chaque élément de T
   l'affecter à A
fin
```

Si le compilateur sait factoriser les communications, il agira source par source, en rapatriant les éléments de T par blocs provenant du même processeur distant.

Le résultat ressemblera donc au code suivant :

```
pour chaque processeur possédant des parties de B(4,20,36...100 , 43..49)
   récupérer cette partie
   la ranger dans T
fin
pour chaque élément de T
   l'affecter à A
fin
```

À ce niveau, on remarque que le compilateur doit donc générer un code capable de déterminer quelle section il doit ramener de chaque processeur. La simulation de ce code devrait donc permettre d'évaluer la quantité de données transmise.

Réalisation

C'est cette idée qui est à la base des travaux qu'a effectué Denis RUCKEBUSCH dans son mémoire de DEA [Ruc98, BDLR98]. Ces travaux ont consisté à rechercher dans le code généré par un compilateur les appels aux procédures qui effectuent le calcul des sections de tableaux à transmettre pour chaque processeur. L'évaluation de ces parties permet alors le calcul des variables définies dans la modélisation (section 4.2.5 page 114).

Nous avons vu au chapitre précédent qu'HPF-BUILDER utilise un serveur d'informations comme intermédiaire pour l'analyse du code. Ce serveur est construit autour de l'analyseur syntaxique du compilateur ADAPTOR.

ADAPTOR fonctionne en plusieurs étapes :

- il commence par une étape de génération d'un arbre syntaxique complet du fichier source (c'est cette étape qu'utilise notre serveur),
- il effectue ensuite une première analyse de cet arbre pour repérer toutes les déclarations et les références qui y sont faites,
- l'arbre obtenu est ensuite modifié par plusieurs passes successives. Ces transformations en font un arbre décrivant un programme FORTRAN effectuant des appels à une librairie de gestion de tableaux distribués, la DALIB,
- il ne reste alors plus qu'à générer le code correspondant à cet arbre pour obtenir un programme FORTRAN fonctionnant en SPMD.

Si on inclut dans le serveur d'informations les étapes de modification de l'arbre, on peut alors utiliser ce serveur pour savoir quels appels à la DALIB sont générés pour une instruction HPF donnée. On peut alors, à partir d'un programme indépendant, effectuer ces appels et obtenir ainsi des informations sur le nombre de communications.

Il faut bien remarquer qu'on ne fait qu'exécuter les fonctions de la DALIB nécessaires aux mouvements de données qu'implique l'instruction à observer. L'ensemble du code HPF n'est pas exécuté, il suffit d'appeler les fonctions qui effectuent la mise en place de descripteurs des données locales, puis les fonctions qui servent au rapatriement des données. Quelques modifications de la DALIB permettent alors de compter les données à transmettre, au lieu d'effectuer réellement les communications. On obtient ainsi le nombre de données à transmettre à partir de chaque processeur.

On a donc moyen d'obtenir les informations nécessaires à l'évaluation des performances, pour un processeur donné, mais pour toutes les itérations d'une instruction parallèle.

On n'a donc plus qu'à itérer cette méthode sur chaque processeur abstrait, ce qui est beaucoup moins coûteux.

Dans le cas d'un nid de boucles séquentielles, cette méthode nécessite quand même une évaluation pour chaque itération des boucles. C'est pourquoi nous allons maintenant présenter une autre méthode, encore plus générale.

4.3.3 Utilisation du calcul polyhédrique

Les fonctions de projections à utiliser pour calculer les placements sont quasiment affines :

Les alignements ne font intervenir que des additions et des multiplications d'indices par des constantes. Les distributions impliquent des divisions par des constantes (qui sont des réciproques de multiplications) et des modulos qu'on peut calculer par des systèmes d'inéquations linéaires.

De plus, l'espace d'itération d'un nid de boucle décrit un convexe de $\mathbb{N}^n$.

Tous ces types de structures et d'opérations (applications linéaires, inverses et manipulations de convexes) peuvent être manipulés par l'outil CIPOL développé par Xavier REDON [Red98]. Il semble donc possible d'utiliser cet outil pour effectuer les calculs d'évaluation.

L'intérêt de cette méthode réside dans l'aspect symbolique des structures manipulées par CIPOL : il peut manipuler des expressions arithmétiques contenant des paramètres.

On peut donc appliquer les fonctions de projection pour obtenir des descriptions des convexes décrits par les ensembles de données source et destination d'une instruction. Ensuite, CIPOL peut évaluer des expressions entre ces convexes pour en calculer les intersections.

De plus, CIPOL permet de calculer le nombre d'éléments d'un tel convexe, c'est-à-dire le cardinal des ensembles manipulés, donc les coûts recherchés.

Par contre, cette méthode est limitée aux cas linéaires. Si un accès est effectué à travers une indirection, ou par une fonction non linéaire (`A(B(i))` ou `A(i*j)`), CIPOL n'est plus utilisable.

Cette méthode n'est pas encore complètement implémentée, car les expressions paramétrées obtenues sont très complexes. Il est nécessaire de travailler encore sur la simplification automatique de ces expressions. De plus, la génération des formules de départ à partir des informations fournies par le serveur n'est pas immédiate.

Les premiers résultats sont présentés dans [BR98]

4.4 Conclusion, perspectives

Ce chapitre a montré comment il est possible de calculer *a priori* une approximation du coût induit par un placement donné et comment on peut présenter graphiquement ces résultats.

Nous avons vu qu'il n'est pas possible de calculer des coût exacts sans se restreindre à un compilateur particulier. Par contre, une bonne indication du comportement du programme peut être obtenue. Les informations à visualiser et les moyens de les calculer ont été exposés.

Nous avons vu que différentes méthodes de calcul de ces informations sont envisageables. Il est notamment possible de simuler le fonctionnement d'un compilateur donné pour en déduire la façon dont il effectue réellement les communications, mais sans pour autant être obligé d'exécuter le code compilé. D'un autre côté, une méthode générique, indépendante de la machine et du compilateur a été présentée. Celle ci est plus souple et fournit des résultats précis au niveau du langage, mais apporte donc moins d'informations en termes de coût réel d'exécution.

Cette méthode est indépendante du compilateur et de la machine cible. De même, la simulation d'ADAPTOR ne prend en compte que les nombres de mouvements de données et les résultats sont donc génériques. Une fois au point, ces méthodes pourront être adaptées enfin de les spécialiser sur un compilateur particulier. Chacune de ces spécialisations sera alors capable de prendre en compte les optimisations spécifiques du compilateur, telles que les options de recouvrements entre processeurs et la vectorisation des communications.

De plus, être dédié à un compilateur permet de savoir comment sont traités les cas insolubles au niveau d'HPF, tels que les affectations à des données répliquées et les communications entre `processors` de géométries différentes.

Enfin, dans ce cas, il devient possible de prendre en compte les caractéristiques de la machine cible afin d'effectuer des évaluations en temps d'exécution.

Chapitre 5

Démonstration

Could I know a correct direction for installing HPF-BUILDER on my system?

Un client potentiel

I got install problem now.
These are error messages ...

Le même, 5 heures plus tard ...

5.1 Présentation

Le point central de cette thèse n'est pas constitué d'une démonstration mathématique ou de l'élaboration d'un modèle théorique, mais d'un prototype d'interface graphique : HPF-BUILDER. Il s'agit d'un logiciel, c'est-à-dire, par définition, quelque chose qu'on ne peut pas montrer sur papier autrement qu'en affichant des copies d'écran une par une. Or, cette solution est lourde et peut démonstrative.

L'intérêt d'HPF-BUILDER est de rendre interactive et plus conviviale l'insertion de directives HPF dans un code. Pour montrer comment nous sommes arrivés à ce résultat, la moindre des choses est donc d'en faire une démonstration elle aussi interactive.

C'est pourquoi nous avons choisi de terminer ce document par un chapitre d'une forme inhabituelle. Cette page constitue une introduction à ce chapitre. La suite n'est pas un document écrit; il s'agit d'une démonstration de l'utilisation d'HPF-BUILDER dont le support est le CD-ROM fourni avec le document. Ce CD-ROM contient les outils nécessaires pour

démarrer une démonstration interactive montrant comment on peut effectuer l'insertion, la visualisation et la modification de directives HPF dans un petit programme d'exemple.

Les deux codes suivants montrent le programme d'exemple utilisé dans la démonstration, avant et après les modifications qui y sont montrées :

```
program test
  integer :: NCol
  parameter (NCol=20)

 ...

  call MV(a,x,y,10)

contains
  subroutine MV(M,V,R, NLine)
    integer :: NLine
    real, dimension(NLine,NCol), intent(in):: M
    real, dimension(NCol), intent(in)      :: V
    real, dimension(NLine), intent(out)    :: R

    integer :: i, k

    R(1:NLine)= 0.0
    do k = 1,NCol
      forall(i= 1:NLine)
        R(i)=R(i) + V(k)*M(i,k)
      end forall
    end do
  end subroutine
end
```

```
...

  subroutine MV(M,V,R, NLine)
!HPF$ PROCESSORS MyProc(NUMBER_OF_PROCESSORS()/4,4)
    integer :: NLine
    real, dimension(NLine,NCol), intent(in):: M
!HPF$ DISTRIBUTE M (BLOCK,CYCLIC) ONTO MYPROC
    real, dimension(NCol), intent(in)      :: V
    real, dimension(NLine), intent(out)    :: R
!HPF$ ALIGN V(:) WITH M(*,:)

    R(1:NLine)= 0.0
    do k = 1,NCol
!HPF$ REALIGN R(i) WITH M(i,k)
      forall(i= 1:NLine)
        R(i)=R(i) + V(k)*M(i,k)
      end forall
    end do
  end subroutine
```

5.2 Démarrage de la démonstration

La démonstration est disponible pour plusieurs architectures et systèmes d'exploitation : PC sous LINUX ou WINDOWS et station SUN sous SUNOS ou SOLARIS.

Sous WINDOWS, il suffit de consulter le contenu du CD-ROM à partir du gestionnaire de fichiers et d'exécuter le fichier `Demonstration.Windows`. Sur système UNIX, il faut effectuer le montage du CD et y exécuter le programme `Demonstration.Unix`. Ce script détecte le type de la machine et lance alors la démonstration si elle est disponible pour cette architecture.

Une fois démarrée, la démonstration propose une grande fenêtre (voir figure 5.1) surmontée d'une barre de menus (repère 1).

La fenêtre principale présente des éléments d'HPF-BUILDER complétés par des messages d'explication (repère 2). L'animation d'une pseudo-souris (repère 5) simule les actions d'un utilisateur de l'interface graphique.

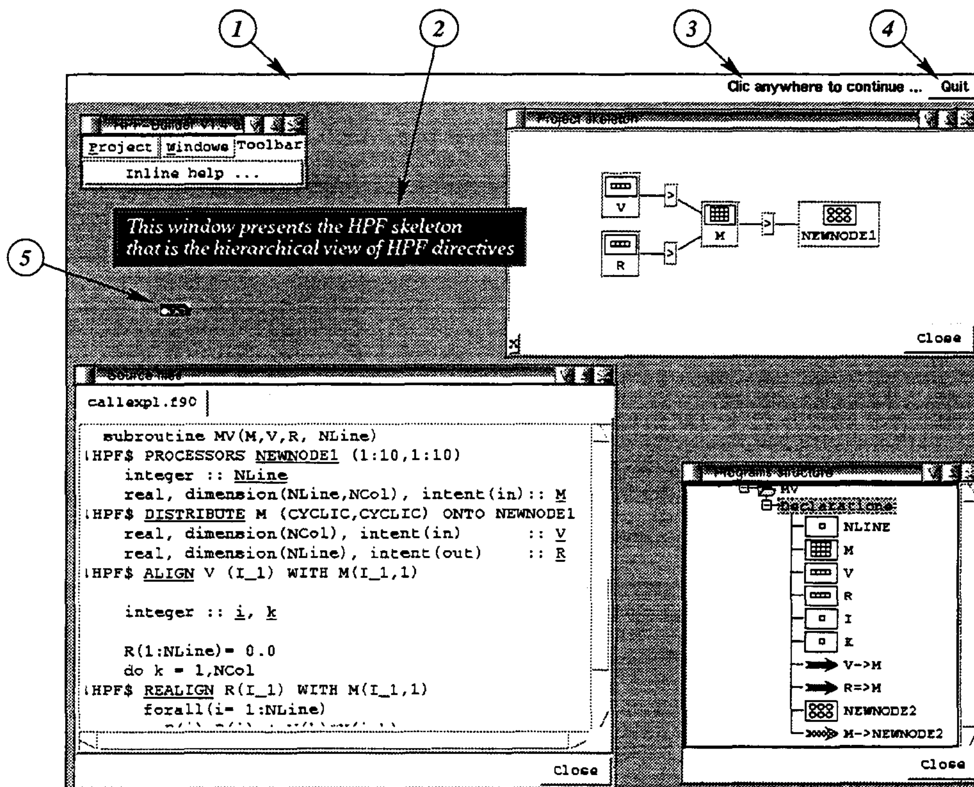


FIG. 5.1: *Vue de la démonstration.*

À droite de la barre de menu, un bouton `Quit` (repère 4) permet de sortir à tout moment de la démonstration. Quand un message *Clic anywhere to continue* apparaît à coté de ce bouton (repère 3), la démonstration stoppe et il suffit de cliquer n'importe où dans la fenêtre pour continuer. Dans le reste de cette barre, des boutons apparaissent régulièrement pour proposer différentes options à l'utilisateur. Ces options sont mise en place de façon à offrir une interface de navigation dans la démonstration.

Lors d'une première utilisation, le lecteur peut choisir l'option *continue* à chaque fois afin de faire le tour de toute la démonstration. Ensuite, il pourra naviguer par les autres boutons pour consulter directement certains points.

5.3 À suivre ...

Rendez-vous sur le CD-ROM ...

Conclusion

Résumé des travaux

Tout au long de ce document, nous avons cherché à établir la nécessité de fournir aux scientifiques des outils d'aide à la programmation parallèle. Pour cela, nous avons cherché à isoler toutes les difficultés qu'ils rencontrent lors de chaque étape de développement d'une application parallèle.

Nous avons étudié quelques outils existants afin d'en faire ressortir les points forts.

À partir de la, nous avons défini l'ébauche du cahier des charges d'un environnement de développement en HPF, de la conception de l'algorithme jusqu'aux derniers réglages de l'optimisation de performances. Cet environnement reprend le principe de nombreux outils existants, mais propose d'unifier leurs interfaces dans une suite logicielle unique.

Parmi les composants proposés, un outil d'aide à l'édition de directives de placement a été cité. Comme il n'existe pas actuellement de tel outil, nous avons détaillé les fonctionnalités qu'il devrait apporter.

Un prototype de cet outil a ensuite été présenté : HPF-BUILDER. Il apporte principalement un moyen de visualiser les directives HPF de placement et permet de les éditer de façon entièrement graphique. Il donne la possibilité d'observer globalement et en détail la structure des directives d'alignement et de distribution. Pour rendre plus attrayante cette présentation, une démonstration interactive accompagne le manuscrit.

Pour étendre l'aide qu'il apporte à l'utilisateur, il faudrait qu'HPF-BUILDER intègre la possibilité d'évaluer avant compilation l'efficacité des placements définis par le programmeur. Une étude préliminaire de cette extension a été présentée : nous avons montré comment devrait être l'interface graphique d'une telle extension et par quels moyens il est possible de calculer les informations que celle-ci doit afficher.

Perspectives

Développer une interface graphique n'est pas une tâche aisée. L'auteur d'un programme trouve forcément ce qu'il fait intuitif, puisque c'est lui qui l'a imaginé. Il faut donc constamment se mettre à la place de quelqu'un qui en sait moins que soi afin de déterminer si ce qu'on fait est réellement simple à utiliser. Dans le cas d'un outil d'aide à la programmation, cette situation s'aggrave, puisqu'on a comme public d'autres programmeurs ! De plus, il est difficile de déterminer si les informations fournies par HPF-BUILDER sont réellement celles dont a besoin le programmeur.

Pour le savoir, il manque à HPF-BUILDER une validation « sur le terrain ». Seule une utilisation du logiciel sur des applications réelles permettra de savoir si cet outil apporte réellement une aide au programmeur. Dans ce but, des contacts avec des utilisateurs potentiels ont été initiés.

Parallèlement à cette phase de test, quelques améliorations restent à apporter au prototype. Il a été vu dans sa description que les cas de variables à plus de trois dimensions ne sont pas encore gérés. De plus, quelques aspects ne sont pas encore au point (l'utilisation de directives dynamiques, notamment, n'effectue pas l'insertion de l'attribut `dynamic`).

De plus, la distribution effective du produit passe par la mise au point d'une procédure d'installation solide, facilement disponible et accessible. Elle nécessite aussi l'écriture d'un manuel d'utilisation.

L'utilisation d'HPF-BUILDER peut très bien ne pas se limiter au portage de programmes FORTRAN. On peut envisager son utilisation dans un but de formation à HPF : la visualisation des directives peut en effet servir de support à l'enseignement du langage.

Enfin, HPF-BUILDER a été développé dans l'esprit d'une intégration à un ensemble d'outils, dans le but d'obtenir un environnement de développement complet. Sa mise en œuvre autour d'un serveur d'information facilite cette intégration.

La principale perspective d'évolution de cet outil est donc soit le développement et l'ajout des autres composants autour du serveur, soit son intégration dans un autre projet, tels que ceux décrits à la fin du chapitre 2 (notamment FITS et HPFIT).

Dans les deux cas, la mise en œuvre effective de l'interface graphique de pré-évaluation s'inscrit dans ces perspectives. On peut imaginer le développement un composant indépendant, utilisant le serveur d'information ou une intégration directe dans l'interface existante d'HPF-BUILDER.

Bibliographie

[1.195] prs 1.1. The ParaDigme Parallel Action. In *Supercomputing'95, research exhibit*, San Diego, California, 1995.

[AC92] American National Standards Institute and Computer and Business Equipment Manufacturers Association. *American National Standard for programming language, FORTRAN — extended: ANSI X3.198-1992: ISO/IEC 1539: 1991 (E)*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1992.

[Ame78] American National Standards Institute. *American National Standard programming language FORTRAN: approved April 3, 1978, American National Standards Institute, Inc.* American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1978.

[Ann] Annai Tool Environment. Swiss Center for Scientific Computing. *http://www.cscs.ch/Official/SoftwareTech/CSCS-NEC/Annai.html.*

[AVS] AVS/Express Overview. Advanced Visual System Inc. *http://www.avs.com/products/expovr.htm.*

[BBG+94] François Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *Proceedings. OONSKI '94*, Oregon, 1994.

[BDLR98] Pierre Boulet, Jean-Luc Dekeyser, Christian Lefebvre, and Denis Ruckebusch. Communication Pre-Visualization. In *HPF Second User Group Meeting*, Porto, Portugal, 1998.

[BKP93] François Bodin, Lionel Kervella, and Thierry Priol. Fortran-S: A Fortran Interface for Shared Virtual Memory Architectures. In IEEE, editor, *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 274–283, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. IEEE Computer Society Press.

[BR98] Pierre Boulet and Xavier Redon. Communication pre-evaluation in HPF. In *EuroPar'98*, SouthHampton, UK, 1998.

[Bra96] Thomas Brandes. *ADAPTOR User's Guide (Version 4.0)*. GMD, 1996. Available via anonymous ftp from `ftp.gmd.de` as `gmd/adaptor/docs/uguide.ps`.

[Bur] Margaret M. Burnett, Visual Language Research Bibliography. *http://www.cs.orst.edu/~burnett/vpl.html.*

[BZBB98a] Thomas Brandes, Falk Zimmermann, Christian Borel, and Marc Brédif. Evaluation of High Performance Fortran for an Industrial Computational Fluid Dynamics Code. In *VecPar'98, 3rd International Meeting on Vector and Parallel Processing*, pages 467–480, Porto, Portugal, 1998. Lectures Notes in Computer Science.

[BZBB98b] Thomas Brandes, Falk Zimmermann, Christian Borel, and Marc Brédif. Porting of the Industrial Computational Fluid Dynamics Code AEROLOG to HPF. In *HUG'98, 2nd HPF User Group Meeting*, Porto, Portugal, 1998.

[CMZ92] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Vienna Fortran — A Fortran Language Extension for Distributed Memory Multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[Coc] The COCKTAIL Compiler Toolbox web page. GMD/SCAI, Sankt Augustin, Germany. *http://www.gmd.de/SCAI/lab/adaptor/cocktail.html.*

[Cra84] Cray Research, Inc. *Fortran (CFT) reference manual.* Number SR-0009 in Publication. Cray Research, Inc., Minneapolis, MN, Revision K edition, 1984.

[Din] Dinemas. Pallas Gmbh. *http://www.pallas.de/pages/dimemas.htm.*

[DL97] Jean-Luc Dekeyser and Christian Lefebvre. HPF-BUILDER: A visual environment to transform FORTRAN 90 codes to HPF. *International Journal of Supercomputing Applications and High Performance Computing*, 11(2):95–102, 1997. présenté à *Workshop on Environments and Tools For Parallel Scientific Computing*, Faverges de la Tour, France, 1996.

[Edg92] Stacey L. Edgar. *Fortran for the '90s: Problem Solving for Scientists and Engineers.* Computer Science Press, Inc., 11 Taft Court, Rockville, MD 20850, USA, 1992.

[F 1] Imagine1 Inc. Homepage. Imagine1 Inc. *http://kumo.swcp.com/imagine1/.*

[FC94] Ian Foster and Mani Chandy. Fortran M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 1994.

[For] FORESYS Software Package. Simulog. *http://www.simulog.fr/foresys.*

[For93] High Performance Fortran Forum. High Performance Fortran Language Specification, version 1.0. Rice University, Houston, TX, 1993.

[For97] High Performance Fortran Forum. High Performance Fortran Language Specification, version 2.0. Rice University, Houston, TX, 1997.

[Fos95] Ian Foster, Designing and building parallel programs - concepts and tools for software engineering. 1995.
http://www.mcs.anl.gov/dbpp.

[GDD] GDDT - Graphical Data Distribution Tool. Johannes Kepler University Linz, Systems Programming Institute.
http://www.gup.uni-linz.ac.at:8001/research/datadist/.

[GM] Simson L. Garfinkel and Michael K. Mahoney. *NeXTStep programming.* TELOS (The Electronic Library Of Science).

[GUD96] Marc Gengler, Stéphane Ubéda, and Frédéric Desprez. *Initiation au parallélisme.* Masson, Paris, 1996.

[HB85] Kai Hwang and Faye A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill International, 1985.

[Höh] Robert Höhne, RHIDE.
http://www.tu-chemnitz.de/~sho/rho/rhide/rhide.html.

[HKK+91] Seema Hiranandani, Ken Kennedy, Chuck Koelbel, Ulrich Kremer, and Chau-Wen Tseng. An Overview of the Fortran D Programming System. Technical Report CRPC-TR91121, Rice University, 91.

[HPFa] High Performance Fortran Programming. University of Liverpool, UK.
http://www.liv.ac.uk/HPC/HTMLFrontPageHPF.html.

[HPFb] The High Performance Fortran Forum Home Page. High Performance Fortran Forum.
http://www.crpc.rice.edu/HPFF/home.html.

[HPFc] HPF Front End. Syracuse University et al.
http://www.npac.syr.edu/projects/pcrc/hpffe.html.

[HPFd] High Performance Fortran Integrated Tools.
http://www.ens-lyon.fr/~desprez/FILES/RESEARCH/SOFT/HPFIT/.

[Kin] Bill Kinnersley, The Language List. Computer Science Department, University of Kansas.
http://www.nightflight.com/foldoc/Dictionary.

[LD98] Christian Lefebvre and Jean-Luc Dekeyser. Visualization of HPF data mappings and of their communication cost. In *VecPar'98*, Porto, Portugal, 1998.

[Lef98] Christian Lefebvre. Visual edition of HPF mappings with HPF-BUILDER, poster. In *High-performance Computing and Networking '98*, Amsterdam, Netherlands, 1998.

[Mas91] MasPar Computer Corporation, Sunnyvale, CA. *MasPar Fortran User Guide*, 1991.

[Neb95] Frederik Nebeker. *Calculating the Weather.* Academic Press, 1995.

[Ous] John K. Ousterhout. *Tcl and the Tk Toolkit.* Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California Berkeley, CA94720.

[OV] An Overview Of Computational Science. Computational Science Education Project.
http://csep1.phy.ornl.gov/CSEP/OV/OV.html.

[PAR] PARC History. Xerox Palo Alto Research Center.
http://www.parc.xerox.com/history.html.

[PIP] PIPS: Automatic Parallelizer.
http://www.cri.ensmp.fr/ pips/index.html.

[PRi] PRiSM: Construction Systématique de Programmes Parallèles et Distribués.
http://www.prism.uvsq.fr/french/parallel/paf/autom_fr.html.

[Qui98] *Quid 98.* éditions Laffont, 1998.

[Red98] Xavier Redon. CIPOL: polyhedron calculator. Technical report, Laboratoire d'Informatique Fondamentale de Lille, 1998.

[Ros] Serge Rossi, Histoire de l'informatique. BurpTeam.
http://linux-kheops.org/burpteam/histinfo/histoireinfo.html.

[Ruc98] Denis Ruckebusch. Évaluation des communications générées par un compilateur HPF. Mémoire de DEA, Laboratoire d'Informatique Fondamentale de Lille, 1998.

[Sag] Overview of pC++/Sage++. University of Indiana.
http://www.extreme.indiana.edu/sage/overview.html.

[Sev] Eric Sevault, Fortran2html.
http://www.meteo.fr/perso/eric.sevault/Fortran2html/index.html.

[Thi91] Thinking Machines Corporation. *CM Fortran Reference Manual, version 1.0.* Cambridge, Mass.: MIT Press, 1991.

[Top] TOP500 Supercomputer Sites. Netlib.
http://www.top500.org.

[Tra] Trapper. Genias Software.
http://www.genias.de/products/trapper/trap_index.html.

[Vam] Vampir. Pallas Gmbh.
http://www.pallas.de/pages/vampir.htm.

[Vui98] Jean Vuillemin. Reconfigurable Systems, Past and Next 10 Years. In *VecPar'98*, Porto, Portugal, 1998.

[Wil86] K. G. Wilson. Basic Issues of Computational Science. In *International Conference In Computational Physics*, Trieste, 1986.

[Wil97] M. Williams. *A History of Computing Technology.* IEEE Comp. Soc., 1997.

[XPV] XPVM: A Graphical Console and Monitor for PVM.
http://www.netlib.org/utk/icl/xpvm/xpvm.html.

Table des figures

1.1 Évolution des ordinateurs parallèles. 21
1.2 Évolution des architectures. 22
1.3 forall sans sémantique. 33
1.4 Sémantique du forall en FORTRAN 95. 33
1.5 Les générations de FORTRAN. 36
1.6 Effets des placements en charge et en communications. 37
1.7 Exemple d'alignement. 38
1.8 Autre exemple d'alignement. 39
1.9 Exemples de distributions. 40

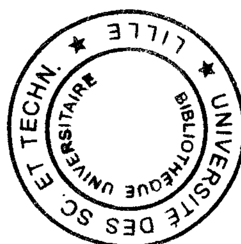
2.1 Description de flux de tableaux en ARRAY-OL. 49
2.2 Construction d'application avec INTERFACE BUILDER. 51
2.3 L'environnement RHIDE. 53
2.4 Application créée avec AVS/EXPRESS et exemple d'utilisation de l'interface de programmation. 55
2.5 Visualisation de passages de messages avec XPVM. 57
2.6 Visualisation d'activités et de passages de messages avec VAMPIR. 58
2.7 Navigation dans un programme avec FORESYS. 60
2.8 Visualisation de données avec DAQV. 63
2.9 Visualisation de données avec ANNAI/DDV. 64
2.10 Statistiques sur les messages avec PABLO. 66
2.11 Graphes de dépendances de données et d'appels avec ANALYST. 68

3.1 Informations à visualiser. 74
3.2 Niveau global. 76
3.3 Niveau hiérarchique. 77
3.4 Niveau directive. 78
3.5 Niveau localité. 78

3.6 Organisation autour d'un serveur d'informations. 80
3.7 Organisation de l'interface graphique. 81
3.8 Insertion de fichiers dans un projet. 82
3.9 Niveau 1 de la visualisation. 83
3.10 Niveau 2 de la visualisation. 84
3.11 Représentation des directives dynamiques. 85
3.12 Niveau 3 de la visualisation. 86
3.13 Niveau 3 de la visualisation (suite). 87
3.14 Niveau 4 de la visualisation. 88
3.15 Niveau 4 de la visualisation (suite). 89
3.16 Visualisation au-delà de 3 dimensions. 91
3.17 Visualisation d'objets très nombreux. 92
3.18 Organisation du serveur d'informations. 94
3.19 Ensembles de points manipulés par les calculs d'images. 98
3.20 Ensembles de points manipulés par les calculs de pré-images. 101
3.21 Exemple de pré-image *stride* ∘ *cyclic*. 104

4.1 Exemple de visualisation des communications. 113
4.2 Tableaux stockés sur `P(4,7)`. 121

5.1 Vue de la démonstration. 127


UNIVERSITÉ DES SC. ET TECHN. LILLE
BIBLIOTHÈQUE UNIVERSITAIRE