École des mines de Paris
Centre de Recherche en Informatique

Thèse
présentée
pour obtenir

le grade de Docteur en sciences
Informatique Temps Réel, Robotique et Automatique
de l'École des mines de Paris

par M. **Fabien COELHO**

Sujet :

Contributions à la compilation du *High Performance Fortran*

Soutenue le jeudi 3 octobre 1996, à 14h00

à l'Écoles des mines de Paris, 60, boulevard Saint-Michel, Paris

Devant le jury composé de :

MM.	Robert Mahl	Président
	François Irigoin	Directeur
	Luc Bougé	Rapporteur
	Paul Feautrier	Rapporteur
	Robert S. Schreiber	Rapporteur
	Thomas Brandes	Examinateur
	Luigi Brochard	Examinateur
	Yves Rouchaleau	Examinateur

Contributions to
High Performance Fortran Compilation

PhD dissertation

Fabien Coelho

à M.

Résumé

Cette étude présente un ensemble de contributions à la compilation du *High Performance Fortran* (HPF), un langage à parallélisme de données basé sur Fortran 90. HPF y ajoute des directives pour spécifier le parallélisme disponible et le placement des données souhaité sur une machine parallèle à mémoire répartie.

Le but de la compilation est de traduire un programme HPF à parallélisme et placement explicites, adressage global et communications implicites, vers un modèle de processus indépendants communiquant par passage de messages, donc avec adressage local des données et communications explicites. Il importe bien sûr que le code généré soit non seulement correct, mais aussi particulièrement efficace. Notre approche a consisté à formuler dans un cadre mathématique les problèmes techniques posés par la compilation de HPF, et à s'appuyer ensuite sur des outils génériques pour produire du code optimisé.

Ce document est composé de quatre parties. Après l'introduction qui présente le domaine du calcul scientifique parallèle et ses enjeux, la première partie introduit le langage HPF, qui simplifie les problèmes de programmation rencontrés dans ce domaine. L'adéquation du langage HPF, ou plutôt son inadéquation à l'état de l'art de son époque est soulignée. Les différentes techniques de compilation proposées depuis pour ce type de langage sont revues en détail.

La deuxième partie discute la définition du langage. Nous analysons certains aspects du langage, relevons des déficiences et proposons des solutions. Une technique de compilation simple visant à éviter des extensions de langage pour déclarer les *recouvrements* de tableaux est également présentée. Enfin nous décrivons une proposition formelle d'extension qui permet d'associer plusieurs placements aux arguments d'une routine.

La troisième partie traite des techniques de compilation développées au cours de nos travaux. Nous présentons d'abord un cadre algébrique affine pour modéliser et compiler les communications et les calculs d'une application en HPF. Ce cadre est ensuite appliqué et étendu au cas des entrées-sorties, éventuellement parallèles. La compilation des *replacements* est ensuite décrite, incluant des optimisations pour éviter des communications inutiles. Le code de communication généré profite des opportunités de partage de charge et de diffusions de messages permises par la duplication des données. Il est démontré optimal, et les performances obtenues sont comparées au compilateur HPF de DEC.

La quatrième partie présente les réalisations effectuées au cours de nos recherches. Il s'agit principalement de l'intégration dans le paralléliseur automatique PIPS d'un prototype de compilateur HPF (représentant 25 000 lignes de code), dont nous détaillons l'implantation, les limitations et les extensions. En particulier, ce prototype inclut l'ensemble des directives HPF de placement, ainsi que les directives de parallélisme de boucles. Les communications sont basées sur la librairie domaine public PVM. Des expériences sur réseau de stations de travail utilisant les codes produits par notre prototype sont également présentées, avant de conclure en rappelant nos contributions.

Mots clef : compilation, *High Performance Fortran*, langage parallèle, parallélisme de données, passage de messages, théorie des polyèdres, génération de code, partage de charge, entrées-sorties, replacements, redistributions, réalignements.

Abstract

This study presents our contributions to *High Performance Fortran* (HPF) compilation. HPF is a data-parallel language based on Fortran 90. Directives are used to specify parallelism and data mapping onto distributed memory parallel architectures.

We aim at translating a global addressing implicit communication HPF program into a message passing parallel model, with local addressing and explicit communication. The generated code must be both correct and especially efficient. Our approach is based on a formalization of compilation technical problems into a mathematical framework and on using standard algorithms for generating optimized code.

This document is divided into four parts. After the introduction which presents scientific computing and its challenges, we describe in the first part the HPF language which solves some of the encountered problems. The weak squaring of HPF with the state of the art in compilation at the time of its design is outlined. Compilation techniques developped for HPF are also detailed as the related work.

In the second part, we discuss HPF design issues. We first analyze HPF language design issues, show deficiencies and suggest new solutions. A simple compilation technique to avoid a language extension aiming at explicitly declaring *overlap* areas is described. Finally we present a possible extension to enable several mappings to be associated to subroutine arguments.

In the third part, we present the compilation techniques. First, we describe a linear algebra framework for compiling computation and communication. This framework is applied and extended to possibly parallel I/O. Compilation of HPF remappings is then addressed, including optimizations to avoid useless remappings, and an optimal technique to manage the actual communication, taking advantage of broadcast and load balancing opportunities linked to data replication. Performance comparisons against the Digital HPF compiler are also presented.

In the last part, we present our prototype HPF compiler (around $25,000$ lines of code) developed within the PIPS automatic parallelizer environment. The implementation, including its weaknesses and extensions, is detailed. In particular, our prototype handles all static and dynamic HPF mapping specifications, as well as parallelism directives. The runtime is based on the public domain PVM communication library. Experiments with codes generated by our compiler on a network of workstations are also described. Finally we conclude.

Keywords: compilation, High Performance Fortran, parallel language, data-parallelism, message passing, polyhedron theory, code generation, load balancing, parallel I/O, remapping, redistribution, realignment.

Remerciements

Je tiens tout d'abord à remercier très vivement François IRIGOIN pour m'avoir reçu dans son équipe, il y a longtemps, à l'occasion d'un stage ingénieur à l'étranger qui a atterri à Fontainebleau et m'a donné le goût d'une recherche pragmatique et sérieuse, à son image.

J'ai alors découvert un monde étonnant, des gens intéressants, des activités motivantes. François m'a proposé un sujet de thèse passionnant, avec en plus un outil incomparable pour mettre en pratique ces recherches, le paralléliseur PIPS, qui est pour beaucoup dans leur aboutissement. J'ai eu avec François la liberté nécessaire pour mener à bien ces travaux, avec en sus un ensemble de conseils prodigués par mon directeur de thèse dont je n'ai entrevu la pertinence que bien plus tard. Il faut dire que François a un talent incomparable pour avoir toujours raison.

Je tiens ensuite à remercier le Prof. Robert MAHL, directeur du centre qui m'a accueilli. Il a été mon professeur puis directeur avec efficacité, m'a soutenu, m'a proposé des activités annexes intéressantes à plus d'un titre, m'a prêté ses élèves de nombreuses heures et enfin a accepté de présider ce jury.

Je veux remercier tout particulièrement mes rapporteurs de thèse : Luc BOUGÉ, Paul FEAUTRIER et Robert SCHREIBER.

Tout d'abord pour avoir accepté de participer à ce jury à une place qui nécessite beaucoup de travail. Ensuite, pour avoir lu ce document éminemment perfectible en beaucoup de détails, et avoir bien voulu comprendre au-delà d'explications parfois embrouillées, la substantifique moelle que j'ai voulu y mettre. Ils ont aussi contribué grâce à leurs nombreuses remarques et leurs suggestions à son amélioration.

Le Prof. Luc BOUGÉ est l'un des moteurs de l'étude du parallélisme de données en France. Il a organisé beaucoup de ces réunions passionnantes du GDR Paradigme, qui m'ont permis de rencontrer tout un petit monde sympathique et vivant qui fait notre communauté, et où l'on apprend plein de choses à propos de domaines proches et connexes.

Le Prof. Paul FEAUTRIER a été mon professeur de DEA. Il a eu alors de don de rendre simple et accessible son domaine. Comme chercheur, il est incontournable. Ses idées, son esprit précis et clair, ses contributions nombreuses et variées mais toujours avec la même rigueur scientifique en font l'un des piliers de ce domaine en France et dans le monde.

Le Dr. Robert SCHREIBER est l'un des principaux animateurs du Forum qui a défini le langage HPF. Ses travaux de recherche portent autant sur la spécification du langage que sur les techniques de compilation. Il a même accepté de prendre en compte certaines de mes suggestions, et nos discussions ont toujours été pointues et argumentées.

Je souhaite également remercier, avec force, mes examinateurs :

Le Dr. Thomas Brandes a lu cette thèse rédigée dans deux langues qui lui sont étrangères. Ses travaux sont proches des miens, puisqu'il développe lui aussi un compilateur HPF, mais avec une approche différente et complémentaire de la mienne. Son opinion sur mes travaux est donc précieuse. En plus de mon amitié, il a toute ma reconnaissance pour avoir accepté de participer à ce jury.

Le Dr. Luigi Brochard fait beaucoup pour HPF puisqu'il en propose l'utilisation pour le calcul scientifique aux clients d'IBM. Il est donc à la fois au contact des produits, des applications et des utilisateurs. Son éclairage particulier lié à son expérience dans le domaine du calcul scientifique parallèle est une contribution importante à ce jury.

Enfin je tiens à remercier le Prof. Yves Rouchaleau, qui a été mon professeur, d'avoir accepté d'honorer de sa présence ce jury dans un domaine qui n'est pas exactement le sien. Mais ses connaissances étendues en mathématique et en algébre, son expérience des outils mathématiques impliqués, en font un oeil à la fois neuf et incisif utile et intéressant pour cette soutenance.

Je tiens aussi à porter une attention toute spéciale aux gens qui ont dû travailler directement avec moi, c'est à dire me supporter.

D'abord un grand merci à Corinne Ancourt, ma chère voisine de bureau, qui a partagé avec moi la rédaction longue et difficile d'un article et de nombreuses discussions autour d'un café à chaque fois généreusement fourni. Sa bonne humeur, ses idées, ses compétences mathématiques, ses opinions et ses remarques me sont très précieuses.

Sans Ronan Keryell, spécialiste à ses heures de LaTeX, amateur éclairé de typographie, ce document n'aurait jamais été ce qu'il est. J'ai profité également de son travail au sein de PIPS et des interfaces pour faire des démonstrations attrayantes de mes travaux. Enfin je tiens à le remercier particulièrement pour m'avoir prêté le chapitre de description globale de PIPS.

Un grand merci à Béatrice Creusillet, qui a du supporter en moi un utilisateur exigeant de ses analyses de régions de tableaux, sans compter les discussions serrées sur les questions de formalisation, et sur le reste.

Pierre Jouvelot est le relecteur et critique infatiguable du centre. Sa maitrîse de l'anglais, son esprit vif et acéré, ses remarques directes, ses opinions tranchées et incisives sur les sujets techniques et autres réveillent, animent ou bousculent la vie du centre, et auront beaucoup contribué à améliorer la qualité de mes travaux.

Jacqueline Altimira est de loin la meilleure secrétaire du centre, même si il n'y en a qu'une. Elle subit avec sourire et bonne humeur et à longueur d'année des chercheurs doublés d'informaticiens, deux catégories de fous bien connues. Elle est l'as du remplissage des papiers administratifs que nous autre simples scientifiques avons souvent du mal à comprendre. Grâce à elle les frais de missions ne sont plus une épreuve, mais au contraire une joie.

Je souhaite également remercier vivement de leur collaboration avec moi Cécile Germain (LRI/Orsay) et Jean-Louis Pazat (IRISA/Rennes). Le développement et la mise au point du cours PRS des Ménuires sur la compilation de HPF aura été l'occasion de nombreux débats au sein du groupe, pour atteindre plus de précision, de généralité et d'efficacité. De la même manière je souhaite remercier pour ses remarques constructives, précises et motivées lors de notre rédaction de la proposition de directive `assume` Henry ZONGARO (IBM Canada).

Je tiens également à remercier les autres membres du CRI, en particulier Jean-Marie KESSIS, Marie-Thérèse LESAGE et Annie PECH qui font fonctionner la bibliothèque, ainsi que Laurent DAVERIO et Nadine OLIVIER qui font fonctionner le système. Je suis également reconnaissant au Prof. Michel LENCI, ancien directeur du centre, pour m'avoir enseigné puis accueilli dans son centre.

Je voudrais aussi remercier les gens qui m'ont donné accès à des machines parallèles chères sans me faire payer, sous reserve que je n'en abuse pas trop quand même, et m'ont aidé à les utiliser correctement : Serge PETITON, Rock BOURDONNAIS, Jean-Luc DEKEYSER, Philippe MARQUET et Serge ALGAROTTI.

Je souhaite remercier pour leur soutien de chaque instant et aux moments plus difficiles ou drôles, mon copain de régiment le conducteur Xavier REDON, mes amis Jean-Philippe, Kélita, Sandrine, etc. Enfin, une mention particulière à mon papa, ma maman et ma femme Vanina qui ont du me supporter au quotidien.

Préambule

Je présente ici quelques éléments qui vous permettront de mieux comprendre l'esprit de ce document, tant sur la forme que sur le fond.

La rédaction

Après quelques années de travail sur un sujet de recherche, il est usuel que le thésard s'adonne à une activité appelée *rédaction de thèse*. Au moment où j'allais me consacrer à cette tâche, un certain nombre de choix se sont offerts à moi : la langue principale de rédaction d'une part, la forme que devait ou pouvait prendre ce travail d'autre part.

Pour ce qui est de la langue, j'aime beaucoup la mienne. Elle m'est particulièrement intuitive. Cependant, chacun souhaite que ses travaux aient le retentissement maximum. Le lectorat potentiel étant, vu le sujet, essentiellement anglo-saxon, l'anglais s'impose au contraire comme unique moyen de communication. Par ailleurs quelques textes législatifs règlent l'emploi de la langue, et l'établissement qui nous accueille conduit sa propre politique en la matière. Vus ces contraintes et souhaits, la solution naturelle est de proposer un document rédigé dans l'une et l'autre langue : à la fois *Molière* (un quart) et *Shakespeare* (trois quarts) – toutes proportions gardées bien sûr !

La seconde question touche à la forme du travail. Trois années de thèse ont été l'occasion de nombreuses réflexions qui ont donné lieu à la rédaction de rapports et articles. Chacune de ces sources est achevée à sa manière, a trouvé un équilibre dans son contexte, entre précision et concision. Par ailleurs elles sont les pierres cohérentes d'un travail morcelé dans le temps et par les sujets abordés, mais dont la finalité unique est de servir le calcul scientifique parallèle. Enfin elles ont été développées, rédigées et influencées par différentes personnes. Je ne souhaite pas diminuer leur unité et pluralité en les diluant, les dénaturant dans un dessein qui serait ma thèse. Je préfère les garder chacune à leur place, en les enveloppant pour montrer sans les affecter leur cohérence et articulations.

Les travaux

Il s'agit donc d'une *thèse sur travaux*, qui unifie un ensemble de contributions poursuivies et développées au cours de mes recherches. Il existe un fil historique à ces développements. Je vais le narrer ici avant d'en présenter une vue plus logique.

Tout a commencé à l'été 1992 où j'ai atterri dans le bureau de mon futur directeur de thèse, François IRIGOIN. Je me suis alors occupé en *suivant* les discussions électroniques du Forum sur la spécification du langage HPF, et en lisant des articles ayant rapport aux problèmes de compilation pour machine à mémoire répartie. Au cours de l'année 1992-1993, j'ai continué de suivre ces développements et j'ai implanté une première mouture de prototype de compilateur incluant simplement la méthode inefficace de résolution dynamique des accès aux variables réparties pour un simple programme. J'ai aussi fait une présentation du langage pour une réunion du GDR Paradigme. Je la compléterai deux ans plus tard pour en faire un tutoriel à Renpar'7.

L'été 1993 a ensuite été consacré à mon stage de DEA où j'ai implanté des méthodes d'optimisation courantes (analyse des recouvrements et vectorisation des communications, ou encore calculs de réductions en parallèle), complété ma bibliographie et effectué des tests sur réseau de stations de travail. Ces travaux ont fait l'objet d'une communication à HPCN'94. Elle couvre les mesures effectuées sur réseau de stations de travail et un modèle pour en évaluer la pertinence. Ce modèle, validé par les expériences, permet également de mesurer l'influence sur les performances des différents paramètres et hypothèses. En particulier, il montre l'importance du nombre de processeurs et de la séquentialisation des communications sur l'efficacité des calculs.

Ensuite, la fin de l'année 1993 a phosphoré fort pour intégrer les compétences mathématiques locales dans le but de compiler du HPF, en abordant pour les problèmes réguliers la modélisation, l'allocation, l'adressage, les communications et les calculs. La modélisation sous forme de systèmes de contraintes est assez naturelle pour les problèmes réguliers, et parce que la répartition des données en HPF s'y prête particulièrement. Cependant la présence d'égalités dans les systèmes implique un Z-module sous-jacent, ce qui est de nature à troubler les algorithmes de génération de code d'énumération. Des transformations des systèmes visant à densifier le polyèdre ont donc été proposées. Le résultat a fait l'objet d'une communication à CPC'93, qui sera transformé après de nombreuses corrections améliorations et autres deboguages en un article dans *Scientific Programming*, trois ans plus tard.

Si l'on pense naïvement que la théorie et l'abstraction règlent tous les problèmes pratiques, il ne restait alors plus qu'à implanter. C'est cependant à cette occasion qu'on découvre les problèmes facilement minimisés ou oubliés, mais qui ouvrent de nouvelles voies de recherche. La stratégie suivie a été d'implanter l'approche décrite en l'incluant de manière incrémentale dans mon prototype. J'étais par ailleurs bloqué par les entrées-sorties sur une application réelle de propagation d'onde sismique, ce qui m'empêchais d'effectuer des essais sur la CM5 du SEH. Dans l'optique d'implanter une technique à base de polyèdre, et commme le cas des entrées-sorties est simplifié (communications de 1 à n), mais présente des parties communes nécessaires (construction des systèmes de contraintes, génération de code, fonctions de support de l'exécutif), il était naturel de commencer ainsi la programmation de méthodes de compilation avancées.

Ces développements se sont déroulés au début de l'année 1994. Par rapport à la méthode général suggérée, d'autres questions ont été abordées. D'abord le problème de l'exactitude ou non des analyses utilisées, et de leur impact sur le code généré. Ensuite, l'intégration des problèmes d'adressage locaux et globaux de manière directe dans le processus de compilation, en reprennant la technique de compression de l'allocation que j'avais initialement développée dans mon compilateur, et non la nouvelle basée sur les transformation de Hermite. Je me suis également convaincu que la duplication au moins partielle des données était un cas courant et intéressant. Finallement, j'ai du décider de la place de l'insertion des communications dans le code. Les tests sur la CM5 qui avaient motivé ce travail s'avéreront décevant pour des raisons diverses : PVM perdait des messages de temps en temps d'une part, et l'essentiel du temps d'exécution se passait dans la gestion très lente des exceptions arithmétiques générées par l'application. Ces travaux sur la compilation des entrées-sorties ont fait l'objet d'une communication à *Frontiers'95*.

La fin de l'année 1994 a été motivée par une application du KFA de simulation de croissance de cristaux de silicium par un maillage 3D. Le code comportait différentes procédures partageant des données cises dans des *commons* Fortran. J'ai donc été conduit à gérer nécessairement ce cas, donc étendre mon compilateur pour gérer des sous-routines. De plus le code présentait un motif de communication particulier pour la mise à jour des bouts d'un torre à chaque itération (décalage d'un sous ensemble de tableau). J'ai implanté simplement la compilation de ce motif de communication sous forme de reconnaissance de forme et d'appel à des fonctions *ad hoc* de l'exécutif. Je n'ai pas fait tourné le code en parallèle parce que la collaboration initiallement envisagée a un peu tourné court. J'ai simplement fait quelques expériences avec le code séquentiel, et ai gagné environ un tiers du temps d'exécution en quelques optimisations élémentaires favorables au cache. L'impact sur mon prototype aura été très significatif, puisqu'il sera alors mieux intégré dans PIPS et acceptera des programmes bien plus réalistes avec des sous-routines.

Le second pas dans l'implantation de méthodes polyédriques était, naturellement aussi, de gérer les replacements de HPF, question que j'abordais le long de l'année 1995. En effet, les communications sont cette fois générale, mais il n'y a pas à se préoccuper des références et autres détails, il s'agit de déplacer un tableau en entier. J'étais aussi motivé pour faire quelque chose à propos de la duplication, sentant bien qu'il y avait à chaque duplication une opportunité pour profiter de la redondance. D'un côté, la diffusion d'un même message vers plusieurs processeurs qui attendent les mêmes données permet une factorisation naturelle des coûts. De l'autre côté, chaque groupe d'envoyeurs possédant les mêmes données peut se partager la charge de l'envoi de ces données, parallélisant ce travail autant que la duplication disponible. Ces considérations supplémentaires se sont parfaitement intégrées dans l'approche compilation des communications à base de systèmes de contraintes, sous forme d'une équation additionnelle de partage de charge et de projections sur les dimensions de diffusions.

Cela étant décidé, l'implantation effective de ces algorithmes de compilation des codes de communications dans mon prototype de compilateur n'aura pas été une mince affaire. Elle impliquera de profonds changements dans la structure du compilateur et dans la manière de gérer les directives, ce qui améliorera encore son intégration au sein de PIPS. Le premier problème était de retrouver la place des directives de replacement dans le code, ce qui était difficile avec des analyseurs syntaxiques distincts pour le code et les directives. De plus, le problème de la gestion effective de l'allocation et de l'adressage de ces tableaux déplacés devait être abordé. En me consacrant à ce point, il est apparu rapidement que l'idée naturelle était de se ramener à une version statique du programme avec des recopies entre tableaux placés statiquement. Cette approche a ouvert tout un pan de réflexion sur les optimisations de flot de données possibles quand plusieurs versions d'un tableau sont disponibles (par exemple, pourquoi détruire l'ancienne version, qui peut s'avérer utile par la suite ?), mais aussi du côté de la spécification du langage, qui ne permet pas ce type de compilation simple et efficace dans le cas général. Je me suis alors interrogé sur les raisons floues qui avaient conduit à ces choix, et leurs conséquences indirectes mais coûteuses sur le développement des compilateurs commerciaux. Bref, alors que je prennais initialement comme sujet de thèse le langage comme une donnée, je me suis dit qu'il était aussi nécessaire de le modifier pour permettre des implantations efficaces.

J'ai donc implanté ces techniques dans mon compilateur, puis me suis lancé dans la rédaction d'un article en collaboration avec Corinne Ancourt pour décrire ces travaux effectués en commun. En écrivant l'article, la question de l'optimalité éventuelle des communications a été abordée, et j'ai modifié l'algorithme de manière a être optimal à la fois en terme de volume de données transférées qu'en nombre de messages. Des tests avec les codes générés pour les replacements ont été faits avec la ferme d'alpha du LIFL, qui était un peu chargée, mais pas trop pour que quelques bonnes mesures reflétant l'intérêt de ces travaux puissent être enregistrées. Les travaux sur la compilation des communications liés aux replacements, l'optimalité obtenue et les expériences feront l'objet d'une publication dans JPDC en 1996. L'aspect transformation d'un programme au placement dynamique en un programme au placement statique avec des recopies entre tableaux, et les optimisations associées n'est par contre (encore) qu'un rapport interne.

Le glissement fondamental opéré cette dernière année, a été de s'intéresser plus directement à la définition du langage, à la lumière de mes expériences en compilation. J'ai commencé à remettre en cause certains aspects du langage que je ne faisais que critiquer auparavant, et qui représentent intrinsèquement des obstacles à une bonne et simple compilation. Mon activité constructive sur les listes de discussion du Forum s'est trouvée accélérée et démultipliée à un moment où la définition de la nouvelle version du langage était à la fois flottante et hésitante, recherchant des simplifications abusives d'aspects pourtant utiles, tels les replacements, mais gardant obstinément des points inutiles comme les placements transcriptifs. Sans compter quelques extensions de langage douteuses comme la déclaration explicite des recouvrements lors de la déclaration du placement des tableaux. J'ai donc milité pour quelques corrections au niveau de la syntaxe, en parvenant à faire changer la définition de la directive `reduction` pour un style *explicite*, mais aussi pour des révisions plus fondamentales, à savoir la simplification plutôt que l'abandon des directives de replacement, et le remplacement des placements transcriptifs proposés par `inherit` en une directive `assume` qui donne quelque information au compilateur. Sur ces derniers points je dois reconnaître que mon influence aura été faible puisque je ne suis pas parvenu à faire modifier le langage. Le résultat de cette activité sera une communication à *EuroPar'96* où je discute les problèmes de conception de HPF.

Enfin l'hiver et le printemps de l'année 1996 a vu une part importante de mon activité consacrée à la préparation d'un cours écrit et oral sur la compilation de HPF pour l'école de printemps PRS sur le parallélisme de données, en commun avec Cécile Germain et Jean-Louis Pazat. Ce cours a apporté la partie *autres travaux* de ma thèse en offrant un survol précis – au risque d'être parfois touffu – des techniques de compilation envisagées pour les langages comme HPF. Il m'aura aussi permis de remettre à jour ma bibliographie. Le printemps de l'année verra la rédaction de cette thèse qui rassemble mes travaux.

Tout en développant mon prototype de compilateur, j'ai amélioré les algorithmes de génération de code de parcours de polyèdre pour en parfaire le déterminisme (générer toujours le même code pour un même système de contraintes), et pour simplifier et optimiser quand possible le code en profitant des informations statiques disponibles. D'un point de vue interne au projet PIPS, j'ai aussi développé un itérateur général `gen_multi_recurse` sur les structures de données qui permet le codage rapide et efficace de nombreuses transformations ou analyses de ces structures. Cet itérateur généralise et optimise l'itérateur initialement codé par Pierre Jouvelot et Rémi Triolet. J'ai révisé ou étendu les outils de développements et de validation : par exemple les tests de non-régression du compilateur sont effectués chaque nuit en parallèle sur plusieurs machines. J'ai aussi travaillé à restructurer l'organisation générale du projet PIPS, modifications nombreuses et incrémentales dont ont souvent eu à souffrir les utilisateurs locaux, avec comme mince satisfaction qu'il s'agissait surement d'un progrès. Des améliorations de robustesse, de portabilité ou encore de précision du compilateur ont eu lieu, en particulier à l'occasion de la démonstration de PIPS au stand Paradigme de *SuperComputing'95*. J'ai aussi été amené à assurer de nombreuses heures d'enseignements à diverses occasions. Tous ces travaux ne trouvent pas ou peu d'échos dans cette thèse qui se focalise sur mon travail de recherche directement lié au langage HPF.

La thèse

Ces travaux sont nombreux et variés, mais il n'est pas difficile de les fondre dans une perspective cohérente et pleine. Ils se fédèrent naturellement en catégories qui tournent autour, en amont comme en aval, des techniques de compilation pour machines à mémoire répartie, en s'appuyant particulièrement sur l'exemple du *High Performance Fortran.*

À l'amont, je discute les critères de choix et les décisions opérées au niveau de la définition du langage. Cette définition a été faite au–delà de l'état de l'art de son époque [3, 5]. Elle peut encore être perfectionnée pour devenir plus utile, plus propre et plus facilement et efficacement implantable [9]. Certaines améliorations ou adjonctions ont été proposées pour lesquelles il existe d'autres solutions [10] si on s'autorise un peu d'imagination. D'autres idées [13] ont fait l'objet d'une proposition formelle au Forum qui organise la spécification du langage.

Au cœur du problème, donc de mon travail, se trouvent les techniques de compilation, sans lesquelles un langage qui vise aux hautes performances n'aura pas d'objet. Ces techniques s'appuient sur l'algèbre linéaire [1, 2] pour résoudre globalement les problèmes d'affectation des calculs aux processeurs, de génération des communications, d'allocation et d'adressage en mémoire. Certains cas particuliers se prêtent à des optimisations plus poussées mais dans un cadre plus simple, comme les communications liées aux entrées-sorties [6] ou à la modification dynamique du placement [8, 11].

En aval, la question de l'implantation effective et des expériences afférentes [14, 4] consomme beaucoup d'énergie, mais est source de satisfactions plus pratiques et réelles. De plus, elle permet de mettre à jour des problèmes ignorés jusque là et suggère donc naturellement de nouvelles voies de recherche et d'approfondissement. Enfin, l'enseignement du langage [7] ou des techniques de compilation [12] permet de reformuler, comparer, préciser et affiner ses propres techniques comme celles d'autres.

Ces différents travaux sont repris tout au long de ce mémoire sous forme de chapitres indépendants les uns des autres. En tête de chaque chapitre je rappelle les auteurs ayant participé à sa rédaction, ainsi que l'état courant du document en ce qui concerne sa publication éventuelle. Chaque chapitre est également précédé d'un résumé (en français) et d'un *abstract (in English)*. Les différents travaux sont fédérés en parties par grands thèmes : présentation, conception, compilation et implantation du langage. Chaque partie est résumée en détail et en français par un chapitre introductif qui présente les principaux résultats. Enfin une introduction et une conclusion générale à ces travaux, qui les remettent en perspective, encadrent le document.

Bibliographie personnelle

[1] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static HPF Code Distribution. In *Workshop on Compilers for Parallel Computers, Delft*, December 1993. To appear in Scientific Programming. Available at `http://www.cri.ensmp.fr`.

[2] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static HPF Code Distribution. *Scientific Programming*, ??(??):??–??, ?? 1996. To appear.

[3] Fabien Coelho. Compilation du *high performance fortran*. In *Journées du Site Expérimental en Hyperparallélisme*, pages 356–370, January 1994.

[4] Fabien Coelho. Experiments with HPF Compilation for a Network of Workstations. In *High-Performance Computing and Networking, Springer-Verlag LNCS 797*, pages 423–428, April 1994.

[5] Fabien Coelho. HPF et l'état de l'art. TR A 260, CRI, École des mines de Paris, July 1994.

[6] Fabien Coelho. Compilation of I/O Communications for HPF. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 102–109, February 1995.

[7] Fabien Coelho. Présentation du *high performance fortran*. Tutoriel HPF – Transparents, Renpar'7, Mons, Belgique, May 1995. Aussi journée PARADIGME (Apr 93), séminaire de l'IDRIS (Jun 94), journée du CEA (Feb 96). `http://www.cri.ensmp.fr/~coelho`.

[8] Fabien Coelho. Compiling dynamic mappings with array copies. TR A 292, CRI, École des mines de Paris, May 1996.

[9] Fabien Coelho. Discussing HPF Design Issues. In *Euro-Par'96, Lyon, France*, pages 571–578, August 1996. Also report EMP CRI A-284, Feb. 1996.

[10] Fabien Coelho. Init-time Shadow Width Computation through Compile-time Conventions. TR A 285, CRI, École des mines de Paris, March 1996.

[11] Fabien Coelho and Corinne Ancourt. Optimal Compilation of HPF Remappings. CRI TR A 277, CRI, École des mines de Paris, October 1995. To appear in JPDC, 1996.

[12] Fabien Coelho, Cécile Germain, and Jean-Louis Pazat. State of the Art in Compiling HPF. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, pages 104–133. Springer Verlag, first edition, September 1996. Proceedings of the PRS Spring School, Les Ménuires, March 1996. Also TR EMP CRI A-286.

[13] Fabien Coelho and Henry Zongaro. `ASSUME` directive proposal. TR A 287, CRI, École des mines de Paris, April 1996.

[14] Ronan Keryell, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, and Pierre Jouvelot. PIPS: A Framework for Building Interprocedural Compilers, Parallelizers and Optimizers. Technical Report 289, CRI, École des mines de Paris, April 1996.

Table des matières

Table des figures

Introduction

Le calcul scientifique

L'évolution du calcul scientifique est guidée par des impératifs économiques, techniques et sociologiques :

- les coûts des calculateurs pour une même puissance baissent de manière constante ;
- à coût constant, les puissances proposées sont de plus en plus grandes ;
- les métiers s'adaptent à ces outils et les incorporent dans les processus de développement.

Pour accompagner cette progression des hautes technologies, les grands pays industrialisés ont développé des programmes de recherches, comme le HPCC aux États-Unis ou le HPCN en Europe. Ces grands programmes visent à soutenir les technologies indispensables au développement des supercalculateurs et réseaux. Ils se fédèrent autour d'objectifs médiatiques et symboliques comme le *Teraflops*, les *Grand Challenges* ou autres records.

Les supercalculateurs

L'histoire des supercalculateurs est une longue suite de progrès aboutissant à l'augmentation constante et exponentielle des puissances de calcul proposées. Grâce à la miniaturisation et à la spécialisation des circuits, l'accélération des horloges des machines [129] a permis de soutenir cette progression durant plusieurs décennies. Cependant les limites physiques sont en passe d'être atteintes, et d'autres voies doivent être explorées pour continuer cette progression. Pour ce faire, l'idée naturelle est le parallélisme, à savoir faire plusieurs opérations de manière concurrente, soit à l'intérieur même du processeur, soit entre processeurs coopérant à une même tâche.

Parallélisme intraprocesseur

À l'intérieur d'un processeur, plusieurs techniques permettent d'exploiter du parallélisme : la création d'unités fonctionnelles indépendantes, leur duplication, et leur caractère éventuellement pipeliné (ou étagé).

- un processeur est composé d'unités fonctionnelles (chargement des instructions, décodage, chargement des opérandes, opérations arithmétiques ou flottantes). Sous réserve que le programme ou le circuit de contrôle du processeur assure le respect des dépendances (c'est–à–dire qu'il attend bien que les résultats nécessaires soient disponibles avant de lancer une opération, et veille à ne pas écraser les résultats attendus par d'autres), ces unités peuvent opérer de manière parallèle. Les différentes unités peuvent être reliées l'une après l'autre et ainsi assurer un fonctionnement en pipeline (le décodage de l'instruction suivante commence par exemple alors que la récupération des opérandes d'une autre instruction est en cours). Elles peuvent éventuellement fonctionner directement en parallèle, plusieurs opérations étant lancées à un instant donné.

– une unité fonctionnelle donnée peut aussi être étagée en interne, de manière à décomposer une opération en sous-opérations indépendantes ; l'opération élémentaire suivante peut alors commencer alors que le résultat de la précédente est encore en cours de calcul mais à un autre étage.

Au niveau des jeux d'instructions, on distingue les processeurs :

super scalaire et VLIW : ils peuvent démarrer plusieurs instructions à chaque cycle, ce qui permet d'exploiter à plein le parallélisme entre unités fonctionnelles disponibles, et d'augmenter le nombre d'instructions exécutées par cycle ; des techniques de compilation sont utiles, particulièrement au niveau des boucles [216, 101, 249] ; les instructions VLIW *(Very Long Instruction Words)* décident statiquement du parallélisme.

Cela correspond à un micro-parallélisme d'instructions (de toutes petites tâches donc).

vectoriel : les opérandes sont des vecteurs, et les opérateurs s'appliquent à des vecteurs et non à des scalaires ; ces instructions utilisent souvent des unités fonctionnelles pipelinées ; cette technique diminue le flot d'instructions à exécuter.

On a là un micro-parallélisme de données.

Cependant cette progression de la puissance de calcul ne doit pas occulter la nécessaire progression de la vitesse d'accès à la mémoire, afin d'alimenter ces unités fonctionnelles [129, 108]. De plus le parallélisme intraprocesseur n'est pas suffisant pour permettre la progression souhaitée.

Parallélisme entre processeurs

L'idée de faire coopérer plusieurs processeurs en parallèle remonte au moins aux années cinquante [84, 244, 227, 251]. L'attrait principal est son extensibilité *(scalability)*, mais la mise en pratique pose de nombreux problèmes particulièrement quand la mise en œuvre remonte au niveau des applications. De nombreuses taxinomies ont été proposées pour désigner, classer et comparer les machines multiprocesseurs [23, 92, 93, 142, 217, 123]. Nous retenons celle de Flynn, doublée de considération sur le modèle mémoire au niveau matériel.

Flot de contrôle : simple (SI, *Single Instruction*) ou multiple (MI, *Multiple Instruction*) ; tous les processeurs exécutent le même programme de manière synchrone, ou bien des programmes différents mais coordonnés.

Flot de données : simple (SD, *Single Data*) ou multiple (MD, *Multiple Data*) ; les instructions portent sur des données distinctes ou non.

Mémoire : physiquement partagée (SM, *Shared Memory*) ou répartie (DM, *Distributed Memory*) ; les processeurs accèdent directement ou non à une mémoire globale.

L'architecture la plus simple et la plus extensible au plan matériel est la MIMD-DM : des processeurs indépendants exécutent leur propre programme sur leurs propres données. Cependant la problématique reste alors entière : comment faire travailler sur une même application des processeurs indépendants ? À un certain niveau le caractère réparti de la mémoire doit faire place à une vue synthétique et homogène, en proposant des données logiquement partagées. Cette thèse aborde les problèmes liés à la programmation d'applications sur ce type d'architecture.

Solutions

Différentes solutions sont envisageables, qui font porter la gestion du partage des données, et donc des communications associées, à différents niveaux :

matériel : la première solution est de proposer un accès uniforme à la mémoire au niveau du matériel qui assure la récupération des données lors d'accès non locaux ; c'est la mémoire partagée, qui souffre de problèmes d'extensibilité.

système d'exploitation : peut gérer le partage ; une solution élégante est de proposer un mécanisme de pagination partagée entre les processeurs [180, 13, 198, 29, 193] ; cette voie a été suivie par feu le constructeur KSR [197, 175].

compilateur : peut être chargé de la détection du parallélisme et de son exploitation à partir de l'application écrite par exemple en Fortran ; ces techniques nécessitent des analyses de programme avancées [239, 44, 86, 20, 259, 127, 18, 76, 75, 73, 176] ; le parallélisme est ensuite extrait [146, 167, 240, 138, 87, 88, 177, 219, 218, 256] ; le placement des données décidé [160, 159, 90, 8, 134, 81, 89, 9, 114, 222, 124, 151, 54, 21, 208, 207] et le tout est ensuite exploité [79, 261, 105, 106, 107, 190, 131, 2, 5, 21].

langage : il peut lui–même spécifier le parallélisme et la répartition des données [78], il ne reste alors au compilateur que le problème de la traduction vers la machine ; cette solution qui est à la base des travaux de cette thèse est détaillée ci-après.

utilisateur : il gère en dernier ressort le problème de la répartition des données et des communications [85] ; des libraires d'échange de messages comme PVM ou MPI fournissent les fonctions élémentaires de communication [243, 103, 83, 22] ; des libraires plus avancées comme ScaLAPACK [82] proposent des services de plus haut niveau incluant des calculs.

Tout cela pour essayer de transformer le parallélisme d'une application en accélération effective, la progression étant de toute façon bornée par la loi d'AMDAHL [3].

Langages parallèles

L'une des solutions envisagées pour faciliter la programmation des machines parallèles à mémoire répartie s'appuie sur un langage de programmation pour spécifier tout ou partie du parallélisme et du placement, autrement laissés à la charge du compilateur ou de l'utilisateur [158]. Ces nouveaux langages sont souvent basés sur un langage de programmation existant, et ajoutent des extensions propres pour préciser le parallélisme et le placement des données.

Base du langage

Un certain nombre de projets se sont focalisés sur la traduction d'un programme déjà parallèle vers un modèle de machine à mémoire répartie [78]. Ces projets se sont basés sur différents langages comme FORTRAN [206] ou C [154] :

Fortran : VIENNA FORTRAN [258, 260, 26, 50], FORTRAN D [99, 131, 192, 132, 241], CM-FORTRAN [237], ADAPTOR [38, 37, 35, 36, 40, 39, 41] ou d'autres [203, 183, 55, 181] qui ont servi de modèle au *High Performance Fortran* [95, 96, 162]

C : PANDORE [12, 10], OCTI [174], POMP C [202], DPC [126, 125], C* [236], HYPER C et d'autres [242, 186, 211].

Autres : MODULA [205, 128], KALI [164, 163], ID NOUVEAU [220, 221], CRYSTAL [179], BOOSTER [199] ...

Le choix de FORTRAN s'impose naturellement dans le domaine du calcul scientifique de par l'existence de standards bien implantés [14, 140]. Il existe cependant d'autres approches comme le langage fonctionnel SISAL [46] qui ne rencontrent pas un écho important dans la communauté.

Extensions de langage

Ces langages permettent à l'utilisateur de préciser du parallélisme au sein de l'application. Ce parallélisme peut être de tâches ou de données. On s'intéresse ici essentiellement au parallélisme de données. Ce modèle de programmation permet d'effectuer des opérations sur des collections de données, typiquement des vecteurs ou des matrices.

Pour spécifier le parallélisme, ces langages proposent essentiellement des boucles dont l'ordre d'exécution n'est pas totalement contraint :

boucle indépendante : elle précise que les itérations sont indépendantes les unes des autres et peuvent être exécutées dans n'importe quel ordre, y compris entrelacé (boucle `independent` de HPF) ;

boucle data-parallèle : elle précise que l'opération peut être appliquée simultanément à tous les éléments d'un tableau par exemple ; elle a une forme implicite avec les expressions de tableaux (en Fortran 90: `A(:)=A(:)+1`), ou bien peut être explicitée (`forall(i=1:n) A(i,i)=1`) ;

masquage : ce procédé permet de sélectionner un sous ensemble d'une collection pour effectuer une opération (`where (A.ne.0) A=1/A`) ;

localisation : certains langages comme Fortran D autorisent l'utilisateur à préciser la répartition des calculs sur les processeurs ;

réductions : cette forme particulière de parallélisme peut être décrite au sein d'une boucle par exemple, ou bien sous forme de fonctions intrinsèques au langage (`s=SUM(A(:,:))`) ; elle peut être ensuite exploitée par le compilateur directement ou bien à travers une librairie.

Pour décrire le placement des données, ces langages disposent de descripteurs spécifiques, qui permettent une plus ou moins grande richesse du placement, richesse qui devra être gérée par le compilateur. Ce placement s'applique généralement à des tableaux, dimension par dimension.

distribution régulière : permet de décrire un placement régulier sur une machine parallèle, typiquement par bloc ou de manière cyclique.

distribution irrégulière : permet une plus grande richesse, par exemple la possibilité d'avoir des blocs de taille différente sur chaque processeur.

alignements : cette technique consiste à décrire le placement d'un tableau relativement à une autre qui sert de référence, et donc de manière indirecte.

dynamicité : le placement précisé peut être modifié au cours de l'exécution du programme. On parlera alors de *replacements*.

La définition du *High Performance Fortran* a repris un certain nombre de ces idées, les a généralisées et standardisées [95, 96, 162]. Ces extensions ont pour l'essentiel pris la forme de commentaires spéciaux, pour assurer la portabilité des codes. Nos travaux ont porté sur ce langage : sa définition et sa compilation.

Contenu de la thèse

Au cours de nos travaux, nous nous sommes intéressé aux problèmes posés par la programmation des machines parallèles asynchrones à mémoire répartie (type MIMD-DM). Nous avons suivi et participé au développement de la solution de type langage de programmation parallèle, qui consiste à décharger l'utilisateur de la mise en œuvre explicite du parallélisme sur une machine à mémoire répartie, mais sans pour autant charger le compilateur de détecter ce parallélisme et de décider du placement des données. Nos travaux ont donc gravité autour des problèmes de compilation des langages de type HPF. Nous sommes remonté à l'amont pour suggérer des extensions ou des améliorations au langage et permettre une compilation plus facile et de meilleures performances. Nous avons montré à l'aval la faisabilité de nos techniques en les implantant dans un prototype de compilateur et en testant les codes générés sur des machines parallèles.

Organisation de la thèse

Cette thèse est découpée en parties regroupant de manière logique différents travaux. Chaque partie est subdivisée en chapitres. Le premier chapitre de chaque partie est une introduction en français aux autres chapitres le plus souvent rédigés en anglais, qui en résume et en présente les principaux résultats. Les chapitres sont généralement indépendants les uns des autres.

Première partie

La première partie se penche sur un type de solution envisagé pour faciliter la programmation des applications sur machines parallèles, à base d'extensions au langage FORTRAN. Le *High Performance Fortran* est présenté en détail, de sa première version aux dernières évolutions encore en discussion (chapitre 1). HPF permet à l'utilisateur d'ajouter à un programme FORTRAN 90 des directives qui spécifient le placement des données et le parallélisme. Il est d'abord montré que la compilation du langage était hors de portée de l'état de l'art de son époque (chapitre 2). Enfin l'ensemble des travaux de recherche s'attachant à la compilation d'un tel langage, formant l'état de l'art de la compilation pour ce type de langage, est exposé (chapitre 3).

Deuxième partie

La deuxième partie présente les aspects en amont des problèmes de compilation, à savoir ceux qui touchent à la définition-même du langage. Elle décrit nos contributions à la définition, ou à l'amélioration de la définition du langage. Un certain nombre de caractéristiques du langage sont discutées, des déficiences sont relevées et des solutions proposées pour régler ces problèmes, qui sont à l'articulation difficile de la définition même du langage, de son implantation et de son utilité pour les applications (chapitre 2). Sur un point particulier pour lequel une extension de langage a été adoptée, afin de déclarer explicitement les zones de recouvrement des tableaux, une technique de compilation simple est décrite pour éviter un recours à l'utilisateur (chapitre 3). Enfin une proposition d'extension au langage HPF est donnée. Elle permet de décrire au compilateur plusieurs placements pour les arguments d'une routine, au bénéfice des performances – le compilateur a plus d'information – et de la qualité logicielle des programmes (chapitre 4).

Troisième partie

La troisième partie entre dans le cœur du sujet, à savoir la compilation du langage. Elle présente des techniques de compilation pour le langage. Ces techniques visent à gérer les données sur une machine parallèle, incluant les communications, l'allocation et l'adressage mémoire pour accomplir les calculs requis. Un cadre général basé sur l'algèbre linéaire, particulièrement la théorie des polyèdres en nombres entiers, est d'abord présenté (chapitre 2). Les communications liées aux entrées-sorties dans un modèle hôte et nœuds, et de possibles extensions pour des entrées–sorties parallèles sont traitées spécifiquement, en s'appuyant sur des analyses de programme précises (chapitre 3). La gestion des replacements de HPF et des optimisations destinées à réduire le nombre de replacements nécessaires à l'exécution sont présentées et discutées (chapitre 4). Enfin les communications liées aux replacements font l'objet d'optimisations particulières pour partager la charge et utiliser des communications spéciales comme les diffusions, et les performances obtenues ont été comparées à celles du compilateur HPF de DEC (chapitre 5).

Quatrième partie

La quatrième partie aborde les questions d'implantation effective et d'expérimentation. La programmation au sein du paralléliseur automatique PIPS d'un prototype de compilateur HPF qui inclut certaines techniques de compilation avancées est présentée. Un survol général de PIPS est d'abord proposé (chapitre 2). Les caractéristiques du prototype et son organisation globale sont ensuite décrites (chapitre 3). Des résultats d'expérimentations de codes simples sur réseau de station de travail sont enfin discutés (chapitre 4).

Première partie

Présentation du *High Performance Fortran*

Chapitre 1

Introduction

Ce chapitre présente le langage HPF. Le contenu correspond aux différents exposés donnés à diverses occasions comme la journée Paradigme sur HPF [57], un séminaire de l'IDRIS (juin 1994), le tutoriel HPF à Renpar'7 [63], ou encore la journée HPF organisée par le CEA (février 1996).

Résumé

Cette partie introduit le *High Performance Fortran*. Ce chapitre présente rapidement le langage HPF et résume en français le chapitre 3. Le chapitre 2 discute de l'inadéquation de HPF à l'état de l'art de son époque, en 1993. Le chapitre 3 passe en revue les techniques de compilation proposées jusqu'à présent. Le langage a été défini dans le cadre d'un Forum regroupant de nombreux intervenants, pour obtenir un langage standardisé dédié à la programmation des machines parallèles à mémoire répartie. Le *High Performance Fortran* issu de ce comité, basé sur Fortran 90, reprend le modèle data-parallèle et y ajoute des directives, constructions et fonctions intrinsèques pour préciser le parallélisme et le placement des données. Le langage est encore en cours de définition, et de nouvelles extensions ainsi que des clarifications et simplifications sont discutées. Nous présentons ici HPF rapidement et renvoyons à l'excellent ouvrage de Koelbel *et al.* [162] pour une description exhaustive du langage.

Abstract

This part introduces the *High Performance Fortran*. This chapter presents the HPF programming language. Chapter 2 discusses HPF weak squaring with the state of the art of its time, in 1993. Chapter 3 reviews compilation techniques suggested so far. HPF has been defined by a Forum aiming at producing a standard language for programming distributed memory parallel machines. The *High Performance Fortran* is based on Fortran 90 and on the data-parallel paradigm. It adds directives, constructs and intrinsics to specify both parallelism and data mapping. The language is still being discussed, both for possible extensions and simplifications. HPF is only outlined in this chapter and we refer to [162] for further technical information.

1.1 Le Forum HPF

La version initiale du *High Performance Fortran* a été développée à partir de janvier 1992 sous l'instigation de DEC et la présidence de Ken Kennedy

(Rice University). Le comité a voté successivement diverses extensions au langage Fortran 90. Ces extensions s'inspirent de différents projets tels Fortran D, Vienna Fortran et CM Fortran.

1.1.1 Composition

Le Forum a regroupé les principaux constructeurs, laboratoires et universités, essentiellement américains, intéressés par le calcul scientifique sur machines parallèles.

Constructeurs : TMC, Convex, IBM, DEC, Sun, nCube, Intel SSD, Archipel, ACRI, PGI, APR ...

Universités : Rice, Syracuse, Cornell, Yale, Vienna ...

Laboratoires : Icase, LANL, LLNL, RIACS, GMD ...

Cette participation large et diversifiée montre le besoin et donc l'intérêt d'une telle entreprise, à un moment où les machines parallèles cherchaient leur marché, et où les utilisateurs voulaient des garanties de portabilité pour assurer la pérennité de leurs investissements.

1.1.2 Objectifs

Au moment de sa constitution, le Forum s'est donné un certain nombre de buts, pour guider ses réflexions. Les principaux objectifs étaient :

Performance : pour les machines MIMD et SIMD, ce qui implique une certaine simplicité du langage.

Ouverture : à d'autres styles de programmation (passage de messages), et à de nouvelles extensions dans le cadre de la recherche.

Compatibilité : avec les standards (Fortran 77, Fortran 90), donc des extensions préférentiellement sous forme de commentaires spéciaux ajoutés au langage.

Rapidité : de la définition du langage, de la disponibilité des premiers compilateurs, et ce avec des critères de validation communs.

Ces objectifs ont été poursuivis au long de l'année 1992 et du début de l'année 1993, à force de travail de spécification et de votes successifs.

1.1.3 Fonctionnement

Le travail de spécification de HPF a été organisé de manière hiérarchique, avec des sous-groupes chargés de rédiger et discuter les points techniques des diverses propositions et des réunions plénières du comité où les décisions d'adoption étaient faites à la majorité, chaque organisation participant régulièrement aux réunions ayant droit à une voix.

Cinq sous-groupes thématiques ont travaillé à la définition des différents aspects du langage, au cours de réunions directes, mais aussi de discussions permanentes sur des listes de messagerie électronique. Chaque sous-groupe était mené par un membre du comité chargé de synthétiser les discussions.

distribute : placement des données (Guy L. Steele Jr.).

f90 : sélection du langage de base (Mary Zosel).

forall : boucles parallèles (Charles Koelbel).

intrinsics : fonctions intrinsèques (Robert Schreiber).

io : entrées–sorties (Bob Knighten).

extrinsics : fonctions extrinsèques (Marc Snir).

L'aboutissement de ce travail collectif est un *langage de comité*, avec ses imperfections (soulignées au chapitre 2) mais aussi ses qualités. Une première version de la spécification du langage a été diffusée en 1993 [95] ainsi qu'un journal des développements n'ayant pas donné lieu à insertion dans la langage [94]. Après une année de commentaires publics et d'amorce d'implantation, une seconde version [96] a été diffusée en 1994 qui corrige et précise la précédente. Le langage comprend aussi un sous-ensemble officiel, le HPF *Subset*, dont le but était de faciliter le développement des premiers compilateurs sur une base simple mais considérée comme utile.

1.2 Le langage HPF

Une excellente présentation du langage HPF est disponible [162] en anglais. En français, l'introduction de [56] ou les transparents de [63] forment une source de départ. Nous ne faisons ici que survoler les principaux aspects du langage.

1.2.1 Modèle data-parallèle

Le langage reprend le modèle de programmation à parallélisme de données [34] issu de APL [141, 27]. L'idée est de pouvoir préciser des opérations non pas sur un scalaire, mais directement sur une collection, généralement des tableaux. Ce modèle est déjà présent en Fortran 90, à travers les expressions de tableaux et la notation dite des sections de tableaux.

Fortran 90 est intégralement repris et étendu par HPF qui reprend également l'idée des directives qui ne sont a priori que des commentaires pour un compilateur non HPF. Cette technique permet d'envisager des codes portables, exécutables sur machines parallèles avec un compilateur spécial et sur une simple station de travail avec un compilateur standard. Le *High Performance Fortran* contient donc Fortran 90. Un tel choix n'est pas sans impact de par la taille du langage et la complexité résultante pour le compilateur. Cependant une telle décision a été ressentie également comme porteuse d'avenir dans la mesure où Fortran 90 est perçu comme *le* langage d'avenir du calcul scientifique.

À notre humble avis il importait surtout qu'un programme HPF soit aussi un programme FORTRAN 90. En effet, il est intéressantque des codes spéciquement développés pour obtenir de très hautes performances sur des architectures parallèles puissent fonctionner ou être essayés sur des architectures séquentielles. Cependant, certaines extensions de HPF ne sont pas sous forme de commentaires (`pure`, `forall`), il en résulte donc la propriété contraire, tout programme FORTRAN 90 arbitraire et pas spécialement parallèle doit pouvoir être interprété comme un programme HPF.

1.2.2 Placement des données

À FORTRAN 90 sont ajoutées des directives de placement des données sur les processeurs. Ce placement est à deux niveaux : l'alignement et la distribution. Il existe une version statique de ces directives qui déclare le placement souhaité, et une version dynamique qui modifie le placement courant des données, pour celles déclarées `dynamic`.

Distribution

Elle précise le découpage d'un tableau de données sur un tableau de processeurs abstraits (déclaré par la directive `processors`). Ce découpage est spécifié par la directive `distribute`. Il est défini dimension par dimension, et peut être `block`, `cyclic`, ou cyclique généralisé `cyclic(`n`)`. Un arrangement de processeurs peut dépendre à l'exécution du nombre de processeurs disponibles grâce à des fonctions intrinsèques de requête système (comme `NUMBER_OF_PROCESSORS()` ci-dessous). La versions dynamique de `distribute` est `redistribute`.

```
      real A1(100), A2(100,100), A3(100,100,100)
chpf$ processors P1(20), P2(4,4), P3(NUMBER_OF_PROCESSORS())
chpf$ distribute A1(block) onto P1
chpf$ distribute A2(cyclic,block) onto P2
chpf$ distribute A3(*,cyclic(5),*) onto P3
```

Une distribution `block` découpe les données en blocs réguliers et affecte un bloc par processeur sur la dimension concernée. La notation `block(n)` permet de préciser la taille des blocs souhaitée plutôt que d'avoir une valeur par défaut calculée. Une distribution `cyclic` attribue les éléments un à un aux processeurs, et reprend cycliquement au premier processeur jusqu'à épuisement des données à répartir. Une distribution `cyclic(n)` répartit les éléments par blocs et cycle sur les processeurs si nécessaire, comme on distribue les cartes trois par trois en reprenant au premier joueur jusqu'à épuisement des cartes au tarot.

Alignement

L'idée est de préciser d'abord les relations de placement entre données. Un seul des tableaux est ensuite distribué effectivement. Cette distribution entraîne implicitement celle des données alignées. La relation d'alignement est définie par la directive `align` entre tableaux, ou éventuellement vers un tableau artificiel (appelé `template`) ; La sémantique de l'alignement est très riche : les tableaux peuvent être alignés, translatés, inversés, transposés, écrasés, dupliqués, éclatés les uns par rapport aux autres, par le jeu de fonctions affines entre dimensions. La version dynamique de `align` est `realign`.

```
!hpf$ template T(100,100)
!hpf$ align A(i,j) with T(i,j)     ! alignement direct
!hpf$ align H(i,j) with T(i+2,j-1) ! translation
!hpf$ align L(i,j) with T(101-i,j) ! inversion sur i
!hpf$ align B(j,i) with T(i,j)     ! transposition
!hpf$ align C(i) with T(2,i)       ! localisation
!hpf$ align D(*) with T(50,50)     ! localisation et écrasement
!hpf$ align E(i) with T(101-i,*)   ! inversion et duplication
!hpf$ align F(i,j) with T(2*i,4*i) ! éclatement
!hpf$ align G(*,i) with T(i,*)     ! duplication et écrasement
```

Appels de procédures

Le langage autorise la description du placement des arguments d'une fonction. Celui-ci peut être *descriptif* (on avertit le compilateur que les données seront placées comme décrit), *prescriptif* (le compilateur doit forcer le placement) ou *transcriptif* (au compilateur de gérer ce qui vient, comme il peut).

Un point obscur de la définition du langage est le suivant : la déclaration à l'appelant des placements souhaités par les appelés n'est semble-t-il pas obligatoire, ce qui suggère (1) que le compilateur doit être conservatif et passer par défaut quantité d'information sous forme de descripteurs pour chaque donnée, et faire des vérifications à l'exécution sur la conformité éventuelle des arguments, et (2) que c'est l'appelé qui doit par défaut mettre en œuvre les replacements requis alors qu'il ne dispose d'aucune information sur le placement des arguments. Imposer à l'utilisateur de déclarer les prototypes de fonctions et placements des arguments semble une contrainte raisonnable, qui simplifierait le travail du compilateur et de l'exécutif. De telles contraintes améliorent aussi la qualité logicielle des codes. Elles sont obligatoires dans d'autres langages tel C++. Bien que ce ne soit pas l'habitude de la communauté Fortran d'annoncer au compilateur ce qui est fait, la déclaration obligatoire des interfaces de routines et du placement des arguments semble une politique souhaitable.

1.2.3 Parallélisme

Enfin les expressions de tableaux et réductions de FORTRAN 90, par exemple `A=B+C` ou bien `x=SUM(A)` ou les données `A B C` sont des matrices, ont été complétées, sous forme d'une construction parallèle (`forall`), d'une directive (`independent`) et de nouvelles fonctions intrinsèques (`MINLOC` ...).

Forall

Cette construction généralise les expressions de tableaux en explicitant l'index, de manière à permettre par exemple une transposition. Elle peut être masquée par un tableau de conditions booléennes. L'ordre implicite des calculs du `FORALL` est l'ordre data-parallèle, *i.e.* la partie à droite de l'affectation doit être calculée sans modification de la partie gauche.

```
    FORALL(i=1:n, j=1:m, A>0)
```

```
    A(i,j) = A(i-1,j)+A(i,j-1)+A(i+1,j)
  END FORALL
```

Independent

Cette directive spécifie que les itérations de la boucle suivante sont indépendantes les une des autres, et peuvent donc être exécutées dans un ordre arbitraire, choisi par le compilateur. Elle s'applique à une boucle `DO` ou `FORALL`; `new` précise les variables scalaires qui sont locales aux itérations, chaque processeur doit alors avoir sa propre version de ces variables.

```
!hpf$ independent, new(j)
      do i=1, n
!hpf$   independent
        do j=1, m
           A(i,j) = B(i,j)
        enddo
      enddo
```

Réductions

Les réductions de FORTRAN 90 (telles `SUM`, `PROD`, `MIN`) sont complétées (`MINLOC`, `MAXLOC`, `IAND` ...) et étendues aux calculs à préfixe parallèle (opérations qui accumulent et conservent les valeurs intermédiaires du calcul de réduction dans un vecteur par exemple).

Le *High Performance Fortran* incorpore donc des extensions qui permettent à l'utilisateur de préciser à la fois le parallélisme et le placement, au moins pour les applications *régulières*.

Pour aller plus loin au niveau des optimisations, le mécanisme dit des fonctions extrinsèques donne la possibilité d'écrire des parties d'une application en passage de messages, et à la main, dans le modèle SPMD.

1.3 Évolution de HPF

À partir de 1995 le Forum a repris la spécification du langage dans le but de l'enrichir [97]. Le cours des discussions a cependant été autre. Il est probable que les nombreuses extensions projetées resteront à l'état de spécifications approuvées faute d'implantation. En effet, le développement des compilateurs pour la première version du langage s'est révélé plus ardu que prévu, et le langage est maintenant plutôt sur la voie de la simplification. Au risque même d'en retirer inutilement la richesse, de manière à faciliter le développement des compilateurs.

Diverses extensions ont cependant été adoptées ou sont en cours d'adoption [224]. Elles concernent notamment l'inclusion de parallélisme de tâches [98] au niveau du langage, en sus du parallélisme de données. Ces adjonctions sont guidées par des applications ou classes d'applications qui en bénéficieraient grandement dans l'optique de leur portage en HPF.

Cependant il n'est pas certain que ces extensions soient soutenues par des implantations montrant à la fois leur utilité et leur implantabilité. De plus, certaines extensions sont ajoutées parce qu'utiles et compilables dans un certain contexte, mais leur implantation au sein d'un compilateur se doit de les mettre en œuvre dans tous les contextes, y compris les moins favorables, ce qui augmente les coûts de développement.

1.3.1 Placement étendu

La richesse et la généralité des placements réguliers autorisé par HPF, au travers des directives `align` et `distribute`, est l'une des difficultés de sa compilation. Cependant ces placement restent réguliers par leur essence, et ne conviennent pas à certaines applications. Le Forum a donc étudié de nouvelles extensions pour le placement. Quelques exemples :

blocs irréguliers : l'utilisateur précise la taille pour chacun des blocs de la distribution.

```
      integer blocks(8)
!hpf$ processors P(8)
!hpf$ distribute A(block(blocks))
```

fonction : le numéro de processeur est précisé indirectement par un tableau.

```
      integer map(n)
!hpf$ processors P(10)
!hpf$ distribute a(indirect(map)) onto P
```

section : une distribution concerne une section de processeurs.

```
!hpf$ processors P(4,4)
!hpf$ distribute (block,block) onto P(1:2,2:4) :: ...
```

1.3.2 Parallélisme étendu

Au delà du placement, de nouvelles extensions sont également proposées pour décrire plus de parallélisme, mais aussi guider plus finement le compilateur. Quelques exemples :

réductions : la directive `reduction` permet de préciser qu'une boucle est parallèle sous réserve qu'une opération sur un scalaire soit une réduction. Chaque processeur peut faire les opérations en local sur sa propre copie de l'accumulateur, sous réserve de l'initialiser à une valeur bien choisie fonction l'opérateur concerné (par exemple 0 pour une somme, 1 pour un produit, $+\infty$ pour un minimum) avant la boucle et de recombiner les valeurs calculées localement sur chaque processeurs entre elles et avec la valeur initiale au sortir. Une recombinaison nécessite peu de communications, avec des volumes faibles, et certaines machines proposent de telles opérations implantées au niveau matériel.

```
!hpf$ independent,
!hpf$.  new(x), reduction(s)
      do i=1, n
        x = SQRT(A(i))
        s = s + x
        A(i) = A(i) + x
      enddo
```

placement des calculs : l'utilisateur précise explicitement sur quels processeurs doivent être effectuées les opérations.

```
!hpf$ independent
      do i=1, n
!hpf$   on home(B(i))
        A(i+1) = B(i)*C(i)*D(i)
      enddo
```

tâches : précision du parallélisme de tâches, par exemple sous forme de sections de codes, ou bien sous forme de programmes MPI qui sont eux-mêmes des programmes HPF.

De ces nombreuses extensions suggérées et définies, peu feront partie du langage final. C'est sans doute heureux pour le développement des compilateurs, mais moins pour certaines applications qui éventuellement auraient besoin de ces extensions pour que leur portage en HPF autorise de bons espoirs d'accélération.

1.4 HPF et l'état de l'art

Le chapitre 2, rédigé en français, est issu de [58, 60]. Il présente rapidement le langage HPF et les difficultés de compilation posées au moment de sa conception, qui généralisait des langages de recherche pour lesquelles les techniques de compilation n'étaient que partiellement développées et implantées. Ces problèmes sont liés à la richesse du langage, essentiellement en ce qui concerne le placement des données à travers alignements et distributions généralisées. De nombreux problèmes attendent (allocation, addressage, communication) une solution. La structure mathématique de ces questions pousse à l'optimisme.

1.5 État de l'art de la compilation de HPF

Le chapitre 3 présente en anglais les différentes techniques de compilation proposées pour HPF. Il a été l'objet d'un travail commun avec Cécile Germain et Jean-Louis Pazat dans le cadre de la préparation d'un cours sur la compilation du langage HPF pour l'école de printemps du GDR-PRC Parallélisme, Réseaux et Systèmes (PRS) aux Ménuires en mars 1996 [69]. Nous en résumons ici le contenu en français.

1.5.1 Problèmes de compilation

La section 3.2 décrit les problèmes rencontrés pour la compilation du langage. Ceux-ci proviennent des applications, du langage et de l'architecture visée par le compilateur.

Application : une certaine régularité et localité est nécessaire pour que le parallélisme de l'application puisse être exprimé par le langage et exploité par le compilateur.

Langage : le *High Performance Fortran* n'est pas un langage simple ; sa taille est importante à cause de l'inclusion intégrale de FORTRAN 90 ; l'héritage de FORTRAN 77 doit être géré dans le contexte étendu des extensions de langage de HPF ; enfin la richesse des directives proposées permet de décrire des placements étranges, mais aussi de dissimuler au compilateur le placement des données, prévenant ainsi toute optimisation statique.

Cible : le choix du modèle a aussi un impact significatif sur le compilateur ; on distingue l'architecture de la machine, SIMD ou MIMD à mémoire répartie, le modèle de communication, à passage de messages ou à base d'écritures et lectures directes dans la mémoire distante d'un autre processeur ; en présence de matériel spécialisé pour certains types de communications le compilateur se doit d'en tirer profit.

1.5.2 Survol de la compilation

La section 3.3 survole les différents problèmes qui doivent être abordés et résolus par le compilateur. Celui-ci doit gérer lors de l'exécution le caractère réparti de la mémoire, et donc résoudre les références aux données en adressage local ou bien en communication selon que la donnée nécessaire à un calcul est locale ou distante. Ces différents problèmes sont : le modèle cible de programmation, l'assignation des calculs aux différents processeurs, la technique de la résolution dynamique des accès, la génération de code et les diverses optimisations proposées.

Modèle SPMD : le compilateur ne peut pas générer un code particulier pour chaque nœud en restant extensible, il vise donc à produire un code paramétrique dont l'exécution dépend de l'identité du processeur ; c'est cependant bien le même exécutable qui s'exécute sur chaque nœud.

Règle des calculs locaux : cette règle permet d'affecter aux processeurs les calculs, en désignant celui qui est propriétaire de la variable affectée par une assignation ; ce choix n'est pas forcément optimal, mais il simplifie la compilation et la compréhension des utilisateurs.

Résolution dynamique des accès : la résolution locale ou distante des accès peut être opérée au niveau élémentaire en contrôlant chaque référence au moyen d'une garde apropriée ; ce procédé hautement inefficace a l'avantage de pouvoir être appliqué dans tous les contextes de manière directe, et est à la base de toute compilation comme technique par défaut.

Ensemble d'itérations: les décisions du compilateur en matière d'affectation des calculs aux processeurs se formalisent simplement de manière ensembliste, en décrivant l'ensemble des itérations que doit calculer chaque processeur ; il en est de même pour les données placées sur les processeurs ; il est alors aisé d'en déduire les données nécessaires aux opérations à effectuer et donc les ensembles de données à communiquer.

Génération de code : à partir de ces ensembles, des codes de haut niveau peuvent être décrits, où il suffit d'énumérer les ensembles ; il apparaît cependant que ce ne sont que des vœux pieux, et que c'est la manipulation pratique de ces ensembles, et la capacité à générer des codes d'énumération rapides qui est la clef de la compilation de HPF ; le *comment* plutôt que le *quoi* est le vrai problème.

Optimisations : diverses optimisations au niveau des communications ont été proposées également pour en réduire le nombre et le contenu ; c'est la vectorisation, l'aggrégation et la coalescence des messages ; on cherche également à recouvrir les temps de communication par des calculs utiles, avec des techniques de pipeline.

La section se termine en montrant que les données manipulées par un nid de boucles parallèles avec des bornes et des accès affines peut se formaliser dans le cadre mathématique des polyèdres en nombres entiers, à savoir sous forme de systèmes d'équations et d'inéquations affines paramétriques. Ce cadre mathématique général et puissant est à la base, directement ou indirectement, implicitement ou explicitement, des techniques de compilation proposées.

1.5.3 Références régulières

La section 3.4 présente les techniques de compilation pour les accès réguliers, *i.e.* ceux dont le motif d'accès peut être connu et manipulé à la compilation. Trois types de techniques sont présentés : les formes explicites, les automates et les méthodes polyédriques. Il est important de noter que de très nombreux travaux s'attaquent à cette question, et qu'il est en conséquence important d'en distinguer les hypothèses pour parvenir à les classer et les comparer.

Formes explicites : cette technique vise à produire une formule directement applicable qui décrit l'ensemble souhaité ; la résolution manuelle et paramétrique des cas les plus généraux est une entreprise titanesque (il existe cependant des algorithmes pour le faire automatiquement), ces techniques se penchent donc sur des cas simplifiés, en s'appuyant sur les sections de tableaux (triplets premier élément, dernier élément et pas) ; cette représentation est fermée pour l'intersection et la transformation affine, ce qui suffit à décrire les ensembles de communication quand on accède à des sections de tableaux de données distribués `block` ou `cyclic` ; pour manipuler des distributions cycliques générales, le code gère à l'exécution des ensembles de sections de tableau ; cette technique transfère à l'exécution la détermination des ensembles utiles, et est limitée à des cas simples.

Automates : cette deuxième technique vise à l'efficacité, en cherchant à déterminer le motif de répétition des accès pour une boucle et un tableau donnés, à le stocker dans une table de transitions et à réutiliser ce motif pour parcourir très rapidement les adresses locales nécessaires ; la classe d'ensembles pour lesquels on peut déterminer un automate est pour l'instant limitée, et cette technique charge l'exécutif de générer et gérer les tables de transition.

Polyèdres : cette technique est la plus générale pour la complexité des ensembles qui peuvent être manipulés, *i.e.* des nids de boucles affines avec des accès affines aux tableaux de placement HPF arbitraires ; elle se base sur des algorithmes généraux pour produire des codes d'énumération correspondant aux différents ensembles ; mathématiquement, il s'agit de manipuler des ensembles de points entiers convexes, ou unions de tels ensembles, ce qui correspond à des sous-classes des formules de Presburger ; pour les cas simples les résultats sont tout à fait similaires à ce qui est obtenu à la main avec la méthode des formes explicites ; pour les cas plus complexes, par exemple des domaines d'itération triangulaires, la résolution manuelle n'est pas envisageable ; l'efficacité des codes générés est liée à la simplicité des ensembles à parcourir.

1.5.4 Problèmes irréguliers

Lorsque les ensembles ne peuvent pas être déterminés à la compilation, par exemple à cause d'accès indirects à des tableaux, cette opération est remise à l'exécution. Ces techniques sont l'objet de la section 3.5. Elles consistent à générer un code qui détermine d'abord les ensembles en question (partie *inspecteur*), puis réutilise ces ensembles pour assurer les communications et les calculs (partie *exécuteur*). Cette technique peut être efficace si les mêmes ensembles sont utilisés plusieurs fois.

1.5.5 Allocation et adressage mémoire

Enfin la section 3.6 discute le problème de l'allocation et de l'adressage de la mémoire. Elle doit être gérée pour stocker les données locales mais aussi les copies temporaires des données distantes. Le compilateur doit faire certains choix, entre économie de mémoire et rapidité des fonctions d'accès.

Temporaires : l'allocation des temporaires n'est pas un problème facile, lorsque des données peuvent être à la fois locales ou distantes ; en effet, si un stockage distinct est employé, le code doit effectuer le choix de l'adresse à référencer effectivement au cœur des boucles, générant alors des gardes trop coûteuses ; si on veut découper à plus haut niveau les calculs en fonction de la localisation, locale ou temporaire, de chaque référence, on se trouve devant une explosion exponentielle de la taille du code, sans compter la faisabilité douteuse de l'opération (comment décrire et manipuler ces ensembles ?) ; la seule solution générale est de gérer les données temporaires et locales de manière homogène, soit en recopiant les données locales dans l'espace des temporaires, soit en proposant une méthode de gestion des données distantes qui soit homogène à celle des données locales.

Adressage : il doit être rapide car il n'est pas raisonnable d'espérer de bonnes performances si le code doit passer son temps à faire des calculs d'adresses, par exemple des conversions d'adresses globales en adresses locales ; il peut être suffisant de pouvoir gérer efficacement de manière incrémentale des adresses locales au cœur d'un nid de boucle ; une bonne technique d'adressage peut imposer un certain niveau d'allocation mémoire inutile, qui doit être limité autant que possible.

Un certain nombre de techniques applicables dans des contextes différents ont été proposées. Elles sont décrites et comparées en fonction des critères présentés ci-dessus.

Recouvrements : dans de nombreuses applications les nœuds n'accèdent qu'à leur données locales ou à des données immédiatement voisines ; pour ce cas la technique de l'allocation des recouvrements, qui consiste à étendre l'allocation locale de manière à laisser de l'espace pour les données voisines, règle alors du même coup le problème de l'adressage et de l'allocation des temporaires, pour un cas simple mais néanmoins très fréquent.

Compression : dans le cas général du placement HPF impliquant à la fois un alignement et une distribution cyclique généralisée, on compresse simplement l'espace concerné, soit par ligne soit par colonne, de manière à retirer les trous réguliers ; une telle technique est compatible avec celle des recouvrements.

Pagination : il s'agit de découper l'espace du tableau en pages, chaque processeur n'allouant ensuite que les pages ou parties de pages pour lesquelles il possède localement des données ; cette technique permet d'intégrer naturellement et homogènement les données temporaires ; elle nécessite cependant plus de travail de gestion à l'exécution, et souffre d'un surcoût d'allocation très important pour les distributions cycliques.

Autres : il a aussi été suggéré de compresser complètement l'espace des données, en posant alors des problèmes d'adressage difficiles ; les tables de hachage ont également été envisagées.

Globalement, ces techniques montrent que le compilateur est tributaire de l'information dont il dispose statiquement pour générer un code aussi efficace que possible, quand l'application s'y prête.

Chapitre 2

HPF et l'état de l'art

Ce chapitre est un compte rendu écrit de la présentation orale faite aux Journées du SEH 1994 [58]. Il constitue le rapport EMP CRI A-260 [60].

Abstract

HPF (*High Performance Fortran* is a new standard dataparallel language for Distributed Memory Multiprocessors. The user advises the compiler about data mapping and parallel computations thru directives and constructs. This paper presents HPF and discusses HPF adequacy to the current state of the art in compilation for DMM.

Résumé

HPF (*High Performance Fortran*) est un nouveau langage de programmation standard pour machines à mémoire répartie. Il propose un modèle de programmation à parallélisme de données. Des directives et des constructions, qui précisent le placement des données et le parallélisme présent dans l'application, aident le compilateur. Cet article présente HPF et discute l'adéquation du langage à l'état de l'art en matière de compilation pour machines à mémoire répartie.

2.1 Introduction

« Toujours plus vite » : l'histoire de l'informatique scientifique, et plus particulièrement celle des supercalculateurs, semble suivre cette devise depuis les origines. Pour cela, les innovations technologiques et architecturales se sont succédées. L'une de ces innovations consiste à donner à ces machines une architecture parallèle massive, en composant un supercalculateur à partir de processeurs du commerce, souvent RISC, et en les reliant par un réseau de communication performant. La mémoire de ces machines est partagée si chaque processeur élémentaire (nœud) accède à la totalité de la mémoire. Elle est répartie, si un nœud n'accède directement qu'à sa mémoire locale, les accès à distance devant faire appel à d'autres techniques comme l'échange de messages.

Ce type d'architecture à mémoire répartie est particulièrement attrayant pour les constructeurs, car le principe-même en est extensible : le nombre de processeurs sur le réseau de communication peut être étendu pour augmenter la puissance de la machine, sans que le lien mémoire processeur ne devienne *a priori* un goulot d'étranglement. En contrepartie de cette extensibilité, ces machines sont difficiles à programmer. Comme les données ne sont pas partagées par les processeurs, le programmeur doit gérer l'adressage local et les communications entre nœuds, tâche difficile qui alourdit le développement. De plus, de par l'absence d'un modèle de programmation portable, la pérennité de ces investissements importants n'est pas assurée.

Deux voies sont explorées actuellement, qui visent à proposer sur ces machines des modèles de programmation similaires à ceux disponibles sur les machines traditionnelles. La première transmet le problème du maintien de la cohérence des données et de la génération des messages au système d'exploitation et au matériel [180], avec un modèle à parallélisme de tâche. La seconde s'appuie uniquement sur le compilateur [45, 105], en s'inspirant du parallélisme de données des machines SIMD. Dans les deux cas, les langages de programmation ont été étendus de manière à assister le processus de traduction, que ce soit FORTRAN S [29] qui aide le système d'exploitation à gérer la mémoire virtuelle partagée, ou VIENNA FORTRAN [50, 260] et FORTRAN D [131, 241] qui assistent le compilateur dans la génération des échanges de messages.

Les prototypes de recherche ont montré que cette dernière approche était réaliste. Des outils de ce type sont proposés par certains constructeurs, tel que TMC avec CM-FORTRAN. Cependant l'absence de portabilité de ces langages constituaient encore un frein à leur adoption. Pour répondre à ce besoin, un processus de standardisation s'est mis en place à l'initiative de DEC fin 1991, pour aboutir à une spécification de langage, le *High Performance Fortran* (HPF) [95], en 1993.

Dans cet article, nous discutons l'adéquation de HPF à l'état de l'art en matière de compilation pour machines à mémoire répartie, en décrivant les problèmes posés par la compilation de HPF. Les principales caractéristiques de HPF seront d'abord présentées en section 2.2. Dans la section 2.3, les différents problèmes posés par sa compilation seront exposés. Enfin quelques éléments de solution seront suggérés.

2.2 Présentation de HPF

À l'initiative de DEC, le monde du calcul parallèle (constructeurs, universitaires, utilisateurs) s'est réuni en Forum aux États-Unis, en 1992-1993, pour spécifier un nouveau langage de programmation pour les machines à mémoire répartie. L'objectif était de proposer un langage standard *de fait*, adopté par la plupart des constructeurs. Cette démarche est de nature commerciale, puisque l'existence d'un standard permet de rassurer l'acheteur de supercalculateurs parallèles quant à la durée de vie de ses investissements, et faire disparaître du même coup l'un des freins à l'expansion du marché de ces machines.

L'idée initiale, s'inspirant de FORTRAN D et VIENNA FORTRAN, était d'ajouter des directives de compilation à un langage séquentiel de manière à guider le compilateur sur le placement des données dans la mémoire répartie et sur l'exploitation du parallélisme. Le langage spécifié dépasse pour partie les objectifs assignés, et en laisse d'autres de côté, comme la gestion des entrées-sorties. HPF est basé sur le standard FORTRAN 90. Le choix d'un Fortran dans le domaine du calcul scientifique, langage auquel les utilisateurs sont habitués, s'imposait. HPF y ajoute des directives, de nouvelles structures et de nouvelles fonctions intrin-

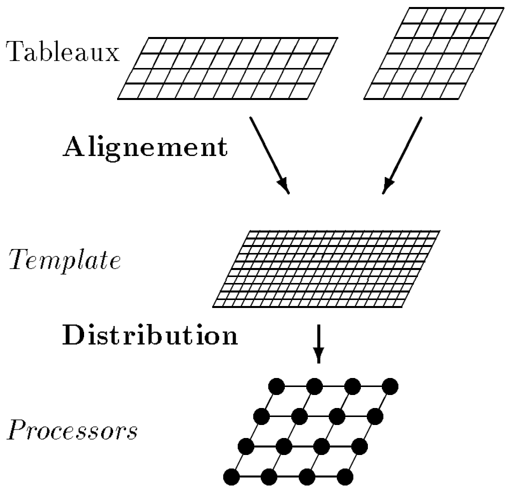


FIG. 2.1 – *Placement des données en HPF*

sèques. Le reste de la section présente les directives de compilation de HPF pour le placement des données, puis les directives et structures parallèles.

2.2.1 Placement des données

En HPF le programmeur décrit le placement des données de son application dans la mémoire répartie de la machine au moyen de directives. Le compilateur exploite ces directives pour allouer la mémoire sur chaque processeur, gérer l'adressage et générer les communications. Ces directives permettent d'équilibrer (statiquement) la charge entre les processeurs (*i.e.* d'exploiter de parallélisme), tout en réduisant les communications qui constituent le goulot d'étranglement pour ces machines.

Le modèle de placement proposé par HPF comporte deux niveaux (figure 2.1). Les tableaux de données du programme sont placés sur une vue cartésienne des processeurs. Un objet intermédiaire, le *template* (gabarit), sert à mettre en correspondance les données des différents tableaux relativement les uns aux autres, avec la directive d'alignement `align`. Il est ensuite découpé régulièrement avec la directive `distribute` pour répartir les données alignées sur chaque morceau de *template* entre les processeurs. Les alignements et distributions sont déclarés statiquement, et peuvent être modifiés dynamiquement avec les directives `realign` et `redistribute`. À l'entrée des procédures le programmeur peut également décrire le placement des paramètres attendus, le compilateur devant assurer les mouvements de données nécessaires.

L'alignement est lié à l'algorithme. Il caractérise la localité des données : deux éléments de différents tableaux placés sur un même élément de *template* seront finalement sur un même processeur. Techniquement, la directive d'alignement spécifie la mise en correspondance des éléments de tableau sur les éléments du *template* dimension par dimension, au moyen de fonctions entières affines d'une variable (figure 2.2). La directive d'alignement permet, par le jeu de ces fonctions affines, de transposer, retourner, écraser, éclater, dupliquer, décaler régulièrement les éléments des tableaux les uns par rapport aux autres.

```
      real, dimension(100,100):: a,b
chpf$ template t(100,200)
chpf$ align a(i,j) with t(101-j,2*i)
chpf$ align b(*,i) with t(*,i)
chpf$ processors p(4,4)
chpf$ distribute t(block,cyclic(10)) onto p
chpf$ independent(i)
      do 10 i=1,100
 10     a(i,3)=b(10,2*i)
c transposition partielle
      forall(i=1:80,j=1:80) b(i,j)=b(j,i)
```

FIG. 2.2 – *Exemple de directives et boucles HPF*

La distribution est elle liée à l'architecture globale du parallélisme dans la machine, pouvant impliquer par exemple plusieurs niveaux comme des *threads*, des processus, des processeurs avec mémoire partagée ou enfin reliés par un réseau de communication externe. Elle précise la granularité du parallélisme souhaitée. Chaque dimension du *template* peut être distribuée `block(n)` ou `cyclic(n)` sur une dimension des *processors*.

Ce modèle de placement, bien que donnant une grand latitude à l'utilisateur, reste « régulier ». Il ne convient pas à tous les types d'applications, mais se prête bien aux problèmes data-parallèles impliquant des calculs simples sur des géométries cartésiennes, qui sont la cible essentielle du langage HPF.

2.2.2 Parallélisme et autres extensions

Le parallélisme est décrit explicitement en HPF au moyen de boucles parallèles et de fonctions de réductions. HPF ajoute deux « sémantiques » de boucle (figure 2.3) : la boucle indépendante (directive `independent`), où les itérations sont indépendantes les unes des autres, et la boucle à parallélisme de données, avec une notation implicite (sections de tableau : `a(2:n)=a(1:n-1)`) et une forme explicite avec l'instruction `forall`.

Un grand nombre de fonctions intrinsèques de calcul de fonctions réductions et à préfixe parallèle sont également mises à disposition du programmeur, pour calculer les minimum, maximum et leur localisation, les somme, produit, « et » et « ou » logiques des éléments d'un tableau, etc. Certaines de ces fonctions proviennent de FORTRAN 90.

HPF ajoute également des fonctions intrinsèques nouvelles, des fonctions d'enquête système permettant de connaître à l'exécution les caractéristiques disponibles de la machine (comme le nombre de processeurs) et la possibilité d'importer des fonctions externes à HPF en déclarant une interface. Se rendant compte de la taille et de la difficulté du langage pour les compilateurs, le Forum a spécifié un sous-ensemble « minimum » de HPF. Il comprend principalement FORTRAN 77, les directives de placement statique et une forme réduite de `forall`.

```
chpf$ independent
      do 10 i = 1, 3
        lhsa(i) = rhsa(i)
 10     lhsb(i) = rhsb(i)
```

```
forall(i = 1:3)
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
endforall
```

FIG. 2.3 – *Boucles parallèles de HPF*

2.3 Compilation de HPF

Les compilateurs devront être performants, car les utilisateurs attendent de ces machines de hautes performances. Les principaux problèmes rencontrés portent sur la taille du langage, la sémantique étendue du placement, l'énumération des itérations d'une boucle locale à un processeur, la génération des messages, les appels de procédures avec des paramètres répartis, les entrées-sorties et les fonctions intrinsèques.

FORTRAN 90 [140] est le dernier standard FORTRAN. Il ajoute à FORTRAN 77 les caractéristiques des langages impératifs modernes telles les structures, les pointeurs, l'allocation dynamique, les modules et une syntaxe étendue. Le résultat d'une décennie de standardisation est un très gros langage dont les premiers compilateurs commerciaux — pour machines séquentielles — commencent seulement à apparaître. Le langage est si complexe que le DoD (ministère de la défense américain) l'a rejeté pour l'exécution de ses contrats.

HPF propose une sémantique de placement « régulier » très générale. Ce placement n'est parfois connu qu'à l'exécution (réalignements et redistributions, appels de procédures, paramétrage du placement par des fonctions d'enquête système), les informations sont alors très réduites pour compiler efficacement. De plus, les placements impliquant des alignements non unitaires ($ai+b, a \neq 1$) et des distribution cycliques généralisées (`cyclic(`n`)`, $n \neq 1$) sont pour l'instant au-delà l'état de l'art de la recherche, et donc a fortiori du savoir faire des implanteur du langage. Enfin la gestion de la duplication des données complique le maintien de la cohérence et les algorithmes de génération de messages. À partir de la description du placement, le compilateur doit allouer la mémoire de manière réduite sur chaque processeur tout en garantissant un adressage efficace, malgré certaines irrégularités permises par le placement.

La compilation performante des boucles parallèles implique d'abord la génération d'un code efficace (de surcoût de contrôle minimal) permettant d'énumérer les itérations locales à chaque nœud. Cette énumération doit être liée à la méthode d'allocation choisie pour les données, pour exploiter la mémoire cache de la machine. Les boucles parallèles impliquent le plus souvent des communications, certaines données n'étant pas disponibles localement. Il faut déterminer à la compilation ces ensembles, quelle que soit la répartition des différentes données entre les processeurs. Le compilateur doit également allouer les tableaux temporaires pour stocker, le temps des calculs, ces données distantes et assurer l'adressage de ces temporaires plutôt que des données locales quand nécessaire. Enfin l'utilisateur attend que les caractéristiques particulières de la machine soient exploitées.

Le passage de paramètres répartis au niveau des appels de procédure pose aussi un problème : FORTRAN 90 permet de passer des sous-tableaux de tableaux comme arguments, ce qui complique la tâche du compilateur lorsque le tableau initial est déjà réparti. Plusieurs compilations sont nécessaires en fonction des différents contextes d'appels. De plus, même si les réalignements et redistributions explicites ne sont pas dans le sous-ensemble de HPF, les redistributions à l'entrée et à la sortie des procédures nécessitent la génération de code comparable.

Lors de la spécification de HPF, le Forum n'a pas trouvé de point d'accord pour le traitement des entrées-sorties. Il n'y a pas de consensus sur un paradigme unique d'entrées-sorties parallèles. Il faut exploiter la structure particulière du système d'E/S parallèles de la machine cible pour stocker par morceaux les données, et pouvoir récupérer une description du stockage parallèle des données pour les recharger en mémoire ultérieurement. Ce problème délicat nécessite une aide du programmeur pour le compilateur, et est à l'ordre du jour du second tour de discussion du Forum HPF qui a repris en janvier 1994.

Enfin HPF possède de nombreuses fonctions intrinsèques, venant à la fois de FORTRAN 90 et de la nouvelle spécification. De plus ses fonctions sont souvent très paramétrables. La première difficulté est leur nombre : il faut écrire un code correct et efficace pour chacune de ces fonctions, même pour un ordinateur séquentiel. La seconde difficulté est d'un autre ordre de grandeur : elles doivent être implantées de manière à fonctionner en parallèle, quelle que soit la répartition sur la machine des tableaux passés en argument. Il est raisonnable de supposer qu'une implantation spécifique soit souhaitable pour chaque distribution différente des données envisagée, ce qui multiplie directement le travail de codage de ces fonctions pour le langage HPF.

2.4 Conclusion

De part ces nombreux problèmes, le développement d'un compilateur complet et optimisant pour HPF est une entreprise qui semble au-delà de l'état de l'art. Les prototypes de compilateur pour des langages comparables qui existent actuellement ne gèrent que des alignements qui sont de simples décalages, et des distributions par bloc ou cycliques simples ($n = 1$). L'allocation des temporaires et la génération des communications sont performantes quand les motifs d'échange de messages sont réguliers, dans le cas de recouvrements au voisinage : les données distantes sont stockées en élargissant les déclarations locales des tableaux, ce qui permet en plus d'adresser ces données comme si elles étaient locales. Des librairies optimisées implantent les algorithmes ou communications courants (transposition, inversion de matrice, etc.).

Pour le reste, on doit faire appel à des techniques de résolution dynamique des accès aux variables réparties : chaque accès à un élément de tableau éventuellement distant doit être gardé de manière à vérifier sa localité et générer les communications si nécessaires. Cependant de telles techniques sont incompatibles avec une exécution efficace des codes. Les compilateurs utilisent les routines « `PARTI` » qui permettent de déterminer dynamiquement le motif de communications au cours d'une phase d'examen à l'exécution *(inspector)* et de réutiliser ce motif plusieurs fois par la suite pour effectuer réellement les communications *(executor)*.

HPF hors de l'état de l'art ? Des techniques ont été proposées qui permettent de gérer des cas de placements complexes en HPF, en s'appuyant sur les automates [53], les sections de tableaux [117, 230] ou la théorie des polyèdres [5], qui poussent à l'optimisme. Cependant il est difficile de déterminer si la lourdeur du développement des compilateurs l'emportera sur la petite taille du marché attendant ces derniers. Il est probable que très peu de compilateurs commerciaux seront disponibles même si portés sur de nombreuses machines. La réussite rapide de ces projets, et l'efficacité que montreront les compilateurs dans la pratique, conditionnent l'avenir de ce langage.

Chapitre 3

State of the Art in Compiling HPF

Fabien Coelho, Cécile Germain et Jean-Louis Pazat

Ce chapitre a fait l'objet d'un cours à l'école de printemps PRS des Ménuires sur le parallélisme de données en mars 1996. L'objectif était de présenter l'état de l'art de la compilation de HPF et les problèmes posés [69]. Il doit paraître dans la collection LNCS chez Springer-Verlag, édité par Guy-René Perrin *et Alain* Darte.

Résumé

Proposer à l'utilisateur un modèle de programmation basé sur le parallélisme de données est une chose. Faire ensuite fonctionner les applications de manière très efficace est le problème posé ici pour un langage qui vise aux hautes performances sur les machines massivement parallèles. Ce chapitre discute les problèmes rencontrés pour compiler HPF et présente les techniques d'optimisation proposées jusqu'à maintenant, en ciblant le modèle de programmation SPMD à passage de messages des machines parallèles MIMD à mémoire répartie.

Abstract

Providing the user with a nice programming model based on the data-parallel paradigm is one thing. Running the resulting applications very fast is the next issue for a language aiming at *high performance* on massively parallel machines. This paper discusses the issues involved in HPF compilation and presents optimization techniques, targeting the message-passing SPMD programming model of distributed memory MIMD architectures.

3.1 Introduction

Parallelism is becoming a ubiquitous technique: micro-parallelism inside processors, moderate parallelism inside workstations, and soon parallel personal computers. Paradoxically, the decline and fall of massive parallelism is often announced.

Massively parallel architectures will be killed by shared memory machines, and by fast networks of workstations, because they are too costly and not reactive enough to technology evolution. On the other hand, parallel machines often remain the top performance providers, and at a better cost (by performance unit) than workstations.

The HPF effort may be seen as an attempt to make massively parallel machines usable for their natural applications, which are in the field of scientific computing and number crunching, thus making them attractive for end-users. However, the application field puts heavy pressure on the compiler effort: the users require efficiency and usability, at the same level as what is found on a workstation. But compilers and run-time support for workstations have a long (in time) and large (in involved people) history. HPF was designed in 1993, at a time when the compiler effort was not finished for its predecessor Fortran 90.

At the present time, HPF's usability *and* efficiency goals cannot be fulfilled both for all applications. Regular applications, such as dense linear algebra, enjoy the full benefit of HPF without excessive constraint on the programming style. However, HPF is so rich that, even in this case, the programmer must be aware of the general principles of parallel compiler technology in order to avoid some pitfalls. Moreover, many theoretical results are known, but few have been implemented in commercial compilers. Irregular or dynamic applications are in a poorer state. These applications heavily rely on run-time support. In general, such support has been developed in the message-passing framework. Plugging these tools into HPF compilers is an active research area. This paper will show that an application must present sufficient static knowledge for compiler optimization. Applications that are fully dynamic and irregular, as N-body or tree-code methods, cannot be efficiently ported to HPF just as is. In fact, it is not obvious that they can be efficiently ported to HPF at all, if the target architecture is a classical massively parallel message-passing machine, because such applications are highly adaptive, thus change at each step the distribution of the data and the communication patterns. Between these two extremes, a large number of applications need varying recoding efforts, but may be expected to run efficiently on future HPF compilers.

Parallel compiler technology is rapidly evolving. Hence, this paper focuses on some basic problems related to the really new features of HPF: general parallel assignment and data placement functions. The only target addressed here is the message-passing programming model. A number of issues for compiling HPF are first outlined (Section 3.2). Then an overview of HPF optimized compilation techniques and their formalization in an affine framework is presented. The existence of such a framework is the key to understanding the underlying regularity displayed by HPF mappings. The enumeration techniques for regular (Section 3.4) and irregular (Section 3.5) accesses are described. Finally the allocation techniques (Section 3.6) suggested so far are presented, illustrated and discussed.

3.2 HPF Compilation Issues

Let us introduce the three main issues depicted in Figure 3.1. On top is the application the user has in mind. This application is embedded in the data parallel paradigm and implemented using the HPF language. The user suggests through directives the mapping of data onto an abstract distributed memory parallel machine, and also gives hints about the parallelism available in the application. Detecting the parallelism and choosing a good mapping automatically is an active research area and is beyond the scope of this paper. From the HPF code and directives, the compiler is expected to generate some efficient code for the target architecture. All layers of the process contribute to problems for the compiler.

3.2.1 Application

The application is hopefully parallel to a large extent, and presents some kind of locality and repeated reference patterns, so that the communication induced by the distribution of data is small compared to the computations. For some applications the data parallelism may be large, but the problems are irregular and dynamic and the translation into the simple data structures available in Fortran, and the regular HPF mappings cannot express the actual locality. Such irregular problems must rely on advanced runtime librairies for analyzing and reusing communication patterns. More precisely, an application is *regular* if the references to distributed arrays are affine functions of the loop bounds. An application is *irregular* if the references to distributed arrays are data-dependent. Finally, an application is dynamic if its data structures evolve during the computation: in general this happens for irregular applications where the data used in the references to distributed arrays change.

3.2.2 Language

Fortran 90 [140] on which HPF is based, includes many features and intrinsics. Several years were necessary for the first compilers to be available. Also *old* features inherited from Fortran 77 [14], such as commons, storage and sequence association, and assumed size arguments, are closely linked to the idea of a sequential machine and contiguous memory and cannot be implemented simply on a distributed memory machine because the compiler does not know about the real structure intended by the user. On this large base were added HPF extensions [96]. Correctness must be addressed, that is every Fortran feature must coexist with every HPF feature. Tricky details must be systematically handled that are not related to performance but require energy and manpower.

The second problem is the variety of HPF mappings allowed, through alignments and distributions. Alignment strides and general cyclic distributions may lead to unexpectedly strange and apparently irregular mappings, as shown in Figure 3.2 where shaded elements show the array elements mapped onto the corresponding processor. One processor gets all the elements despite the cyclic distribution in one example; in the other, the even processors get twice as many array elements as the odd ones. Moreover, these possibly complex mappings, sometimes involving replication, may not be known at compile time, because of ambiguity allowed by remappings (Figure 3.3) or because they may depend on runtime values (for instance, the number of available processors), and also because of the `inherit` directive (Figure 3.4) at a procedure interface.

Application	Regularity Locality Parallelism
Language	HPF extensions Fortran 90 Fortran 77
Target	Programming model Runtime support Architecture

Figure 3.1: 3 layers view

```
      real A(n)
chpf$ processors P(4)
chpf$ template T(2*n)
chpf$ align A(i) with T(2*i)
chpf$ distribute T(cyclic(3)) onto P
```

```
      real A(n)
chpf$ processors P(4)
chpf$ template T(4*n)
chpf$ align A(i) with T(4*i)
chpf$ distribute T(cyclic) onto P
```

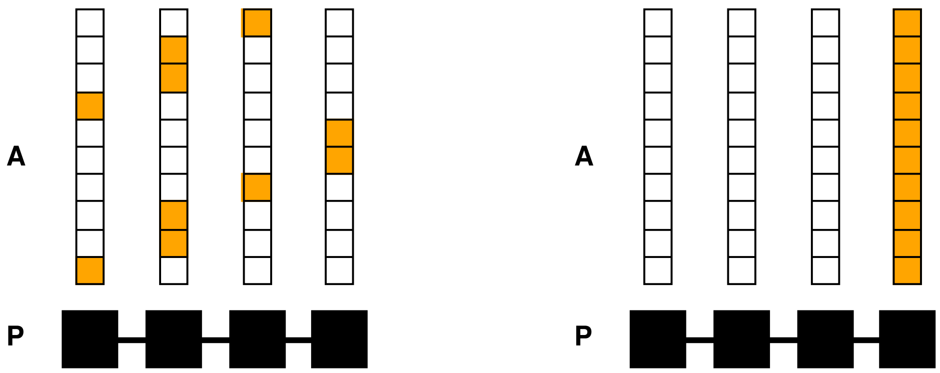


Figure 3.2: Examples of strange mappings allowed by HPF

```
chpf$ distribute A(cyclic)
      if (A(1).lt.0.) then
chpf$   redistribute A(block)
      endif
! now the compiler cannot know whether
! A is block- or cyclic-distributed.
      B(:) = A(:)
```

Figure 3.3: Ambiguity of re-mappings

```
      subroutine foo(A)
      real A(10)
chpf$ inherit A
! the compiler cannot know about
!   A mapping.
      A(:) = ...
      ...
```

Figure 3.4: `inherit` Un-knowledge

```
  SEQUENTIAL
! x on P1
! y on P2
x = f(y)
y = g(x)
```

```
        MESSAGE-PASSING
! P1              ! P2
t = rcv P2        snd y to P1
x = f(t)
snd x to P2       t = rcv P1
                  y = g(t)
```

```
         GET PUT SYNCHRO
! P1                  ! P2
t = get y on P2
x = f(t)
put x as s on P2
synchro               synchro
                      y = g(s)
```

Figure 3.5: Message-Passing/Get Put Synchronize

independent forall $(i \in \mathcal{I})$ A($f(i)$) = B($g_1(i)$) + B($g_2(i)$)

Figure 3.6: Generic parallel assignment

3.2.3 Target

Different target programming models and architectures require different techniques and optimizations. First, SIMD (Single Instruction Multiple Data) or MIMD (Multiple Instruction Multiple Data) architectures may be targetted. For MIMD architectures, a SPMD (Single Program Multiple Data) program parametrized by the processor identifier must be generated. The rationale for generating few programs (typically one or two) to be run identically in parallel on the nodes of the target machine is obvious: it is the only scalable (*i.e.* extensible to a large number of processors) technique. The underlying communication model may be message passing *à la MPI* or a get, put and synchronize model (Figure 3.5), requiring a different style of optimization. The communication to computation speed ratio as well as the communication latency and bandwidth often differ, changing the optimal choice for compilation. Some machines have special hardware for particular communication patterns (broadcasts, synchronizations, reductions, shifts) which is significantly more efficient [207] than simple point to point communications, so that an efficient compiler *must* use them. The diversity of target architectures results in a necessary diversity or adaptability of the compilation techniques.

3.2.4 Outline

This paper focuses on issues related to the efficient management of HPF's very general regular mappings, allowing alignment strides and general cyclic distributions for distributed memory MIMD architectures. The compiler must switch from the global name space and implicit communications of HPF to the local name space and explicit communications of the target SPMD message-passing programming model. The efficiency of the allocation, addressing, communication and temporary managements are the key points for achieving acceptable performance.

3.3 Compilation overview

Let us focus on the generic example in Figure 3.6 where `A` (lhs) is assigned and `B` (rhs) is used within a parallel loop. Iteration set $\mathcal{I}$ and access functions g_1 and g_2 may be arbitrary. The correctness of the parallelism requires that f is injective on $\mathcal{I}$. From such a loop the compiler must (1) assign iterations to processors, (2) determine the resulting communications and (3) generate code for the SPMD message passing model. If these tasks cannot be performed by the compiler, they are delagated to the runtime system, with the inefficient but necessary *runtime resolution* technique. Each point is discussed in the following.

Processor 0	Processors 1-2	Processor 3
	send y(0)	send y(0)
do j = 0:2	do j = 0:2	do j = 0:2
x(j) = y(j+1)	x(j) = y(j+1)	x(j) = y(j+1)
receive(temp)	receive(temp)	
x(3) = temp	x(3) = temp	

(a)

Processor p
if (p $\neq$ 0) send y(0)
do j = 0:2
x(j) = y(j+1)
receive(temp)
if (p $\neq$ 3) x(3) = temp

(b)

Figure 3.7: Hand-made and SPMD code with OCR

3.3.1 Compiling for the SMPD model

When considering the SPMD execution model as the target of the compilation process, the most important fact is that the generated code must be *generic*: the code is identical on all processors, necessarily parametrized by the processor identifier. When considering distributed memory architectures as the target of this SPMD code, the most important characteristic is the memory hierarchy: access to remote data is much slower than access to local data. Finally, when considering a message passing model for communication, the memory hierachy has qualitative effects: access to remote data requires knowing which processor owns the data. From all these features, the compiler makes fundamental choices:

- mapping of computations, that is assigning iterations ($\mathcal{I}$) to processor(s);
- allocating a local representation for the distributed data on each processor;
- code generation scheme, involving the actual translation from global to local adresses and the scheduling of communications with respect to computations.

3.3.2 Owner Computes Rule

For assigning computations to processors, compilers generally rely on the *owner computes rule* (*OCR*). The rule states that the *computation* for a given array assignment must be performed by the *owner* of the assigned reference. This simplifies the generated code: once a value is computed on a processor, it can be stored directly without further communication. The *OCR* is not necessarily the best choice and possible alternative solutions have been investigated [32]. Making this choice visible to some extent to the user is proposed in some languages, with the ON clause. Given the assignment, the compilation problems are basically identical with all schemes, hence we assume the *OCR* in the following.

Figure 3.7 examplifies the application of the *OCR* to the compilation process. Arrays X and Y are supposed to be distributed `block(4)` onto four processors. In both cases, x and y are the local arrays implementing the global arrays X and Y, and are accessed by local addresses. The code in (a) is tailored to the specific processors, not SPMD. The generic code in (b) involves some guards.

3.3.3 Run-time resolution

The simplest compilation scheme [45] uses *run-time resolution* of accesses to distributed data (*RTR*). Each processor executes the whole control flow of the program, but each reference to distributed data is guarded by a locality check. Communication is then generated between the owner and any processors that may need the data (see Figure 3.8). This technique can be implemented simply [59] as a rewriting of the syntax tree of the original program.

Run-time resolution is highly inefficient, and the inefficiency increases with the degree of parallelism. First, each process scans the entire iteration space and executes the whole control flow of the program. Worse, for each index, a guard is evaluated; the guard is pure overhead compared to the sequential execution. Next, as the scheme is based on global indices, a large number of address translations from global to local indices must be performed at run-time. Finally, the communication is not vectorized but performed in an elemental level. Checking ownership is simple when addresses are explicit functions of loop indices, for instance affine functions; from i, $f(i)$ is computed, and the ownership is derived from the formulas in Figure 3.12. When adresses are themselves distributed array references, the *RTR* scheme must be applied twice.

Clearly, optimizations are necessary. However, possible optimizations will depend heavily on the nature of the application. If the references are regular, aggressive compilation analysis is possible. Basically, a large fraction of the guards and address translations will shift from execution time to compile time. For irregular applications, the run-time resolution overhead can be amortized either over repeated executions of a parallel loop, for instance if the parallel loop is enclosed in a sequential one, or across code parts that require the same address translations. It must be stressed that, if an application is essentially dynamic and irregular, it cannot benefit from more than a limited number of optimizations, with respect to run-time resolution. Therefore choices have to be devised at the algorithmic level, or even at the application level. For instance, in the field of parallel finite elements problems, if one asks for programming in HPF, domain decomposition methods may be easier and more effective than global sparse computations.

```
// source code: A[i] = f(B[j])

Oa = owner(A[i]);
Ob = owner(B[j]);
if (Oa==me) {
  tmp = (Ob==me)? B[j]: recv(Ob);
  A[i] = f(tmp); // computation!
} else {
  if (Ob==me) send(B[j],Oa);
}
```

notations:

- `me`: processor identifier
- `tmp`: temporary local data
- `owner(x)`: the owner of `x`
- `send(x,p)`: non block. send `x` to `p`
- `recv(p)`: blocking receive from `p`

Figure 3.8: Runtime resolution on a single statement

3.3.4 Index and communication sets

The compiler decision for the *OCR* assigns loop iterations to processors. This can be formalized into a parametric set of iterations, and the resulting communications into sets of array indices that must be communicated between processors. For clarity, the generic example in Figure 3.6 where `A` (lhs) is assigned and `B` (rhs)

send $\mathcal{S}(p, *)$
receive $\mathcal{S}(*, p)$
for $i \in \mathcal{C}(p)$ ...

Figure 3.9: Simple SPMD code

send $\mathcal{S}(p, *)$
for $i \in \mathcal{C}_{\mathrm{local}}(p)$ ...
receive $\mathcal{S}(*, p)$
for $i \in \mathcal{C}_{\mathrm{remote}}(p)$ ...

Figure 3.10: Overlap SPMD code

is referenced within a parallel loop is considered. An example is presented in Figure 3.13. For each processor and array the mapping directives define a set of owned array references:

$$\mathcal{O}_{\mathtt{A}}(p) = \{a | \mathtt{A}(a) \text{ is mapped on } p\}$$

Each processor is assigned a set of iterations to compute with the *OCR*:

$$\mathcal{C}(p) = \{i \in \mathcal{I} | f(i) \in \mathcal{O}_{\mathtt{A}}(p)\}$$

In turn, this set defines the set of references needed by a processor to perform the computation, that is the set of viewed elements. In our example:

$$\mathcal{V}_{\mathtt{B}}(p) = \{a | \exists i \in \mathcal{C}(p), a = g_1(i) \text{ or } a = g_2(i)\}$$

From these set one can define the sets of array references that are to be communicated from processor p to processor p':

$$\mathcal{S}_{\mathtt{B}}(p, p') = \mathcal{O}_{\mathtt{B}}(p) \cap \mathcal{V}_{\mathtt{B}}(p')$$

3.3.5 Code Generation

Several variations are possible in defining the previous high level sets, each directed towards a particular code generation scheme. In any case, the SPMD code will have to *scan* these sets, that is enumerate all their elements. The simple code presented in Figure 3.9 performs first the communication and then the computations ($*$ stands for *all possible values*). The code presented in Figure 3.10 from [163] overlaps local computations and communication by splitting $\mathcal{C}$. Computations involving *remote* references are delayed. The purpose of this split scheme is *latency hiding*, by performing the *local* computations while the messages are being transmitted on the network. However, it is shown in Section 3.6 that the split scheme creates other penalties that may exceed its benefits.

In both schemes, the messages may be built one after the other for each target processor, or all at the same time; the latter may simplify the enumeration part at the price of having to wait for all the messages to be ready before issuing the actual communication. This distinction also holds for the receiving messages. Thus the very same sets and high level SPMD code can lead to very different communication behaviors.

Parallel loops more general than the example of Figure 3.6, in particular forall loops including many statements, may be split into a sequence of parallel one-statement loops, or compiled as a whole. In the latter case, the iteration set $\mathcal{C}$ is fundamental. The union of the statement sets of each statement in the loop forms the Reduced Iteration Set (RIS) of the loop [232]. If the $\mathcal{C}$ of a particular statement in the loop is a proper subset of the RIS, then the corresponding statement must be guarded.

In [200] a general calculus for building these sets is presented. However the main issue is the practical management of these parametric sets at compile- and run-time. What data structure? What operators? Cost and approximations? Code generation issues? The choice of data structures ranges from array sections to polyhedra for regular references, and includes lists for irregular applications. It should be noted that the more general the data structure is, the more expensive the operators are, but also the fewer the needed restrictions are on $\mathcal{I}$, f, g_1 and g_2. A typical restriction is that for general polyhedron-based techniques [5, 172] the iteration space and references are described by affine functions.

Desirable features for such a data structure are:

- minimality with respect to the described set, to avoid guards;
- minimality with respect to the number of operations involved; in particular, if an array reference of the SPMD code is expressed in the native global framework, it requires an expensive runtime translation to local addresses.
- closure under common operators (union, intersection, difference, etc.)
- easy to scan in the SPMD programming model; loops are the best choice, lookup tables must be kept to a moderate size.

It is not at all obvious that a perfect data structure exists. Thus all compilation techniques must make choices between these properties, and afford the implied approximation and other costs.

3.3.6 Optimizations

Overhead in SPMD code with respect to sequential code has two sources: extra computation and control, and communication. The first issue is tackled in the next section; this section sketches communication optimizations. Clearly, communication is pure overhead and must be treated with special care, because of its overwhelming cost. Optimization occurs at two levels: local optimization deals with the communication created by a given parallel loop; global optimization considers the analysis and scheduling of communications across the whole code, at the intra- and inter-procedural level.

At the local level, because the start-up cost is much higher than the per-element cost of sending a message, a major goal for a compiler is to combine messages from a given source to a given destination. For parallel loops, this results in moving communication outside the loop. This leads to three optimizations [131]. *Message vectorization* packs the references associated with the same right-hand-side expression. *Message coalescing* packs the references associated with different right-hand-side expressions. *Message aggregation* allows reuse of previously sent data. Vectorization and coalescing reduce the number of messages, while aggregation diminishes the overall volume of communication. It has been suggested [112] to extend the concept of vectorization to collective communication.

```
      real A(0:42)
chpf$ template T(0:127)
chpf$ processors P(0:3)
chpf$ align A(i) with T(3*i)
chpf$ distribute T(cyclic(4)) onto P
      A(0:42:3) = ...
```

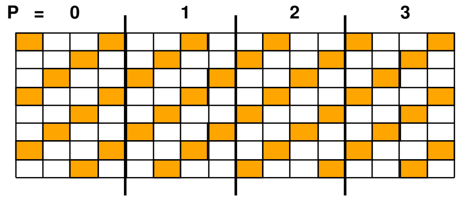


Figure 3.11: Mapping example from [52]

A comparative study of the respective impact of each optimization has been made [241]. Vectorization was found to be the most important, and the most consistent across the benchmarked codes. Also coalescing is very useful to typical communications arising from references to a same array in stencil computations. However, some applications may also benefit a lot from aggregation. For instance, some molecular dynamics code use lists of interacting particles. As interaction is guided by physical distance, if particles A and B interact with particle C, they will probably interact with each other. If particles A and C are mapped on the same processor and B is remote, the data for B will be sent to interact with C, and does not need to be sent once more for A.

The placement of communication within the program answers the following question: where should a communication take place in order to minimize the overall communication, remove useless communication, and minimize latency. Consider the code fragment:

```
A(:,5) = B(:)
forall (j=1:10) A(:, j) = B(:)
```

Without analysis, two groups of communication is necessary: the first statement copies `B` to the owner of column 5 of `A`, and the loop copies `B` on each column of `A`. If the second communication is moved above the first assignment, the first communication may be removed. A data-flow approach is well-suited to this problem [2, 245, 51, 113]. Through advanced program analyses, the source of a value in a program is identified (last write tree, array data flow graph) and the communication is issued accordingly.

3.3.7 Affine framework

Let us introduce the general mapping example in Figure 3.11. Array `A` is mapped onto a processor arrangement `P` through a stride 3 alignment and a general cyclic-4 distribution. The picture shows the resulting mapping of the array (shaded elements) and the template onto the 4 processors. The 128 template elements cycle over the processors: each row is a cycle of 16 template elements. Contiguous blocks of 4 elements are attached to each processor. For instance `T(18)` is on processor $\psi = 0$, at cycle $\gamma = 1$ and offset $\delta = 2$.

From this picture the mapping geometry of array elements seems regular. This regularity comes from the affine constraints that can be derived:

`A(0:42)`	$0 \leq \alpha \leq 42$
`T(0:127)`	$0 \leq \theta \leq 127$
`P(0:3)`	$0 \leq \psi \leq 3$
`align A(i) with T(3*i)`	$\theta = 3\alpha$
`cyclic(4)`	$0 \leq \delta \leq 3$
`distribute T(cyclic(4)) onto P`	$\theta = 16\gamma + 4\psi + \delta$

Integer solutions to this set of equalities and inequalities provide an exact description of the HPF mapping, that is the array element (α), the processor onto which it is distributed (ψ), the cycle it belongs to (γ) and so on. The first inequalities just reflect the Cartesian declaration constraints. Then the alignment constraint links the array element α to its corresponding template cell θ. The block width is expressed on the block offset δ. Finally the distribution equation decomposes the template space θ into the corresponding processor ψ, cycle γ and offset δ.

Such an affine formalization is presented in [5]. All mapping issues, including alignment strides and cyclic distributions as in this example, but also partial replication and multidimensional distributions can be included. Moreover, the elements accessed by an array section (independent affine functions of one index) and also by some loop nests, as well as the allocation and addressing issues on a given processor, can be described within the same framework. For the array assignment in Figure 3.11, after introducing index variable i, we have:

`0:42:3`	$0 \leq i \leq 14$
`A(0:42:3)`	$\alpha = 3i$

global to distributed:	$\gamma = \lfloor \theta/pk \rfloor, \psi = \lfloor \theta \bmod pk/k \rfloor, \delta = \theta \bmod k$
distributed to global:	$\theta = pk\gamma + k\psi + \delta$

Figure 3.12: HPF distribution functions

The very simple example of expressing the distribution function shows the benefits of an affine framework. Figure 3.12 gives the conversion formulas for an array indexed by θ distributed cyclic(k) on a p processor set (all arrays are considered indexed from 0). Often, the compilation process will have to translate the θ framework into the (γ, ψ, δ) framework. The simplest example is applying the *OCR*, but computing local addresses is also closely related to this translation. For this purpose, the formulas in the second column are not convenient, because symbolic computations using integer division are restricted. An affine framework is a practical alternative: the equation in the first column describes the same $(\theta, \gamma, \psi, \delta)$ set as the equations of the second column, if we add that δ and ψ are bounded ($0 \leq \delta \leq k-1$ and $0 \leq \psi \leq p-1$). Instead of constraining the way the compiler must translate global addresses to local addresses, the constraints are expressed in a non directive way. The compiler can take advantage of this to compute the local addresses so as to respect these constraints, but not necessarily to start its computation from global addresses.

3.3.8 The alignment/distribution problem

When an array is its own template, it has a regular HPF distribution (`cyclic(k)` or `block(k)`). Allocating the corresponding memory space is easy if the processor extents are known at compile time: roughly, the local array size is the block size times the number of cycles. Simple and efficient local addressing schemes can be derived.

Problems come from alignment. In a pseudo-mathematical formulation, one can state that HPF distributions are not closed under alignment: combining an HPF distribution and an HPF alignment does not always result in an HPF distribution. Figure 3.11 and Figure 3.2 show such examples: the distributions may be considered as an irregular (but static) mapping, because the block size varies from processor to processor, and worse, inside each processor.

What new issue does these complex distributions raise? The code generation issues are solved by enumerating complex sets at runtime to perform computations and communications. However, even if such techniques can be reused, the problem is actually new: here the compiler must *allocate* memory space for these irregular blocks. The allocation scheme may be optimal in memory space: all the elements of an array placed on a given processor are allocated contiguously, possibly at the expense of efficiency in physical address computation. At the other extreme, one can consider allocating the whole template, then wasting memory space, but resuming to a regular case. Typical allocation schemes are detailed in Section 3.6.

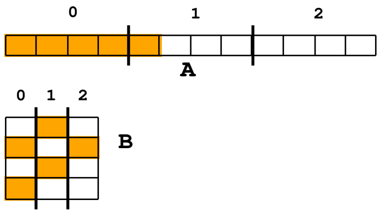
3.4 Regular References

The sets describing computations and communications presented in Section 3.3 must be enumerated on each node so as to actually perform the computations and communications. We have sketched before some criteria for this enumeration to be efficient. Much work has been devoted to the enumeration problem, with various assumptions about the mapping (block, cyclic, general cyclic distribution, with or without alignment stride), the access (shift, array section or full loop nest) and with various goals (enumerating the local iterations, or generating communication). In this section, where local addresses are concerned, the underlying space is the template of each array: physical addressing will be considered in Section 3.6. This means that the expected result is scanning δ, ψ and γ variables, that is dealing with the set in the abstract processor, block and cycle space. Three approaches can be identified: *closed forms*, *automata* or *linear integer programming* techniques such as Diophantine equations and polyhedron theory. They are presented in the following, with an emphasis on the assumptions needed by the techniques.

3.4.1 Closed forms

The closed form approach consists of finding parametric descriptions of the different sets. A closed form is a formula that just needs to be evaluated once missing values are known. It is used at runtime to compute the different sets or to issue the communications and to perform the computations. Many important and practical cases have closed form solutions: For instance, block-distributed arrays and shift

```
      array A(0:11)
      array B(0:11)
chpf$ processors P(0:2)
chpf$ distribute A(block) onto P
chpf$ distribute B(cyclic) onto P
      A(0:4) = B(1:9:2)
```



$p =$	0	1	2
$\mathcal{O}_{\mathtt{A}}(p)$	0:3	4:7	8:11
$\mathcal{O}_{\mathtt{B}}(p)$	0:9:3	1:10:3	2:11:3
$\mathcal{C}(p)$	0:3	4	-
$\mathcal{S}_{\mathtt{B}}(0, p)$	3	9	-
$\mathcal{S}_{\mathtt{B}}(1, p)$	1:7:6	-	-
$\mathcal{S}_{\mathtt{B}}(2, p)$	5	-	-

$$\begin{aligned}
\mathcal{D}_{\mathtt{A}} &= 0:4 \\
\mathcal{D}_{\mathtt{B}} &= 1:9:2 \\
\mathcal{D}_{\mathtt{B}} &= 2\mathcal{D}_{\mathtt{A}} + 1 \\
\\
\mathcal{C}(p) &= \mathcal{O}_{\mathtt{A}}(p) \cap \mathcal{D}_{\mathtt{A}} \\
\mathcal{V}_{\mathtt{B}}(p) &= 2\mathcal{C}(p) + 1 \\
\\
\mathcal{S}_{\mathtt{B}}(p, p') &= \mathcal{O}_{\mathtt{B}}(p) \cap \mathcal{V}_{\mathtt{B}}(p')
\end{aligned}$$

Figure 3.13: Closed form example: Computation and communication sets

accesses communications of the borders is vectorized outside of the loop nest after guard elimination [260] (Figure 3.24).

The closed form approach generally applies to array sections references. An *array section* is a regular set of integers described by three parameters: the first value, the last value, and an intermediate stride (the default stride value is 1):

$$(l : u : s) = \{i | \exists j \geq 0, i = l + sj, (s > 0 \wedge i \leq u) \vee (s < 0 \wedge i \geq u)\}.$$

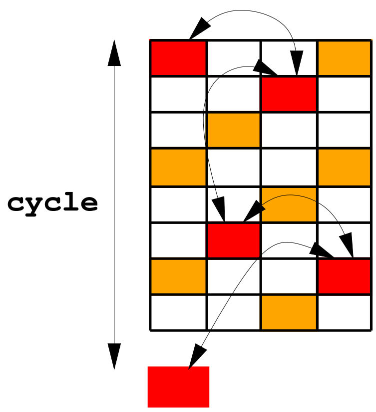
Array sections are closed under intersection ($\cap$) and affine transformation ($a.S + b$ where a and b are integers and S is a section).

The reason for the interest in array sections is that they are powerful enough to express the different sets [161] and their computations as outlined in Section 3.3.4 for array expressions. Also scanning an array section by a `do` loop is trivial. First, the index sets are array sections on block or cyclic distributed dimensions. Figure 3.13 shows an example. In general, if a one-dimensional array `A` of size n is distributed `block(`k`)` and array `B` is cyclic-distributed onto q processors then with $0 \leq p < q$:

$$\begin{aligned}
\mathcal{O}_{\mathtt{A}}(p) &= (pk : \min(pk + k - 1, n)) \\
\mathcal{O}_{\mathtt{B}}(p) &= (p : n : q)
\end{aligned}$$

where p remains to be evaluated. Since the only operators used to build the compute and communication sets are the intersection and affine transformation, and since the original sets (owned elements and accessed array sections) are already array sections, the resulting sets are array sections.

This basic technique was extended [228, 117, 25] to manage general cyclic distributions. A general cyclic distribution is viewed as a block distribution on more *virtual* processors, which are then assigned to *real* processors in a cyclic fashion. This is the *virtual* cyclic view. The dual approach is to consider a cyclic distribution and then to group blocks of *virtual* processors. For alignment




```
// Delta holds the pattern of local
// steps, computed at runtime
Delta[4] = { 2, 5, 2, 3 } ;

// index in the pattern
i = 0 ;
a = initial ;

// loop to scan the pattern
while (a <= final) {
  // computation with address a
  ... ;

  // next location
  a += Delta[i] ;
  i = (i+1) % 4 ;
}
```

Figure 3.14: FSM-based enumeration

strides [147], the sets are computed at the template level and then compressed to remove the holes and generate local addresses.

This simple and parametric technique was quickly introduced in commercial compilers. However it is limited to array section accesses: simple triangular areas cannot be handled. The virtual processor extension results in many intersections to be performed at runtime, for each pair of virtual processors. From a practical point of view, the technique looks like the PARTI system (Section 3.5), *i.e.* communications are precomputed (on a dimension per dimension, processor per processor and array section basis) and used afterwards.

3.4.2 Finite State Machines

General cyclic distributions display periodic patterns on the cycle (γ) axis for regular section accesses: the processor and offset may become identical while enumerating some sets. For instance, in Figure 3.11, the access pattern for array section `A(0:42)` repeats each four cycles. Hence, the local enumeration loop has to be unrolled to some extent: for the references corresponding to the first period, tables are precomputed, or divide and remainder computations are performed on the fly; for the following references, the appropriate stride is used. The period depends on the distribution and alignement parameters.

The FSM technique looks for the basic pattern and uses an automaton to enumerate the elements. It was first suggested in [52]. The algorithm builds at runtime the required transition table, which is then used as the base of the scanning. The pattern of the example in Figure 3.11 is shown in Figure 3.14. The array elements are light shaded but the accessed ones (array section `A(0:42:3)`) are dark shaded.

The algorithm for building the transition table initially deals with an array section and a general cyclic distribution. The extension to alignment strides simply applies this algorithm twice. The generalization to multidimensional arrays is straightforward. The technique handles simple regular communications, such as shifts. The initial algorithm was improved to generate the transition tables faster [130, 152, 238]. In [153] the method is extended to enumerate affine and coupled subscript accesses for general cyclic distributions. Such extensions typically require several tables per array dimension to handle the various indexes appearing in an affine subscript.

These automaton-based techniques are fast at enumerating local elements. They handle directly the local addresses generation with maximum cache benefits, while other techniques generate higher level code which depends on subsequent optimizations for handling actual addresses efficiently. They are especially targeted to array sections. Extensions are needed to handle general communication sets. The automaton technique solves one of the HPF compilation problems: enumerating sets efficiently. However, these sets (array section on general cyclic distributions) are limited with respect to the description of general communications involving arbitrary HPF mappings. Also the transition tables must be managed at runtime. Another view one can have from the FSM approach is that it allows to accelerate enumeration of complex sets by precomputing the patterns. However, generating the transition tables is as complicated (or simple) as enumerating the required elements directly.

3.4.3 Polyhedra and lattices

Other techniques rely on parametric sets defined by affine constraints and generate code directly from this representation. Such a representation can be built for any parallel loop with affine access functions and iteration domain. It follows from the affine framework sketched in Section 3.3.7: systems of linear equalities and inequalities are derived from the HPF array mapping specification, the iteration set and the array accesses. These sets define polyhedra. The integer points of these polyhedra express the computations and the necessary communications. Enumerating these sets so as to actually perform the computation or generate communications falls under the general category of scanning polyhedra (or projections of polyhedra).

Let us consider the assignment `B=A` in Figure 3.15. The distributions involve partial replication along some dimension of the source and target processor arrangements. The mapping is depicted in Figure 3.17. From this code and following the affine framework one can derive the systems in Figure 3.16. It incorporates the mapping equations, a local addressing scheme (β), the array reference (a simple $\alpha'_1 = \alpha_1$) and an additional equation to express the desired load-balance to benefit from replication. From these equations communication code as presented in Figure 3.18 may be generated. It scans the solutions to the system so as to generate the required communication and performs the assignment. General examples can be handled with this approach.

For generating such a code, many degrees of freedom remain and various approaches have been suggested. The generic sets may be decomposed [247, 104] into linear independent Diophantine equations that are solved using standard methods. Full HPF mappings for array sections [247] and full affine accesses and iteration space for cyclic distributions [104] are considered. The latter paper performs unimodular changes of basis on general affine problems to build simpler array section problems. Moreover, the enumeration scheme is directed by communication vectorization, leading to a particular order of variables in the enumeration scheme. A linear Diophantine equation is:

$$\{(i,j) \in Z^2 | ai = bj + c\}$$

```
      real A(20), B(20)
c
c source mapping
c
!hpf$ processors Ps(2,2,2)
!hpf$ template Ts(2,20,2)
!hpf$ align A(i) with Ts(*,i,*)
!hpf$ distribute Ts(block,block,block)
!hpf$.     onto Ps
c
c target mapping
c
!hpf$ processors Pt(5,2)
!hpf$ template Tt(20,2)
!hpf$ align B(i) with Tt(i,*)
!hpf$ distribute Tt(cyclic(2),block)
!hpf$.     onto Pt
c
c assignment:
c
      B = A
```

Figure 3.15: Distributed array assignment

`A` mapping and addressing (β_1):

$$\begin{array}{l} 1 \leq \alpha_1 \leq 20,\ 1 \leq \theta_1 \leq 2,\ 1 \leq \theta_2 \leq 20, \\ 1 \leq \theta_3 \leq 2,\ 1 \leq \psi_1 \leq 2,\ 1 \leq \psi_2 \leq 2, \\ 1 \leq \psi_3 \leq 2,\ 1 \leq \delta_2 \leq 10, \\ \alpha_1 = \theta_2,\ \theta_2 = 10\psi_2 + \delta_2 - 10,\ \beta_1 = \delta_2 \end{array}$$

`B` mapping (OCR) and addressing:

$$\begin{array}{l} 1 \leq \alpha'_1 \leq 20,\ 1 \leq \theta'_1 \leq 20,\ 1 \leq \theta'_2 \leq 2, \\ 1 \leq \psi'_1 \leq 5,\ 1 \leq \psi'_2 \leq 2,\ 1 \leq \delta'_1 \leq 2, \\ \alpha'_1 = \theta'_1,\ \beta'_1 = 2\gamma'_1 + \delta'_1 - 3, \\ \theta'_1 = 10\gamma'_1 + 5\psi'_1 + \delta'_1 - 15 \end{array}$$

reference and load-balance

$$\alpha_1 = \alpha'_1,\ \psi'_1 = 4\lambda + 2\psi_3 + \psi_1 - 3$$

Figure 3.16: Full copy system

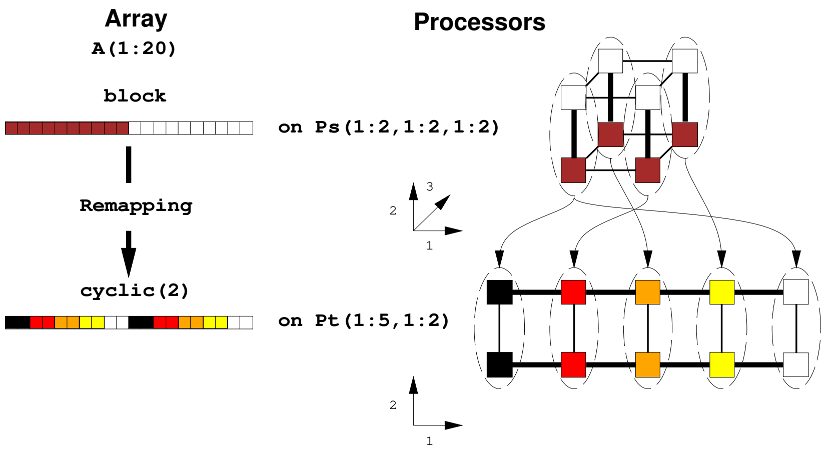


Figure 3.17: Full copy of distributed arrays

```
IF (I AM IN(Ps)) THEN
  ψ = (ψ1,ψ2,ψ3) = MY ID IN(Ps)
  DO λ = 0, (7-ψ1-2*ψ3)/4
    ψ'1 = -2+ψ1+2*ψ3+4*λ
    index = 0
    DO β1 = 2*ψ'1-1, 2*ψ'1
      BUFFER(index++) = A(β1)
    ENDDO
    CALL BROADCAST(BUFFER,Pt(ψ'1,:)) ! send
  ENDDO
ENDIF

IF (I AM IN(Pt)) THEN
  ψ' = (ψ'1,ψ'2) = MY ID IN(Pt)
  DO ψ1 = 1, 2
    DO ψ3 = 1, 2
      DO λ = (5-ψ1-2*ψ3+ψ'1)/4, (2-ψ1-2*ψ3+ψ'1)/4
        DO ψ2 = 1, 2
          IF (Ps(ψ)<>Pt(ψ')) THEN ! receive and unpack
            CALL RECEIVE(BUFFER,Ps(ψ))
            index = 0
            DO β'1 = 2*ψ2-1, 2*ψ2
              B(β'1) = BUFFER(index++)
            ENDDO
          ELSE ! local copy
            DO β1 = 2*ψ'1-1, 2*ψ'1
              β'1 = β1+2*ψ2-2*ψ'1
              B(β'1) = A(β1)
            ENDDO
          ENDIF
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDIF
```

Figure 3.18: Generated SPMD code outline

It is related to the intersection of array sections. However, unlike the closed form approach, the decomposition process into these simple equations is systematic.

This decomposition needs not to be performed explicitly [5, 172, 21]. Generic algorithms for automatically generating codes that enumerate solutions to systems of linear equalities and inequalities, based on Fourier elimination [136, 7, 170] or on a parametric simplex [86] are used. These algorithms are potentially exponential, but practical cases are usually handled quickly. The generated code efficiency depends on the ability of the code generation algorithm. Polyhedra are scanned with loop nests involving sometimes complex bounds (min, max, integer divisions) that require special optimizations.

Because these techniques are based on parametric sets, communication and enumeration issues are quite natural; in theory they do not require special attention. However, practice shows that care must be taken to avoid inefficient enumerations. Such techniques clearly address the need for compile-time management of index sets. Some implementations of polyhedron based techniques have been developed for general HPF mappings, for I/O related communications [61] and for remappings [68], with associated optimizations that takes advantage of load balancing and broadcast opportunities.

3.5 Irregular problems

In order to compile general parallel loops with irregular accesses to distributed variables, it is necessary to rely on the run-time to compute the execution set $\mathcal{C}(p)$ and the communication sets $\mathcal{S}_{\mathsf{B}}(p, p')$.

We describe here how these accesses are handled in the PREPARE compiler [248]. In the PREPARE compiler, all HPF data-parallel constructs are transformed into a standard construct (the ArrayAssign) in the intermediate representation level. The compilation scheme used for an ArrayAssign [43] generates a three-phase code: the **work distributor**, the **inspector**, and the **executor**. The work distributor determines how to spread the work (iterations) among the HPF processors. The inspector analyzes the communication patterns of the loop, computes the description of the communication, and derives translation functions between global and local index spaces, while the executor performs the actual communication and executes the loop iterations. Each phase relies on run-time functions embedded in the PARTI+ library.

3.5.1 The Parti+ library

The purpose of the PARTI and PARTI+ libraries is to compute non-local accesses at run-time and to perform global exchanges needed to access non local data. The PARTI+ library is an extension of the PARTI library developed at the University of Vienna ; it supports all HPF distribution and alignment types and operations on all Fortran 90 intrinsic data types; moreover, a part of the PARTI kernel was reviewed and an optimized version is implemented in PARTI+. The three main functions of PARTI+ are described below:

cview

The cview routine computes, for a given distributed array, a *schedule* that is used to bring in the non-local data accessed by the calling SPMD process within the body of an ArrayAssign handled by the distributor-inspector-executor technique. The computed schedule is a structure that contains the necessary information to identify the locations in distributed memory from which data is to be accessed. This routine also computes the total number of the distant accesses (offproc) performed by this process, and generates an indirection table (loc) that is used to transform global references into local memory addresses. While constructing the indirection table loc, the cview routine filters out the duplicate references.

gather

The gather routine performs a global exchange between all the calling processes. The data to be exchanged are characterized by the schedule passed as a parameter (this schedule should have been previously computed by the cview routine). At the end of this exchange, all the distant data accessed by a given process within the corresponding parallel loop are locally available to this process: the distant values are stored in the buffer part of the extended local segment. Once these distant elements are received and stored, the array elements can be accessed via the local reference list (loc) that is previously computed by the cview routine.

scatter

The scatter routine performs a global scatter exchange between all the calling processes. The data to be exchanged are characterized by the schedule passed as a parameter (this schedule should have been previously computed by the cview routine). At the end of this exchange, all the non-local data written by a given process within the corresponding parallel loop[1] are copied back to their owners (i.e. the processes to which these elements were assigned by the data distribution specification): the distant values were stored in the buffer part of the extended local segment during their computation. The array elements were accessed via the local reference list (loc) that was previously computed by the cview routine.

3.5.2 Generated code

We describe here the main characteristics of the generated code for the example depicted in Figure 3.19

Prologue

The generated code begins with the initialization of the mapping descriptors associated with the distributed arrays A and B: the information about local segment size, dimension, shape, distribution and processor array is stored there. Then the local segment of B is allocated.

[1] for which a work distribution that deviates from the *owner-computes rule* has been applied

```
        INTEGER, PARAMETER:: N1 = 4, N2 = 8, M1 = 2, M2 = 4
        REAL, DIMENSION(N1,N2) :: A, B
!HPF$   PROCESSORS P(M1,M2)
!HPF$   DISTRIBUTE (CYCLIC,BLOCK) ONTO P :: A
!HPF$   DISTRIBUTE (BLOCK,CYCLIC) ONTO P :: B
        . . .
        FORALL (i=1:N1 , j=1:N2)
           A(fl(i),gl(j)) = B(fr1(i),gr1(j)) * B(fr2(i),gr2(j))
        END FORALL
```

Figure 3.19: Example of irregular accesses in an Array Assign statement

Work distributor phase

On each SPMD process p, the work distributor computes the *execution set*, $\mathcal{C}(p)$, i.e. the set of loop iterations to be executed by p. In our example, the work distributor applies the *owner-computes rule* (OCR) work distribution strategy, but the PARTI+ library provides features (the scatter routine) that allow us to deviate from this rule whenever another work distribution policy might be better. For the example in Figure 3.19, the execution set is given by:

$$\mathcal{C}(p) = \{(i,j) \mid (fl(i), gl(j)) \ \in \mathcal{O}_{\mathtt{A}}(p)\}$$

where $\mathcal{O}_{\mathtt{A}}(p)$ denotes the set of indices of elements of A that are stored on processor p.

This set is stored in an array called the *Execution Set Descriptor* (ESD). While filling the ESD, the total number of accesses to B performed by the process is computed and stored.

Inspector phase

The inspector performs a dynamic loop analysis. Its task is to describe the necessary communication by a set of *schedules* that control runtime procedures moving data between processes in the subsequent executor phase. The dynamic loop analysis also establishes an appropriate addressing scheme to access local elements and copies of non-local elements on each process.

The inspector phase starts by the allocation of two integer arrays, glob and loc. The glob array is filled with the value of index expressions of the global references to B performed by the process. Next, the inspecor fills the indirection table loc and evaluates the number of non-local elements of B (offproc) accessed by the calling process. A graphical sketch of how the cview routine works is depicted in Figure 3.20.

Executor phase

The executor is the third phase generated by the compilation scheme used to transform an irregular ArrayAssign. It performs the communication described by the schedules, and executes the actual computations for all iterations of $\mathcal{C}(p)$ as defined by the work distributor phase. The schedules control communication in

i_1 = esd[0][0], j_1 = esd[0][1], ..., i_{ess} = esd[ess-1][0], j_{ess} = esd[ess-1][1]

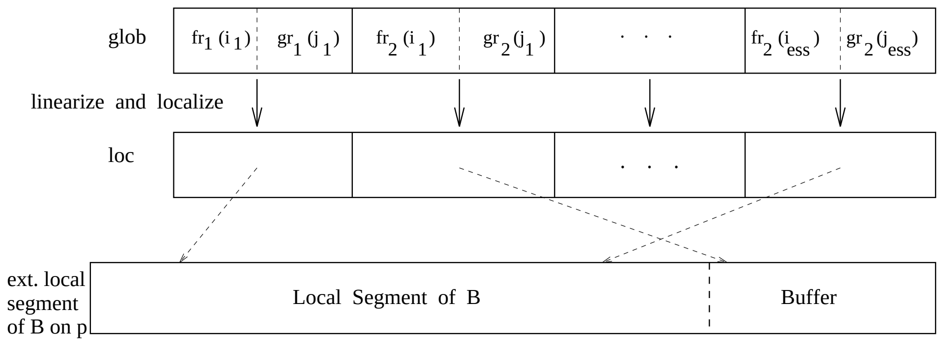


Figure 3.20: Relation between multi-dim. global references and linearized local references

such a way that execution of the loop using the local reference list accesses the correct data in local segments and buffers.

The executor phase first checks if the space allocated for the local segment of B is big enough to store the non-local elements; if not, more space for B is allocated. The non-local data are then gathered and stored in the extended local segment thanks to a call to the gather routine. Once the global exchange is done, the iterations assigned to the process by the work distributor are computed. The iteration space of this loop is characterized by the ESD constructed during the work distribution phase. All the read-accesses to B are performed through the indirection array loc. The *global-to-local* call, g2l(map$_A$, x, y), used in the index expression of the LHS, determines the offset of the element A(x,y) in the local segment of A associated with the calling process.

Epilogue

Finally, the intermediate array that stores the local reference list, the schedule and the ESD are freed.

3.5.3 The ESD and schedule reuse

To make ESD and schedule optimizations possible, the code generation was designed so that it is easy to recognize creation and use of an ESD and a schedule. The generation of Parti+ calls has been separated into a high-level part (that inserts high-level nodes in the intermediate representation) and a low-level part (that expands the abstraction to loops and routine calls). Between those two parts, a dedicated optimizing engine can be inserted. The Schedule Reuse Analysis Engine (SRAE) performs analysis on these high-level nodes and tries to hoist them outside of loops.

$$
\begin{aligned}
&\texttt{forall}\ (i \in \mathcal{C}(p))\\
&\quad \texttt{A}(f'(i)) = (g_1(i) \in \mathcal{O}_{\texttt{B}}(p)\ ?\ \texttt{B}(g_1'(i))\ :\ \texttt{T1}(t_1(i))) +\\
&\qquad\qquad (g_2(i) \in \mathcal{O}_{\texttt{B}}(p)\ ?\ \texttt{B}(g_2'(i))\ :\ \texttt{T2}(t_2(i)))
\end{aligned}
$$

Figure 3.21: Guarded references

$$
\begin{aligned}
&\texttt{forall}\ (i \in \mathcal{C}_{\texttt{ll}}(p))\ \texttt{A}(f'(i)) = \texttt{B}(g_1'(i)) + \texttt{B}(g_2'(i))\\
&\texttt{forall}\ (i \in \mathcal{C}_{\texttt{rl}}(p))\ \texttt{A}(f'(i)) = \texttt{T1}(t_1(i)) + \texttt{B}(g_2'(i))\\
&\texttt{forall}\ (i \in \mathcal{C}_{\texttt{lr}}(p))\ \texttt{A}(f'(i)) = \texttt{B}(g_1'(i)) + \texttt{T2}(t_2(i))\\
&\texttt{forall}\ (i \in \mathcal{C}_{\texttt{rr}}(p))\ \texttt{A}(f'(i)) = \texttt{T1}(t_1(i)) + \texttt{T2}(t_2(i))
\end{aligned}
$$

Figure 3.22: Compute set splitting

$$
\texttt{forall}\ (i \in \mathcal{C}(p))\ \texttt{A}(f'(i)) = \texttt{T1}(t_1(i)) + \texttt{T2}(t_2(i))
$$

Figure 3.23: All in temporary ...

3.6 Memory allocation and addressing

The compiler must use an allocation and addressing scheme for distributed arrays to switch from the global name space provided to the user by HPF to the local name space available on the distributed memory target architecture. This scheme should be driven by the mapping directives.

3.6.1 Issues

The memory allocation problem is twofold: the compiler must allocate permanent memory storage for the distributed arrays; it must also manage some temporary storage for remote references that are used in the computation process. For instance, in Figure 3.7, a permanent storage termed x is required for the local blocks of array X, and a temporary storage termed *temp* is used for the neighborhood of the corresponding block in array Y needed by the local computation.

Allocation and addressing trade-off

The following trade-off must be addressed: On the one hand, the amount of memory allocated to store a given array should be as small as possible, because memory is expensive. Moreover the amount of wasted memory for a given dimension is amplified by multidimensional distributions: if one third of each index space on each dimension of a 3D array is wasted, then 70% (19/27) of the allocated memory is wasted on the whole. On the other hand, the chosen scheme must allow fast local address computations, *i.e.* an efficient *global to local* translation, or at least its efficient incremental management within a loop nest.

Most parallel machines HPF was designed for are cache architectures. It is thus interesting to devise the allocation and enumeration schemes together so as to benefit from cache effects.

Temporary storage management

The second issue is the management of the temporary storage for remote references. The same constraints hold, but the memory can be reused. The temporary space needed may vary from one processor to another. Separating the local and temporary allocations can be unexpectedly costly: Let us consider again the generic example in Figure 3.6, and assume that two separate temporaries are used to store remote *rhs* references. Then for generating the new local computation loop, it must be decided for each reference whether it is local or remote. Guarding all references (Figure 3.21) is too costly. Moving these guards up by splitting the compute set (Figure 3.22) seems attractive but the process is exponential with respect to the number of references and is not necessary possible or efficient. Thus the only general technique (Figure 3.23) beyond overlap allocation which only suits simple cases is to manage local references as the temporary array or *vice versa*, *i.e.* to provide a local allocation scheme which stores remote references homogeneously [185] or to copy the local references into the temporary array [5].

3.6.2 Techniques

Let us focus on the allocation schemes suggested in the literature. These schemes are outlined for basic examples, and their pros and cons with respect to allocation overhead, fast addressing, and temporary storage are discussed.

Regular compression

```
      real A(10,10), B(10,10)
chpf$ processors P(2,2)
chpf$ align A with B
chpf$ distribute B(block,block) onto P
chpf$ independent...
c
c typical stencil computation
c
      DO j=...
        DO i=...
          B(i,j) =
     .      A(i,j)+A(i-1,j)+A(i,j-1)
        END DO
      END DO
```

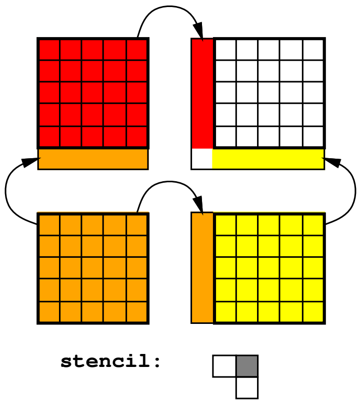
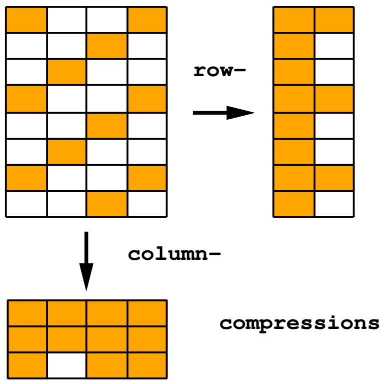


Figure 3.24: Allocation of overlaps

The first and most natural idea for managing local elements is to adapt the initial dimension by dimension Cartesian allocation in order to remove the regular holes coming from the cyclic distribution and alignment. For block-distributed dimensions, this may be done without any address translation [115]. For simple shift accesses the overlap technique [105] extends the local allocation to store the remote elements on the border, simplifying addressing. This suits stencil computations very well as shown in Figure 3.24.

For general cyclic distributions with alignments, both row- and column-compressions are possible [247, 226], as shown in Figure 3.25. The first one suits



2-D template view from $\mathtt{A}(\alpha)$:

$$\theta = 3\alpha$$
$$\gamma = \theta \div 16$$
$$\delta = \theta \bmod 4$$

Row-compression addressing (r, c):

$$r = \delta \div 3$$
$$c = \gamma$$

Column-compression addressing (r, c):

$$r = \delta$$
$$c = \gamma \div 3$$

Figure 3.25: Regular compression for local allocation

large cyclic distributions and the second one small cyclic distributions. The *global to local* functions involve integer divisions and modulus, but this is not an issue: most of the time no actual translation is needed and the *right* local address can be generated directly while enumerating local elements. This scheme is compatible with overlaps and may be improved to reduce the allocated memory but with a higher address computation cost [5].

Packed form

Instead of preserving a structured allocation, it has been suggested to pack local elements as much as possible [52]. This is obviously optimal from the memory allocation point of view. The issue is the addressing problem: accessing the local elements from the base is possible with a finite state machine which computes the relative offsets at runtime and exactly, but this does not correspond necessarily to a simple *global to local* formula. Also the management of such an allocation scheme for multi-dimensional distributions is not obvious, because the first element of each vector is not mapped regularly in memory. Thus the offset depends on the actual number of elements on the processor for each original vector of the matrix. Managing these issues across procedure boundaries seems also difficult because not much knowledge is available at compile time. This scheme suits only the FSM compilation technique presented in Section 3.4.2.

Software pagination

Classical paging systems for memory management can also be used for managing distributed arrays, with both compiler and runtime support [185]. The basic idea is to divide the array space into pages, and then to allocate on a given processor only the pages, or part of pages, on which it owns some array elements. This is depicted in Figure 3.26. Each array element at address i is associated a page

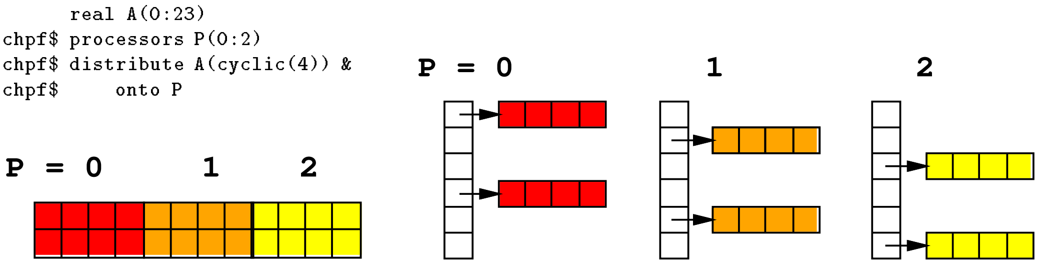


Figure 3.26: Software paging array management

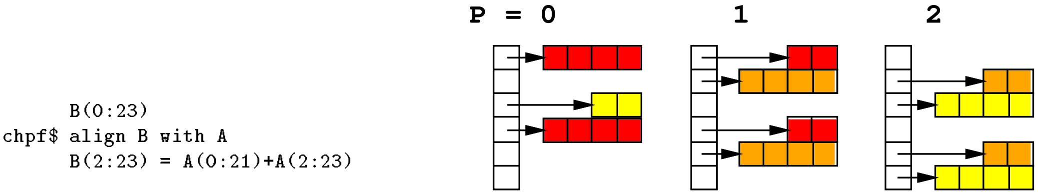


Figure 3.27: Temporary remote references management

number and offset from the page size s:

$$\begin{aligned} \text{page number:} \quad & n = i \bmod s \\ \text{page offset:} \quad & o = i \div s \end{aligned}$$

The runtime stores a vector of pointers to each page, some of which are allocated. The allocation overhead is small for block or large cyclic distributions, but can be as large as the array itself when considering cyclic distributions. Fast addressing is possible, because the page number and offset computations can use shift and mask operations. Locality and thus cache benefits is preserved on a page per page basis. The main advantage of such an allocation scheme is that it integrates naturally temporary management in an uniform way. Indeed, remote references can be stored within the pages they belong to on each processor, as depicted in Figure 3.27. Once transfered, the remote elements are stored as the owned array elements and accessed uniformly from the computation loop. The drawbacks are the allocation overhead to be paid for small cyclic distributions, alignment strides and also multidimensional distributions. Also this scheme has some impact on the runtime system, which much manage the allocated pages, and the addressing, although fast, may not be as optimizable as other simpler schemes.

Hash table based allocation and addressing have also been suggested for temporary reference management. The access cost is then uniform, but the random access in memory does not benefit much from caches. These allocation schemes are very difficult to compare one to the other, because they must be considered within the compilation process as a whole, involving enumeration issues. Also the very same scheme may be managed dynamically or statically, compromising subsequent standard optimizations or requiring special optimizations.

3.7 Conclusion

HPF is a promising language and is the focus of much attention from potential users. It may be a simple and efficient alternative to low level message passing programming. However HPF does not suit all applications. Even for data-parallel applications, a change in data-structures, algorithms and programming habits may be required. Users must understand the compiler technology to get high performance from their code, as they did with vector computers twenty years ago.

From the compiler point of view, the more knowledge available at compile time, the better the optimizations that can be performed and the better the resulting code. This knowledge may be provided in the language, but may also need to be extracted automatically from the program. This analysis may be defeated by bad programming habits, compromising the benefits expected from the parallel version of a program. The language must also allow the user to express his knowledge and advice about the application: some language features such as the `inherit` directive just provide the opposite, no information. Finally compiler technology must catch up with the HPF language, the design of which was beyond the know-how of its time.

We are optimistic that, with effort from users in writing clean data-parallel codes, from the HPF Forum in improving the language design where deficiencies have been detected, and from compiler writers in implementing new compilation techniques, HPF will be viewed as a success story.

Acknowledgment

We are thankful to Robert Schreiber and Monica Lam for their fruitful comments on an early version of this paper.

Deuxième partie

Conception du *High Performance Fortran*

Chapitre 1

Introduction

Abstract

This part presents some contributions to the HPF language conception. This first chapter introduces the next ones. It sumarizes in French the main ideas. Chapter 2 analyzes some features of the language, outlines deficiencies and suggests improvements. Chapter 3 describes a technique for managing overlaps at subroutine boundaries. Finally Chapter 4 is a formal proposal for addition to the language.

Résumé

Cette partie présente différentes contributions et discussions ayant trait à la définition du langage HPF. Ce premier chapitre introduit les suivants, et résume en français les principales idées. Le chapitre 2 analyse certains aspects du langage, souligne des déficiences et propose des solutions. Le chapitre 3 décrit une technique pour gérer les recouvrements lors des appels de procédures. Enfin le chapitre 4 est une proposition formelle d'adjonction au langage pour permettre d'associer plusieurs placements aux arguments d'une routine.

Le langage HPF a été conçu par de nombreuses personnes réunies en un comité, baptisé *Forum* pour l'occasion. Le *High Performance Fortran* (HPF) est le résultat de nombreux compromis. Il existe donc une marge d'amélioration au niveau de la conception, avec pour conséquence des facilités ou difficultés d'implantation et d'utilisation.

1.1 Analyse de la conception de HPF

Le chapitre 2 est une version légèrement étendue de [65]. Il propose d'abord des critères pour évaluer les caractéristiques techniques du langage. Il analyse ensuite certains aspects du langage par rapport à ces critères, et révèle quelques déficiences. Il développe enfin trois adjonctions ou corrections à la définition du langage pour résoudre les problèmes soulevés.

1.1.1 Critères d'analyse

Je suggère à la section 2.2 d'analyser les caractéristiques du langage par rapport à l'application, au langage et à son implantabilité :

Application

utilité pour améliorer les performances.

expressivité pour les besoins réels.

élégance des notations.

Langage

homogénéité des différentes notations.

orthogonalité entre caractéristiques.

complétude des cas décrits.

Implantation

simplicité (potentielle) de l'implantation générale.

efficacité en vue de hautes performances.

séparation de la compilation des routines.

1.1.2 Analyse d'HPF

Divers aspects du langage HPF sont ensuite analysés à la section 2.3, à l'aide des critères précédents :

Inherit permet de décrire un placement inconnu pour un argument d'une routine. Il assure donc simplement la complétude de la description du placement des arguments. Cependant il ne founit aucune information au compilateur sur le ou les placements en question, ce qui le rend à la fois difficile à implanter et potentiellement inutile pour les hautes performances.

Redistributions : bien intégrées dans HPF, mais leur orthogonalité par rapport au flot de contrôle rend possible des cas dégénérés où le compilateur n'a aucune information quant au placement d'une référence. Ces cas sont à la fois compliqués à gérer, et inutiles pour toute application réelle.

Forall n'est pas une boucle parallèle malgré les apparences. Sa sémantique est inhabituelle et il est parfois mal compris et mal utilisé.

Réductions : initialement suggérées avec une syntaxe interne à la boucle, à la fois inhomogènes, verbeuses et nécessitant plus de travail à la compilation, puis corrigées.

Directives de parallélisme : inhomogènes et non orthogonales. Les directives `new` et `reduction` ne peuvent être utilisées qu'attachées à une boucle marquée `independent`.

1.1.3 Suggestions d'amélioration

Enfin dans la section 2.4 je suggère des améliorations de la définition du langage pour régler les problèmes soulevés :

Assume : cette directive vise à remplacer `inherit` pour les cas où l'utilisateur connaît plusieurs placements parmi ceux pouvant atteindre les arguments d'une routine. Elle est détaillée plus avant au chapitre 4.

Redistributions : il est proposé d'interdire les cas dégénérés pour lesquels le placement d'une référence n'est pas connu statiquement du compilateur. Cette contrainte permet une implantation simple et n'interdit aucun cas d'applications réelles pouvant bénéficier des redistributions.

Portée : il est enfin suggéré d'ajouter une directive de portée *(scope)* afin d'utiliser de manière orthogonale les directives `independent`, `new` et `reduction`.

Ces quelques suggestions permettraient d'améliorer à peu de frais, nous semble-t-il, le langage, tant en ce qui concerne son utilité pour les applications, sa netteté en tant que langage de programmation que ses possibilités d'implantation efficace et simple.

1.2 Calcul des recouvrements à l'initialisation

Le chapitre 3 décrit une technique pour gérer le calcul des recouvrements (quel espace supplémentaire allouer pour stocker les éléments des processeurs voisins) dans un contexte de compilation séparée où l'on ne dispose pas de l'information nécessaire à la compilation. La technique présentée est naïve. Elle est simplement destinée à montrer la faisabilité de l'opération, et donc le caractère essentiellement inutile d'une extensions de langage pour requérir de l'utilisateur cette information comme il a été proposé. Cette motivation explique la présence dans la partie consacrée à la définition du langage de cette discussion dont le contenu porte plutôt sur une technique de compilation.

1.2.1 Le problème

La section 3.1 expose le problème. Dans un contexte de compilation séparée des routines, le compilateur ne sait pas pour un tableau local à la routine quelle sera la taille des recouvrements nécessaires, car elle peut dépendre des routines appelées. En conséquence, le code peut être conduit à faire des recopies coûteuses à l'exécution lorsqu'une routine découvre que les recouvrements d'un tableau qu'elle obtient en argument sont insuffisants. Il a donc été suggéré de permettre à l'utilisateur de déclarer explicitement au niveau des distributions les recouvrements nécessaires. Notre objectif est de montrer que ce recours est inutile.

1.2.2 La technique

La section 3.2 propose une solution qui évite de requérir cette information. Elle consiste simplement à en remettre le calcul à l'initialisation du programme. L'idée est qu'ainsi l'information est disponible pour l'allocation des données locales, que ce soit sur le tas ou dans la pile. Pratiquement, la technique génère une routine d'initialisation pour chaque routine compilée. La génération de cette routine ne suppose aucune information interprocédurale, si ce n'est la déclaration de l'interface des routines appelées. Par ailleurs, le compilateur doit générer du code qui fait l'hypothèse qu'il dispose de recouvrements de taille inconnue mais dont il sait qu'ils seront suffisants à l'exécution.

1.2.3 Exemple

La section 3.3 présente le code d'initialisation qui pourrait être généré pour le petit programme qui illustre la possibilité de déclarer des recouvrements explicitement. À l'initialisation du programme (*i.e.* au moment de son lancement), le calcul des recouvrements itère sur le graphe d'appels jusqu'à convergence au point fixe. Cet exemple est particulièrement simple, puisqu'il n'y a pas de récursion. Dans des cas réels, il est très probable que la difficulté soit la même, et donc que la résolution doit très rapide. Cependant notre technique s'accomode de récursions arbitraires.

1.2.4 Algorithme

La section 3.4 décrit rapidement l'algorithme permettant de générer le code d'initialisation pour une routine donnée. D'abord, les déclarations des différents descripteurs nécessaires sont faites. Ils sont locaux pour les données locales, exportés pour les arguments de la routine et importés pour les arguments des routines appelées. Ensuite, la routine d'initialisation est générée. Elle comporte une garde pour n'être appelée qu'une fois au plus par itération de la résolution à l'initialisation du programme des recouvrements nécessaires. La routine appelle les initialiseurs de ses appelées, puis calcule ses propres descripteurs locaux et exportés. Le graphe d'appel des routines reproduit celui du programme, et en matérialise donc la structure, ce qui permet à l'information de se propager.

1.3 Proposition de directive : `ASSUME`

Le chapitre 4 est le fruit d'un travail commun avec Henry ZONGARO [70]. Il s'agit de proposer une nouvelle directive. Cette directive vise à permettre la description de plusieurs placements pour un argument donné d'une routine, en lieu et place de la directive `inherit`. Une implantation possible et simple est également décrite, à base de clonage et de réécriture.

1.3.1 Motivation de la proposition

La section 4.1 donne les raisons qui justifient cette nouvelle directive. Les arguments d'une routine peuvent se voir affecter differents types de placements : pas de placement, un placement descriptif, prescriptif ou transcriptif (avec `inherit`). La compilation de ce dernier cas est a priori particulièrement inefficace, puisque aucune information n'est donnée au compilateur. Elle peut être correcte cependant si le compilateur extrait lui-même les informations nécessaires avec des analyses interprocédurales. Comme l'un des principes directeurs de la conception de HPF est de ne pas requérir de telles analyses de la part des compilateurs, mais de permettre au contraire une compilation séparée raisonnablement performante, d'autres solutions doivent être envisagées.

Lorsque l'utilisateur connaît un ensemble de placements possibles pour un argument, cette connaissance ne peut être exprimée. Soit l'inefficace `inherit` est alors utilisé, soit l'utilisateur duplique à la main sa routine pour compiler chaque instance avec plus d'information ... L'idée de cette proposition est de permettre de décrire plusieurs placements pour les arguments d'une routine.

1.3.2 Proposition formelle

La section 4.2 décrit la syntaxe de cette directive. Elle consiste simplement à séparer différentes descriptions par des marqueurs spéciaux qui séparent les alternatives. Une syntaxe simplifiée des marqueurs est aussi proposée pour les cas simples, qui permet de préciser une alternative à l'intérieur-même d'une directive.

Quelques contraintes régissent l'utilisation de cette directive. La première impose que la spécification apparaisse à l'identique dans l'interface de la routine, de manière à ce que l'appelant dispose également de l'information. La seconde interdit la multiplicité des placements aux données persistantes entre les appels (attribut `save`, données dans un `common`).

Cette proposition permet de réutiliser de manière directe toutes les directives HPF sans contrainte particulière. Le placement des données locales de la routine peut dépendre des différents placements des arguments. Enfin l'utilisation de plusieurs directives *assume* ou de directives *assume* imbriquées est possible.

1.3.3 Exemple

La section 4.3 présente un exemple d'utilisation de la directive *assume*. Le programme principal `assume` appelle une routine `iterate` dans différents contextes décrits à l'aide de la directive proposée.

1.3.4 Conseil d'implantation

La section 4.4 décrit une technique d'implantation au sein d'un compilateur, sans requérir de nouvelles techniques au-delà de l'état de l'art. La technique proposée permet de transformer un programme qui utilise la directive `ASSUME` avec les restrictions imposées en un programme HPF standard. L'idée est de dupliquer la routine en autant d'exemplaires que de placements décrits. Chaque exemplaire est compilé avec un jeu d'hypothèses différentes. Vue de l'appelant, la déclaration à l'identique des placements permet de choisir la meilleure version en fonction du placement courant des arguments. Les cas des données persistantes et des interactions possibles avec la directive `inherit` sont également discutés.

1.3.5 Avantages

La section 4.5 conclut en présentant les avantages de la directive. Celle-ci est facilement implantable sans avancée significative des techniques de compilation. Elle préserve toute la sémantique des placements autorisés par HPF, qui peuvent être réutilisés directement. Les données locales peuvent voir leur placement dépendre du placement des arguments de la routine. En conclusion, cette proposition devrait permettre de remplacer efficacement la plupart (voire toutes à notre avis) les instances de la directive `inherit`.

Le Forum n'a pas retenu cette proposition, au profit d'une nouvelle directive `RANGE` qui indique un ensemble de distributions, mais ne permet pas de préciser des alignements ou des arrangements de processeurs. Cette directive est beaucoup moins générale que notre proposition, et est paradoxalement plus difficile à implanter au sein d'un compilateur. En effet, elle ne se ramène pas à une déclaration statique, mais restreint l'ensemble des placements possibles d'un argument au placement hérité, sans pour autant le préciser. En particulier, le pas des alignements, la duplication partielle des tableaux ou encore la tailles des arrangements de processeurs ne peuvent pas être spécifiés alors que cette information est très utile à la compilation. Elle apporte donc une information de nature différente qui devra être intégrée et gérée spécifiquement par le compilateur. Un aspect négatif supplémentaire est que cette extension ne reprend pas le model de placement de HPF. Elle est donc inhomogène au reste du langage.

Chapitre 2

Discussing HPF Design Issues

Ce chapitre a fait l'objet d'une communication à Europar'96 [65].

Résumé

Alors que le High Performance Fortran (HPF) est à la fois développé et corrigé par le Forum HPF, il est important d'avoir des critères d'évaluation des caractéristiques techniques du langage. Cet article présente de tels critères liés à trois aspects : l'adéquation aux applications, l'esthétique et la pertinence dans un langage, et les possibilités d'implantation. Certaines caractéristiques actuelles de HPF, ou en cours d'étude, sont ensuite analysées selon ces critères. Leur mauvais équilibre nous permet de proposer de nouvelles spécifications pour résoudre les problèmes rencontrés : à savoir une directive de portée, des déclarations multiples de placement et des redistributions simplifiées.

Abstract

As High Performance Fortran (HPF) is being both developed and redesigned by the HPF Forum, it is important to provide comprehensive criteria for analyzing HPF features. This paper presents such criteria related to three aspects: adequacy to applications, aesthetic and soundness in a language, and implementability. Some features already in HPF or being currently discussed are analyzed according to these criteria. They are shown as not balanced. Thus new or improved features are suggested to solve the outlined deficiencies: namely a scope provider, multiple mapping declarations and simpler remappings.

2.1 Introduction

Analyzing and discussing the design of a programming language [231] involves balancing issues related to targeted applications and possible implementations within a compiler. The design must also address the technical and aesthetic quality of features. HPF [162] provides a data parallel programming model for distributed memory parallel machines. It was developed by an informal standardization forum after early commercial [237] and academic [99, 258] projects. Users advise the compiler about data mapping and parallelism through directives, constructs and intrinsic functions added on top of Fortran 90 [140]. The compiler is expected to switch from the global name space and implicit communication of HPF to the distributed local name space and explicit communication of the target parallel machine. This paper studies HPF design issues, after three years of existence and as commercial compilers have been released. The language is being both developed and redesigned by the HPF Forum: new features are considered for addition and some are selected for defining an efficiently implementable kernel. There is indeed an opportunity to improve HPF on the application, language and implementation aspects, as features are questioned at light of experience.

HPF version 1.1 [96] includes directives for describing regular mappings onto multiprocessor machines, parallel constructs, and intrinsics considered as useful or expressing some parallelism. Array mappings are statically declared through `template` and `processors` objects using `align` and `distribute`. Arrays with the `dynamic` attribute may be changed their mapping at runtime with `realign` and `redistribute` *executable* directives. Subroutine arguments may be specified a *descriptive*, *prescriptive* or *transcriptive* mapping: it may be warranted, to be enforced or unknown with `inherit`. Parallel loops are tagged with `independent`. `new` specifies variables private to iterations. `forall` is a data-parallel construct which generalizes Fortran 90 array expressions. Intrinsics are available to query at runtime about data mappings and to specify some reductions missing in Fortran 90.

These features form the core of HPF. After three years of existence, they are reconsidered at the light of early compiler implementations, application ports or teaching sessions. The Forum is redesigning the language through the kernel definition to allow efficient implementations. This paper first presents criteria to analyze HPF features (Section 2.2). Some current and planed features are then analyzed with respect to these criteria (Section 2.3). They are shown as not well balanced. This is the ground for new or improved HPF features which are suggested (Section 2.4) before concluding.

2.2 Analysis criteria

It seems natural to analyze language features with respect to the application, language and implementation aspects which are respectively the concern of users, designers and implementors. For each point, several issues are discussed, which express the interests each group might find in a feature.

2.2.1 Application

The preliminary goal of HPF is to suit data-parallel applications to be run efficiently on distributed memory parallel machines. HPF feature benefits for users can be seen through three perspectives.

Useful: some applications require or benefit from a feature for efficiency. Some unconstrained features such as remappings are not useful in their full generality and are difficult to handle for the compiler.

Expressive: HPF aims at providing to the user a mean to express its knowledge about the application in order to help the compiler. If some knowledge cannot be expressed while it would be useful, the language lacks something.

```
!hpf$ independent
      do k=1, n
!hpf$   independent
        do j=1, n
!hpf$     new
          s = 0.
!hpf$     independent
          do i=1, n
!hpf$       new
            t = b(i,j,k)+c(i,j,k)
            a(i,j,k) = t*t+t
!hpf$       reduction
            s = s + t
!hpf$       reduction
            p = p * t
          enddo
!hpf$     reduction
          p = p * s
        enddo
      enddo
```

```
!hpf$ independent(k,j), &
!hpf$   new(s), reduction(p)
      do k=1, n
        do j=1, n
          s = 0.
!hpf$     independent(i), &
!hpf$       new(t), reduction(s)
          do i=1, n
            t = b(i,j,k)+c(i,j,k)
            a(i,j,k) = t*t+t
            s = s + t
            p = p * t
          enddo
          p = p * s
        enddo
      enddo
```

Figure 2.1: Internal and external directive styles

Elegant: notations must be concise, intuitive and as simple as possible. Thus they are easy to teach, learn and remember. Also users may misunderstood some features which are error-prone.

2.2.2 Language

From the designer point of view, language features must be neat, and have some technical properties which are considered as qualities, or their absence are considered as design mistakes and source of difficulties for users. It is

Homogeneous if different features share a common look and feel. This issue is illustrated in Figure 2.1, where two possible homogeneous directive styles are presented. The first one applies on the next statement and is verbose, while the other is external and applies on the explicitly specified variables in the next loop.

Orthogonal if features are not constrained one with respect to the other. HPF is built orthogonaly (with very few restrictions) on top of Fortran 90: it is up to the compiler to insure that HPF and Fortran features interact as expected. Also for instance Fortran 90 array expressions cannot specify a transposition, because dimensions are taken in the same order. HPF `forall` provides the array expression and dimension orthogonality.

Complete if all cases are expressed. The `inherit` mapping addresses this point (see Section 2.3). While this is a nice property for a language, handling any case with no motivation for efficiency can have a significant impact on the compiler costs, if not on the compiler structure.

```
      subroutine callee(A)
      real A(n)
!hpf$ inherit A
      ! A mapping is not known!
       A = ...
```

Figure 2.2: Inherit unknowledge

```
!hpf$ distribute A(block)
      if (...) then
!hpf$   redistribute A(cyclic)
      endif
      ! A is block OR cyclic
      A = ...
```

Figure 2.3: Remapping unknowledge

2.2.3 Compiler

The last but not least issue is the implementability at a reasonable cost in a compiler. The state of the art evolves continuously because of active research, but general conditions for sound and simple implementations remain. As the market for the language is not very large development costs due to useless features must be avoided. For the implementor a feature must be

Simple: features are added with *some* possible use in mind, but implementors must handle them in *all* cases, including the worst; it may also happen that some future optimizations are prevented or becomes more difficult because of complex features; this issues are illustrated below with remappings.

Efficient: to allow compilation to produce efficient code, as much static information as possible must be provided, and compile-time handling must not be prevented; see the `assume` directive suggestion in Section 2.4.1.

Separable: as interprocedural compiler technology [138] is not yet the common lot of software providers, features must not prevent separate compilation of subroutines; Fortran 90 explicit subroutine interfaces provide the mean to allow both maximum information about callees and separate compilation, if they are both mandatory and extended.

2.3 HPF feature analyses

Let us now analyze some HPF features according to these criteria. They are shown either as not simply and efficiently implementable or miss some useful instances. The difficulty of the design is to balance criteria for a given feature, in order to reach a sound proposal.

Inherit was added for completion purposes when specifying mappings of subroutine arguments. Some passed arguments cannot be described naturally with descriptive mappings, thus the need to be able to specify something else. Compilation of this feature is tricky because the compiler cannot assume any knowledge about the mapping (Figure 2.2). Applications in which different mapping may reach a given subroutine tend to use this non-description while some knowledge could be available. Section 2.4.1 discusses the `assume` directive which can provide more knowledge to the compiler.

Remappings are well integrated in HPF, but not in compilers: they are definitely useful to application kernels such as ADI [151], FFT [118] and linear algebra [209]; the syntax is homogeneous to static mapping directives;

remappings are orthogonal to the program control flow. This very orthogonality is the nightmare of implementors (if any): remappings may appear anywhere without constraints in the program, and may depend on runtime values. Thus for a given array reference the compiler may not know the mapping status of an array, as show in Figure 2.3. This ambiguity similar to `inherit` has a strong impact on the way references can be handled, delaying everything to runtime, and preventing standard analyses and optimizations from giving their full profit. This worst case must be handled before optimizing for programs where there is no such ambiguity. Section 2.4.2 discusses possible improvements in the design.

Forall was added for orthogonality of dimensions in array expressions. It is as implementable as array expressions. Users often feel the `forall` as *the* parallel loop of HPF, because it looks like a loop and is parallel. Moreover early versions of HPF compilers did not include the `independent` directive, thus it was the only mean to specify some parallelism. Most of the time independent foralls are not tagged as such by users. However by doing so unnecessary synchronizations and copies may be performed at runtime to warrant the SIMD execution order of the loop. The feature is somehow misunderstood and misused.

Internal style reductions (Figure 2.1) were first considered by designers [223], and then changed [66]: although equivalent to the external syntax, this choice was not homogeneous with the `new` directive, which also deals with scalars. Moreover the external syntax eases the implementation: in the example the reduction on `p` is over the full loop nest while the one on `s` is only at the `i` level. This level is directly specified with the external syntax, while it must be extracted upward in the code checking for some condition with the internal syntax, requiring more work from the compiler.

Independent, New and Reduction are definitely not well integrated in the language: the directive styles are not homogeneous (Figure 2.1): it is internal for `independent` and external for others. Moreover these directives are not orthogonal: `new` and `reduction` must be attached to an `independent`. However some non independent loops could benefit from these directives, as well as pieces of flat code. Section 2.4.3 develops an homogeneous and orthogonal proposal. This proposal does not change implementability issues, but allows more information to be given to the compiler.

2.4 Improved and additional features

In the previous analysis, several deficiencies of HPF features were outlined: lacks of simple implementability, language homogeneity and orthogonality or expressiveness for some applications. In the following, several new or improved features are suggested to solve these problems.

```
assume-directive =
   ASSUME
     spec-directive-or-list
   END [ASSUME]

spec-directive-or-list =
   spec-directive-list
 [ OR
   spec-directive-or-list ]

spec-directive-list =
   <nothing>
 | spec-directive
 [ spec-directive-list ]
```

Figure 2.4: ASSUME directive proposal

```
attached-directive =
   explicit-independent-directive
 | new-directive
 | reduction-directive
 | ...

explicit-independent-directive =
   INDEPENDENT(...)

scope-directive =
   BLOCK
   statements-and-directive-list
   END [BLOCK]
```

Figure 2.5: Scope for orthogonal directives

```
      subroutine callee(A(100))
!hpf$ ASSUME
!hpf$   distribute A(block)
!hpf$ OR
!hpf$   distribute A(cyclic)
!hpf$ OR
!hpf$   ! nothing, not mapped
!hpf$ END ASSUME
```

Figure 2.6: Assume example

```
!hpf$ DISTRIBUTE A(block,block)
      if (...) then
!hpf$   REDISTRIBUTE A(*, block)
          ...
      endif
! A is (*,block) or (block,*)
! thus must *not* be referenced
!hpf$ REDISTRIBUTE A(*,block)
```

Figure 2.7: Remapping example

2.4.1 Assume

HPF does not allow to describe several mappings that may reach subroutine arguments. Thus either `inherit` is used, causing inefficiency, and/or subroutines are cloned by hand to build several versions to be compiled with more knowledge. There is a tradeoff between efficiency and software quality. The `assume` directive [62, 70] outlined in Figure 2.4 allows to describe several mapping for arguments in the declaration part of the subroutine. Figure 2.6 shows a simple example. It asserts that array `A` may reach the subroutine with a block or cyclic distribution, or no distribution at all (*i.e.* a local array).

It is up to the compiler to manage the cloning and selection of the right version. Some carefully designed constraints are necessary for allowing both simple and efficient implementations: (1) the `assume` directive must also appear in the explicit subroutine interface; (2) the subroutine argument mappings must appear in the same order in both declarations; These two points all together allow separate compilation and compile/link-time handling of the feature.

The expressiveness is maximum, because no constraints are needed on the mapping directives involved: descriptive and prescriptive mappings can be specified for the arguments, as well as on local variables, including partially redundant or useless specifications. Thus no special checking is required. Several and nested `assume` directives allows to describe the cross product of possible mappings. Constraints (1) and (2) allow callers and callees to agree about naming conventions for efficient and separate compile and link time implementation: when compiling a call, the list of assumed mappings is scanned for the best match and this version is called. Its name should depend on the version number, which is the same in both caller and callee declarations. From these constraints, a program with an assume specification can be translated by cloning into a standard HPF program as described in [70].

2.4.2 Remappings

As shown above remappings are perfectly integrated in the language and suited to some applications. It is perfect, but the implementation. Thus the current design of the HPF kernel excludes all remappings. We argue [62] that some could be kept, and especially those that are the most useful. This suggestion is supported by a full implementation in our prototype HPF compiler [68].

The following constraints are suggested: (1) all remappings must be statically known (*i.e.* as constrained as static mapping declarations); (2) array references should have at most one reaching mapping. These constraints disallow all ambiguities at the price of orthogonality: from (1) the mapping cannot depend on runtime values, and (2) implies that remappings cannot appear anywhere. Checking for (2) is similar to the the reaching definition standard data flow problem [1]. This second restriction does not apply to remapping statements: there may be some points in the program where the mapping of an array is ambiguous, provided that the array is not referenced at this point. This is illustrated in Figure 2.7: between the `endif` and up to the last `redistribute` the array mapping is ambiguous to the compiler, thus the array must not be referenced. In such cases the runtime must keep track of the current status of an array in order to perform the right remapping. Remappings are much easier to compile under these constraints.

These restrictions do not prevent any use of remappings in the real application kernels cited in Section 2.3. Indeed, it is very hard to imagine a real application in which the desired mapping of some references would be unknown to the user. As computations are performed by the application, the user has a clear idea of the right mapping at different point and the remapping directives naturally reflect this information to the compiler without ambiguity. The pathological cases that are the nightmare of implementors are not those that would appear in any real application. Thus the suggested constraints, while allowing a much simpler implementation, does not prevent the use of remappings for efficiency.

2.4.3 Scope

As noted above, the `independent`, `new` and `reduction` directives are not homogeneous and orthogonal. Let us first allow for homogeneity the external syntax for `independent`. Figure 2.8 shows a loop example where the non-orthogonality prevents the assertion of useful information: if iterations are performed across processors in the first example, `new` advises the compiler not to transmit and update the `time` variable between all processors, and `reduction` does the same for `s`, avoiding useless communication. These directives may also be useful outside of loops, as shown in Figure 2.9 (with the syntax suggested hereafter): stencil computations typically use particular formula for corners that use temporary scalars

```
!hpf$ new(time), &
!hpf$   reduction(s)
      do i=1, n
        s = s + ...
        time = i*delta
        ! some use of time
      enddo
      ! no use (read) of time
```

Figure 2.8: Orthogonality of directives

```
!hpf$ new(t), reduction(s)
!hpf$ block
      t = a(1,1)+a(2,1)
      a(1,2) = t*t
      s = s + t ! out of a loop
      do i=1, n
        s = s + ...  ! in
        ! some computation
      enddo
!hpf$ end block
```

Figure 2.9: `new` and `reduction` scoping

which can be private to the processor holding the corner.

The scope proposal in Figure 2.5 provides a solution to these problems. Directives are orthogonal one to the other, and apply on the next statement scope, which may be a loop or a scope-provider directive in the spirit of what is suggested for the `on` clause [165]. The `block/end block` provides scoping for flat regions of code. It would be obsoleted by the addition of such a structure to Fortran.

For consistency and homogeneity, these directives must be active for the next statement scope. Thus `independent(i)` implies that all loops with index `i` in the scope are parallel. Parallel loop indexes are necessarily and implicitly private. Moreover `new` and `reduction` do not make sense on `forall`, because no scalar can be defined within this construct. Scopes may be nested: in Figure 2.1 scalar `s` is private to the full loop nest but is protected as a reduction in the inner loop. This proposal gives more homogeneity and orthogonality to HPF. Implementation issues are not changed: more information than before is available to the compiler, that can only benefit to applications.

2.5 Conclusion

HPF design has been discussed with respect to application, language and implementation issues. Several features have been analyzed and deficiencies outlined. Such an analysis is obviously much easier after experience. Then new features were suggested to improve the language with respect to our criteria. An `assume` directive, a new scope provider and simpler remappings were described. Possible simple and efficient implementation techniques were presented.

Other HPF features could also be discussed: replication of data through `align` seems tricky for users. The simpler and intuitive syntax suggested in Fortran D and Vienna Fortran could allow this specification in `distribute` in most cases. The `on` directive currently discussed is in the internal style spirit and includes a private scope provider. This does not improve HPF homogeneity and orthogonality.

Discussing design issues is a difficult task. Features are developed and discussed independently one of the other, thus homogeneity issues and possible interactions are easily overlooked in the difficult process of writing a sound and correct specification. Possible alternatives to design decisions are not obvious, they require imagination. Although a Forum may be able to select good ideas once expressed and formalized, such ideas are not always found. Some problems come only to light after experience. Teaching, implementing and using the language helps to clarify many points, to identify new problems and to offer new solutions.

Acknowledgment

Thanks to: Charles Koelbel, Robert Schreiber, Henk Sips and Henry Zongaro for electronic discussions about these issues; Corinne Ancourt, Denis Barthou, Béatrice Creusillet, Ronan Keryell and the referees for comments and corrections.

Chapitre 3

Shadow Width Computation at Initialization

L'objet principal de ce chapitre est de montrer qu'une extension de langage pour déclarer explicitement les recouvrements requis ou utiles à un tableau n'est pas nécessaire, même avec une contrainte de compilation séparée. Il est disponible sous forme de rapport interne EMP-CRI A 285 [67].

Résumé

Une technique standard pour compiler les programmes à base de *stencils* sur les machines à mémoire répartie est d'allouer une bordure pour stocker les copies des éléments appartenant aux processeurs voisins. La gestion de ces bords au niveau des appels de procédures par les compilateurs HPF n'est pas évidente car les appelants ne connaissent pas les besoins des appelés. Il a donc été proposé de permettre leur déclaration explicite. Nous présentons une technique pour calculer cette information à l'initialisation du programme. Elle peut alors être utilisée pour allouer de la mémoire sur la pile ou dans le tas. Cette technique permet la compilation séparée des programmes qu'il y ait ou non récursion. Le calcul de la taille des bordures est simplement effectué au moment de l'initialisation du programme et est basé sur des conventions simples à la compilation.

Abstract

To compile stencil-like computations for distributed memory machines a standard technique is to allocate shadow edges to store remote elements from neighbor processors [106]. The efficient management of these edges by HPF compilers at subroutine boundaries is not obvious because the callers do not know the callees' requirements. It was proposed to allow their explicit declaration [196]. We present a technique to make this information automatically available on entry in the program, ready to be used for heap and stack allocated data. It suits separate compilation of programs involving arbitrary recursions. The shadow width computation is simply delayed at init-time and is based on simple compile-time conventions.

3.1 Problem

Extending the local declarations of distributed arrays so as to store remote elements is a standard technique for managing temporary data allocation of stencil-like computations. The management of these overlaps at subroutine boundaries is not obvious for non interpocedural compilers because of the following issues:

- the callees do not know about incoming array available overlaps.
- the callers do not know about the callees' requirements for overlaps.

The first problem is not a big issue. Indeed, the available overlap can be passed from the caller to the callee through conventional descriptors, and the callee program can be compiled to managed unknow but sufficient overlap declarations. However, if the overlap declaration is not sufficient, the callee must copy the parameter into a larger array in order to match the required overlap. It is desirable to avoid such useless runtime copies. In ADAPTOR [42], all arrays are heap allocated. When larger overlaps are required, the array is dynamically updated, and will remain with this larger allocation until its death. This limits the number of runtime copies required. However, they can be avoided.

3.2 Technique

We present a technique to make the callees' requirements available at runtime to the caller. This technique makes the following assumptions:

- explicit declaration interfaces about the callees are available to the caller.
- this interface describes the mapping of dummy arguments.
- the dummy arguments are passed by assumed shape.
- a subroutine can be compiled with a *unknown but sufficient* overlap for dummy arguments.
- the call graph may be arbitrary.
- data that may require overlap are heap or stack allocated (but the size and initial offsets should not be compile-time decided as it may be the case with *commons* for instance).
- local overlaps required for each subroutine are compile-time constants.

Under these assumptions, simple compile time conventions allows to generate some initialization functions which will compute the overlaps at init-time. When the program is launched, these intialization functions are called to instantiate the needed overlaps for every array along the call graph. Arbitrary recursion requires a fixed point computation, thus these initialization functions must be called till stability. They mimic the program call graph and propagate the required overlaps from callees to callers. The technique is the following:

- compile-time
  - compile each routine with an *unkown but sufficient* overlap assumption about the dummy arguments, as it is already done for array arguments the sizes of the dimension of which are unknown at compile-time.

  - global shadow width descriptors are created for the subroutine dummy arguments.
  - local shadow width descriptors are created for its local arrays.
  - an initialization function is created for these descriptors.
    * it calls the subroutine callees' associated initialization functions.
    * it uses its callees' exported descriptors to compute and update its own local and exported descriptors.
    * if exported descriptors are modified, then the computation is not yet stable and will have to be recomputed again.
    * care must be taken to manage recursion, that is not to run several time the same initializer for a given iteration.

- init-time
  - the main's initializer is called, and it will propagates the initialization to the whole callgraph.
  - the process is iterated till stability.
  - the convergence of the iterative process is insured because:
    * each computed overlap is always increasing at each iteration, as a maximum of other increasing overlaps required by callees.
    * the overlaps are bounded by the largest overlap required over the whole program, which is a compile-time constant.

The idea of the technique is to propagate the possible actual to dummy argument bindings so as to compute the overlaps of all local variables, which will be available and large enough for all possible argument passing through the callgraph. The actual possible implementation described hereafter is obviously naive and not very efficient, and is just intended at showing the feasability of such a computation at init-time and with separate compilation.

3.3 Example

For the example in [196] shown in Figure 3.1 we would have (`$` is used as a special character to avoid any interaction with user's global identifiers):

- global initializer, to be called by `__main`.

```
int $_not_stable;
void $_initialize()
{
  int iter = 1;
  do {
    $_not_stable = 0;
    $_initialize_main(iter++);
  } while ($_not_stable);
}
```

```
      program main

      real, dimension(500,500):: A
!hpf$ distribute (block,block):: A

      interface
         subroutine sub(D)
            real, dimension(500,500):: D
!hpf$       distribute (block,block):: D
         end subroutine
      end interface

      ...

      do i=1, 1000
         call sub(A)
      end do

      ...

      end program


      subroutine sub(D)

      real, dimension(500,500):: D
!hpf$ distribute (block,block):: D

      forall(i=1:500,j=1:500)
         d(i,j) = 0.25* (d(i+1,j)+d(i-1,j)+d(i,j+1)+d(i,j-1))
      end forall

      end subroutine
```

Figure 3.1: Simple code with hidden overlaps

- program `main` initializer:

```
static int $_main_a_1_lo; // LOwer overlap for A of MAIN, dim=1
static int $_main_a_1_up;
extern int $_sub_1_1_lo; // LOwer overlap of arg number 1 of SUB, dim=1
extern int $_sub_1_1_up;
void $_initialize_main(int iter)
{
  static int done = 0; // last iteration computed
  if (iter==done) return; // once per iteration!
  else done=iter;

  $_initialize_sub(iter); // CALLEES

  $_main_a_1_lo = $_sub_1_1_lo; // LOCALS
  $_main_a_1_up = $_sub_1_1_up; // max of requirements...
  // NO GLOBALS
}
```

- subroutine `sub` initializer:

```
int $_sub_1_1_lo = 0; // LOwer overlap for arg number 1 of SUB, dim=1
int $_sub_1_1_up = 0;
extern int $_not_stable;
void $_initialize_sub(int iter)
{
  int tmp;
  static int done = 0;
  if (iter==done) return; else done=iter;

  // NO CALLEES
  // NO LOCALS
  // GLOBAL DUMMY ARGUMENTS:
  tmp = 1;
  if (tmp!=$_sub_1_1_lo) {
    $_sub_1_1_lo = tmp;
    $_not_stable = 1;
  }

  tmp = 1;
  if (tmp!=$_sub_1_1_up) {
    $_sub_1_1_up = tmp;
    $_not_stable = 1;
  }
}
```

Note that a similar template is generated for every routine (here `main` and `sub`). The initialization routines do not require any interprocedural assumptions but the interface declarations which allow to reference the callees exported overlap descriptors. The iterative process is necessary to solve what is a fixed point computation over the callgraph, and the initial guard allows to manage arbitrary recursion and enforces that each initialization is run only once per iteration.

3.4 Algorithm

Here is the algorithm for the compiler generation of the required initialization functions, using the C language. When compiling routine r:

- for each distributed argument number i and dimension d that may be distributed and have an overlap generate the exported descriptors:

  `int $_`r`_`i`_`d`_lo = 0, $_`r`_`i`_`d`_up = 0;`

- for each argument number i and dimension d of callee c of r that may be distributed and require an overlap generates the imported descriptors:

  `extern int $_`c`_`i`_`d`_lo, $_`c`_`i`_`d`_up;`

- for each dimension d of local array of r named a that may be distributed and require an overlap generates the local descriptors:

  `static int $_`r`_`a`_`d`_lo, $_`r`_`a`_`d`_up;`

- generates the stability checker:

  `extern int $_not_stable;`

- generates the initialization function:

  - generates the prolog of the function:

    ```
    void $_initialize_r(int iter)
    { int tmp; static int done = 0;
     if (iter==done) return; else done=iter;
    ```

  - for each callee c of r generates a call to its initializer:

    ```
     $_initialize_c(iter);
    ```

  - initialize each local array descriptor `$_`r`_`a`_`d`_`s (where a is the array name, d the dimension number, s is `lo` or `up`) as the maximum of the local overlap required l and the callee arguments a may be bound to in routine r.
  - initialize each global array descriptor `$_`r`_`i`_`d`_`s as above, and update the `$_not_stable` variable if it is modified:

    ```
     tmp = ...
     if (tmp!=$_r_i_d_s) {
     $_r_i_d_s = tmp, $_not_stable = 1; }
    ```

  - generates the initialization function postlog: `}`

- the following routine must be called before main for computing the overlap at init-time:

```
int $_not_stable;
void $_initialize()
{
  int iter = 1;
  do {
    $_not_stable = 0;
```

```
    $_initialize_main(iter++);
  } while ($_not_stable);
}
```

- stack or heap allocation of array a of r at runtime must use the overlap computed for each dimension, that is $\$\_r\_a\_d\_s$.
- the caller must pass to the callees the available overlaps, which are insured to be greater or equal what is required.

3.5 Conclusion

We have presented a technique to compute at init-time the overlap required by callees, so as to avoid useless copies at runtime. It simply delays the interprocedural computation of this information when an interprocedural view of the program is possible in a separate compilation environment, that is at runtime! The technique suits heap and stack allocated data. It can be extended for:

- managing init-time known required overlaps (the required overlap was assumed to be a compile-time constant);
- managing dynamically allocated and pointed array overlaps, by computing at compile-time conservative assumptions about the possible aliasing of pointers and actual data.

Such a technique can also be used in the context of optimized library calls, if the initialization functions and argument descriptors are made available by these libraries, following the same conventions.

Chapitre 4

ASSUME Directive Proposal

Fabien Coelho et Henry Zongaro

Ce chapitre présente la spécification de la directive **ASSUME** *proposée au Forum HPF, et refusée. Il fait l'objet du rapport interne EMP CRI A-287 [70].*

Résumé

Cette proposition d'extension pour HPF permet la déclaration de plusieurs placements au lieu d'un seul aux frontières des sous-routines. Des marqueurs au niveau des déclarations séparent différentes descriptions de placement données au compilateur pour les arguments des sous-routines. Toute la richesse des directives HPF de placement peut être utilisée. Une technique de réécriture est aussi présentée qui permet de traduire un tel programme en un programme HPF standard.

Abstract

This proposal allows the declaration of several mappings instead of just one at subroutine boundaries. Markers within subroutine declarations separate different argument mapping assumptions provided to the compiler. Full HPF specification directives can be expressed. A rewriting-based implementation technique is also suggested to translate such a program into a standard HPF program.

4.1 Motivation

Subroutine dummy arguments can be given *descriptive*, *prescriptive* or *transcriptive* (`inherit`) mappings through HPF specification directives, or no mapping at all. While descriptive and prescriptive are thought as efficiently manageable, transcriptive can lead to very poor performances because no static knowledge is provided to the compiler.

If several actual argument mappings can reach a subroutine, such information cannot be expressed to the compiler. Thus, in order to compile the subroutine with some knowledge, the compiler must extract some information from the program, what can be done only through interprocedural analyses by inspecting call sites for the routine. However, it has been a design principle during the specification of HPF that no interprocedural compilation technique should be mandatory for compiling HPF and producing reasonnably efficient codes.

An alternative to the poor performances of `inherit` in the absence of interprocedural analyses is hand cloning: the routine is duplicated and different mappings are associated to the arguments. each routine is then compiled with some direct descriptive or prescriptive mappings, and the right version must be called, maybe through some runtime switching. This leads to poor software engineering practices. Such simple issues as cloning a routine for different argument mappings and substituting the right version to call in its callers is definitely the job of the compiler.

This proposal aims at allowing the specification of several mappings that may reach a routine, with minimum addition to the HPF mapping semantics. Another issue is to preserve separate compilation. A very important point which drives this proposal is its implementability and efficiency with the current compiler technology, while preserving maximum flexibility and expressiveness for users. A possible implementation described as an advice to the implementors is to actually clone the routines following the user prescriptions and substitute the right version in the callers.

4.2 Proposal

The idea is to specify a set of mappings instead of one mapping for subroutine dummy arguments. Each subset should be consistent on its own; that is the specification should make sense for the compiler. It can be seen as some kind of interface to the caller in a callee, as there are interfaces about callees in callers.

4.2.1 Full syntax

The syntax of the assume directive consists of markers to distinguish between different standard HPF specification directive sets to be taken into account by the compiler. Each line with the assume directive scope should appear after a *directive-origin* or be a comment. The syntax is the following:

assume-directive **is**
    `ASSUME`
    list-of-or-separated-specification-directives
    `END ASSUME`

list-of-or-separated-specification-directives **is**
    list-of-specification-directives
    [`OR`
    list-of-or-separated-specification-directives]

list-of-specification-directives **is**
    [*specification-directive*

[*list-of-specification-directives*]]

Constraint 1 *The assume-directive may appear within the declarations of the subroutine. If so, it* must *also appear in the subroutine explicit interface, the mapping specifications related to dummy arguments being in the very same order. (The rationnale for this is to allow both caller and callee to aggree about naming conventions if a simple rewriting-based technique is used for implementing the feature).*

Constraint 2 *A variable with the `SAVE` attribute or which is a member of a common must not be specified several mappings through the assume-directive (thus these variables may appear in an assume-directive, provided that they are mapped the very same way in each set of specifications, what means that their mapping specifications could be factored out of the assume-directive).*

Each *list-of-specification-directives* in the *list-of-or-separated-specification-directives* specify a distinct set of assumptions about mappings that may reach the subroutine arguments. The compiler can take advantage of the provided information as described in Section 4.4.

4.2.2 Shorthand syntax

A shorthand syntax is also suggested to handle simple cases. It applies to any *specification-directive*. In such a directive, the `[`, `|` and `]` markers specify directly within the directive different assumptions, as depicted in Figure 4.1. This syntax is not as expressive as the full syntax specified above.

```
begin [ one | two | three ] end
```

would be strictly equivalent to:

```
ASSUME
  begin one end
OR
  begin two end
OR
  begin three end
END ASSUME
```

Figure 4.1: `[` and `]` shorthands for assume

Here are some examples of shorthand syntax specifications:

```
!hpf$ template T(100)
!hpf$ align with T( [ i | * ] ) :: A(i)
!hpf$ processors P [ (4) | (8) ]
!hpf$ distribute ( [ block | cyclic ] ) onto P :: T
```

Note 1 *The suggested shorthand syntax expressiveness is quite restricted compared to the full syntax. Also it implies implicitely the cross product of specifications if several are used.*

- `[`, `|` and `]` characters might cause interaction problems for some national character sets. We might think of `{` and `}` in their place? Or of trigraphs, such as `??(` and `??)`?

- Shorthand syntax issue: It seems clear that no context free grammar should be looked for for handling such a construct. The only convenient approach rather seems to have a mandatory preprocessing of the directive lines at the lexical level that would look for the markers and rewrite the code to the full syntax.

4.2.3 Remarks

Note 2 *The list-of-specification-directives may be empty, specifying that the arguments are not explicitely mapped.*

Note 3 *The list-of-specification-directives may include directives on variables local to the subroutine.*

Note 4 *Arbitrary mappings (*descriptive, prescriptive *or even* transcriptive*) can be specified within the assume scope, including not only alignments, distributions, but also processors and template declarations.*

Note 5 *There may be other specification-directives outside the scope of the assume-directive.*

Note 6 *There may be several assume-directives within the declarations of the subroutine, specifying the cross product of the specifications. For instance both specifications in Figure 4.2 are equivalent.*

Note 7 *assume-directives may be nested, specifying the cross product of the specifications. For instance both specifications in Figure 4.3 are equivalent.*

4.3 Example

Let us consider the example in Figure 4.4. Program `assume` calls subroutine `iterate` in four different contexts. The assume directives allows the caller and callee to agree upon the possible mappings of arguments. Maximum static knowledge is provided to the compiler, for both handling the caller and the callee.

- program `assume`
  - four local arrays are declared, of which three are distributed.
  - subroutine `iterate` interface is declared, which uses a nested assume directive; it includes a `processors` declaration.
  - three possible reaching mappings are specified, of which two are prescriptive and the last one is empty; thus it specifies three calling contexts: block, cyclic and not distributed.
  - The first three calls to `iterate` correspond to the three contexts: no remapping is required at runtime.
  - For the last call, the compiler must chose which *version* to call. Both prescriptive mappings are candidate. The compiler may chose the first one, remapping array `D` to block, or apply any criterion, for instance the expected cost due to the remapping.

```
ASSUME
  d1
OR
  d2
END ASSUME
ASSUME
  d3
OR
  d4
END ASSUME
```

```
ASSUME
  d1
  d3
OR
  d1
  d4
OR
  d2
  d3
OR
  d2
  d4
END ASSUME
```

Figure 4.2: Several *assume-directive*s

```
ASSUME
  ASSUME
    d1
  OR
    d2
  END ASSUME
  d3
END ASSUME
```

```
ASSUME
  d1
  d3
OR
  d2
  d3
END ASSUME
```

Figure 4.3: Nested *assume-directive*s

- subroutine `iterate`
  - one local array (`TMP`) is declared.
  - a saved variables is also declared (`count`)
  - the assume directive specifies the mapping of the local array and dummy argument.
  - the same three reaching mappings are specified.
  - it would not make much sense to have the `TMP` to `X` alignment or the `processors` declaration outside of the assume because there are no distribution for the third case.

4.4 Advice to implementors

In this section we describe a possible implementation based on cloning. Such an implementation technique does not put new requirements on the compiler technology: the cloning procedure suggested hereafter is a rewriting technique applied on an assume specification to match a standard HPF program, thus can be applied with the current state of the art in compilation. The contraints specified on the assume specification allow this technique to be applied in a separate compilation context. This technique is only illustrative.

Compilation of the callee

The callee is compiled by cloning the subroutine for each specified mapping set. For the example in Figure 4.4 it would lead to three subroutines. Some naming convention must be chosen so as to differentiate the different versions. Let us assume it consists of appending $\_i$ to the subroutine name, where i is the instance number of the assumption set. This would lead to the three versions depicted in Figure 4.5. These three subroutines are standard HPF subroutines.

Handling of the `SAVE` attribute

In order to stick to the `SAVE` semantics the cloned versions must share the same address space for these data. The rewriting process can put them in some special `COMMON` (or `MODULE`) especially created for sharing data between cloned versions. Constraint 2 allows such a special area to be created as depicted in Figure 4.5 for saved variable `count`.

Compilation of the caller

From the caller point of view, the very same cloning must be applied on the assume part specified in the interface. The very same naming convention can be applied, because due to Constraint 1 the interface is in the same order as in the callee declaration. The caller must also switch the call sites to the right version, depending on the mapping of the array. This leads to the standard HPF program depicted in Figure 4.6.

```
  program assume
    real, dimension (100) :: A, B, C, D
!hpf$ processors P(10)
!hpf$ distribute onto P :: A(block), B(cyclic), D(cyclic(5))
    interface
      subroutine iterate(X)
        real, dimension(100) :: X
!hpf$   ASSUME
!hpf$     processors Px(10)
!hpf$     ASSUME
!hpf$       distribute X(block) onto Px
!hpf$     OR
!hpf$       distribute X(cyclic) onto Px
!hpf$     END ASSUME
!hpf$   OR
          ! X not distributed!
!hpf$   END ASSUME
      end subroutine
    end interface
    call iterate(A) ! block
    call iterate(B) ! cyclic
    call iterate(C) ! not distributed
    call iterate(D) ! compiler chosen version (=> remapping)
  end program

  subroutine iterate(X)
    real, dimension(100) :: X, TMP
!hpf$ ASSUME
!hpf$   align with X :: TMP
!hpf$   processors P(10)
!hpf$   ASSUME
!hpf$     distribute X(block) onto P
!hpf$   OR
!hpf$     distribute X(cyclic) onto P
!hpf$   END ASSUME
!hpf$ OR
        ! X not distributed!
!hpf$ END ASSUME
    integer, save :: count
    count = count + 1
    TMP = X*X ! some computation...
    X = 1 + X + TMP + TMP*TMP
  end subroutine
```

Figure 4.4: Example with assume

```
  subroutine iterate_1(X)
    real, dimension(100) :: X, TMP
!hpf$ align with X :: TMP
!hpf$ processors P(10)
!hpf$ distribute X (block) onto P
    common /iterate_save/ count
    integer count
    count = count + 1
    TMP = X*X ! some computation...
    X = 1 + X + TMP + TMP*TMP
  end subroutine

  subroutine iterate_2(X)
    real, dimension(100) :: X, TMP
!hpf$ align with X :: TMP
!hpf$ processors P(10)
!hpf$ distribute X (cyclic) onto P
    common /iterate_save/ count
    integer count
    count = count + 1
    TMP = X*X ! some computation...
    X = 1 + X + TMP + TMP*TMP
  end subroutine

  subroutine iterate_3(X)
    real, dimension(100) :: X, TMP
    common /iterate_save/ count
    integer count
    count = count + 1
    TMP = X*X ! some computation...
    X = 1 + X + TMP + TMP*TMP
  end subroutine
```

Figure 4.5: Cloned callee versions for example in Figure 4.4

```
  program assume
    real, dimension (100) :: A, B, C, D
!hpf$ processors P(10)
!hpf$ distribute onto P :: A(block), B(cyclic), D(cyclic(5))
    interface
      subroutine iterate_1(X)
        real, dimension(100) :: X
!hpf$   processors Px(10)
!hpf$   distribute X (block) onto Px
      end subroutine
      subroutine iterate_2(X)
        real, dimension(100) :: X
!hpf$   processors Px(10)
!hpf$   distribute X (cyclic) onto Px
      end subroutine
      subroutine iterate_3(X)
        real, dimension(100) :: X
      end subroutine
    end interface
    call iterate_1(A)
    call iterate_2(B)
    call iterate_3(C)
    call iterate_1(D) ! compiler chosen version (=> remapping)
  end program
```

Figure 4.6: Caller version after rewriting

Handling of INHERIT

It may happen that the actual argument mapping of an array is unknown from the caller, due to `inherit` specified on one of its own dummy arguments. In such cases the rewriting must mimic the decision process usually handled by the compiler, as depicted in Figure 4.7: array `Y` is unknown to the compiler, hence the generated code dynamically checks for the actual mapping in order to call the best version of `iterate`.

```
  subroutine inherited(Y)
    real, dimension(100) :: Y
!hpf$ inherit Y
!hpf$ distribute * onto * :: Y
    ! iterate interface declaration...
    call iterate(Y)
  end subroutine

  subroutine inherited(Y)
    real, dimension(100) :: Y
!hpf$ inherit Y
!hpf$ distribute * onto * :: Y
    ! iterate cloned interface declaration...
    if (Y is distributed block) then
      call iterate_1(Y) ! no remapping
    elif (Y is distributed cyclic)
      call iterate_2(Y) ! no remapping
    elif (Y is not distributed)
      call iterate_3(Y) ! no remapping
    else
      call iterate_1(Y) ! remapping required
    end if
  end subroutine
```

Figure 4.7: Inherited unknown mapping management

4.5 Conclusion

The assume directive proposal for specifying several mappings at subroutine boundaries has important advantages, with respect to its expressiveness, its implementability and its simplicity:

- The full expressiveness of HPF mappings is used.
- Different mappings that depend on the reaching mappings may be specified for local arrays.
- Prescriptive mappings can be used as a default case: indeed the compiler can chose it as the version to call, whatever the incoming distribution, but at price of a remapping.

- It can be rewritten simply into a standard HPF program; thus it serves software engineering quality purposes without many requirements from the compiler technology.
- Hand-optimized library versions with different mapping assumptions can also benefit from this directive, by providing a unified interface to the function. For example, if `matmul` is available as library functions optimized for `(block,block)` (version 1), `(*,block)` (version 2) and *no* (version 3) distributions then the following interface would match these functions:

```
  interface
    subroutine matmul(A,B,C) ! computes A = B x C
      real, dimension(:,:) :: A, B, C
!hpf$ ASSUME
!hpf$   align with A :: B, C
!hpf$   distribute A(block,block)
!hpf$ OR
!hpf$   align with A :: B, C
!hpf$   distribute A(*,block)
!hpf$ OR
!hpf$ END ASSUME
    end subroutine
  end interface
```

- It provides a nice alternative to most uses of `inherit` in any real world program that is expected to be *high performance*, thus it should be considered for the core language. (`inherit` could be moved to approved extensions?)

Users should be warned that specifying an *assume-directive* can lead to high compilation costs due to the possible cloning performed by the compiler.

4.6 Open questions

- Some `empty` or `nothing` or `void` directive could be allowed so as to make it clear that *no* assumption is an assumption within an *assume-directive* when no directives are specified for a case.

```
!hpf$ ASSUME
!hpf$   align with T :: A, B
!hpf$   distribute T(block)
!hpf$ OR
!hpf$   nothing
!hpf$ END ASSUME
```

- consider another keyword instead of `ASSUME`?
- consider another keyword instead of `OR`: `ELSE`, `|` or whatever?
- Derived type mappings? no *assume-directive* within a type declaration, but may be allowed for variable instances.

```
      subroutine sub(a)
```

```
      type mmx
        real det, array(10000,10)
      end type mmx
      type(mmx) :: a
!hpf$ ASSUME
!hpf$   distribute (block,*) :: a%array
!hpf$ OR
!hpf$   distribute (cyclic,*) :: a%array
!hpf$ END ASSUME
      ...
      end subroutine
```

Troisième partie

Compilation du *High Performance Fortran*

Chapitre 1

Introduction

Abstract

This part presents our contributions to HPF compilation. This first chapter introduces in French the next ones. Chapter 2 presents a general linear algebra framework for HPF compilation. Chapter 3 focuses on I/O-related communications. Chapter 4 describes a compilation technique and associated optimizations to manage remappings with array copies. Finally Chapter 5 presents an optimal communication code generation technique for array remappings.

Résumé

Cette partie présente nos contributions à la compilation du langage HPF. Ce premier chapitre introduit les suivants et en résume en français les principales idées. Le chapitre 2 présente un cadre algébrique affine pour modéliser et compiler HPF. Le chapitre 3 se focalise particulièrement sur les communications liées aux entrées-sorties. Le chapitre 4 décrit une technique de compilation et des optimisations associées pour gérer les replacements avec des copies de tableaux. Enfin le chapitre 5 présente la génération de code de communications optimales pour les replacements les plus généraux de HPF.

1.1 Un cadre algébrique pour la compilation de HPF

Le chapitre 2 est le fruit d'un travail commun avec Corinne ANCOURT, François IRIGOIN et Ronan KERYELL [5, 6]. Il présente un cadre général pour la compilation de HPF basé sur l'algèbre linéaire en nombres entiers. L'idée est de montrer que ce cadre permet non seulement de formaliser de manière mathématique les contraintes liées au placement et aux accès, mais aussi peut servir de base à un schéma de compilation général qui inclut les communications et l'allocation des données.

1.1.1 Hypothèses

La section 2.1 présente les hypothèses et montre leur généralité. Nous nous attachons à décrire de manière générale les ensembles d'éléments nécessaires à l'exécution d'un nid de boucles aux bornes affines et aux itérations « indépendantes » (directive HPF `independent`) sur une machine à mémoire répartie. Les accès aux tableaux de données sont supposés également affines. Aucune restriction n'est faite sur le placement des données à travers les directives HPF, si ce n'est que ce placement, incluant le nombre de processeurs, est supposé connu par le compilateur. Ce dernier suivra la règle des calculs locaux.

Dans un souci de généralisation, nous montrons comment des instructions ou constructions `forall`, mais aussi des boucles dont on impose le placement des calculs (extension `on` de FORTRAN D) peuvent être ramenées à des nids de boucles « indépendantes ».

1.1.2 Formalisation des directives

La section 2.2 décrit la formalisation des déclarations et directives de placement du langage en un système d'équations et d'inéquations affines en nombres entiers. L'idée est d'affecter à chaque dimension de chaque objet (tableau, *template*, *processors*) une variable en décrivant les indices. Ensuite, les différentes déclarations sont traduites en contraintes affines sur ces variables et entre ces variables :

Notations : a désigne le vecteur des indices d'un tableau, t celui d'un *template* et p celui d'un arrangement de *processors*. On a par exemple `A(`a_1`,`a_2`)`.

Déclarations : une déclaration de tableau se formalise simplement comme une contrainte de borne sur la variable. Par exemple `A(0:100)` conduit à $0 \leq a_1 \leq 100$. Pour simplifier la présentation toutes les déclarations démarrent à 0.

Alignement : il lie une dimension de tableau à une dimension de *template* au moyen d'une forme affine. Un alignement entre la $i$ème dimension d'un tableau sur la $j$ème du *template*, avec un pas x et un décalage y (`align A(...,i,...) with T(...,x*i+y,...)`) devient l'égalité : $t_j = xa_i + y$.

Distribution : elle place les éléments de *template* sur les processeurs disponibles, dimension par dimension. Dans le cas général des distributions `cyclic(`n`)`, deux variables supplémentaires sont nécessaires.

> **Décalage ℓ :** il mesure le décalage d'un élément à l'intérieur d'un bloc sur un processeur donné. On a la contrainte $0 \leq \ell_k < n$.
>
> **Cycle c :** il désigne le numéro de cycle autour de l'arrangement de processeurs pour un élément de *template* donné.

À l'aide de ces deux variables supplémentaires, on peut décrire une distribution générale de la $j$ème dimension du *template* sur la $k$ème dimension du *processors* de taille 5 (`distribute T(...,cyclic(20),...) onto P`) avec une égalité : $t_j = 100c_k + 5p_k + \ell_k$.

Une formulation matricielle complète est décrite qui incorpore la description des dimensions de duplication. Les solutions entières du système complet décrivent les éléments de tableau, mais aussi leur placement, à travers les variables de dimension du *processors*.

Enfin l'espace d'itération du nid de boucles et les accès aux tableaux sont par hypothèse également placés dans un cadre affine, après normalisation des boucles. Ces ensembles peuvent dépendre de paramètres de structures disponibles à l'exécution.

1.1.3 Schéma de compilation

La section 2.3 survole le schéma de compilation proposé. Il s'agit de décrire précisément, au moyen de la formalisation décrite à la section précédente, les ensembles de données paramétriques (fonction de l'identité du processeur, mais aussi de paramètres de structures) nécessaires à la compilation : données locales, données à calculer, données à communiquer, etc. On se limite à un nid de boucles parallèles. Ces ensembles sont définis en combinant les systèmes d'équations.

Local : les données locales à un processeur sont simplement dérivées du système décrivant le placement et les déclarations.

Calcul : les données à calculer sont déterminées avec la règle des calculs locaux. Il s'agit des données locales qui sont définies par le nid de boucles. Cet ensemble est calculé en faisant l'intersection des données locales avec les données définies par le nid de boucles.

Besoin : les données dont a besoin un processeur sont celles nécessaires à ses calculs locaux. La détermination de ces données est une source potentielle d'approximation.

Envoi : les données à envoyer d'un processeur p à un processeur q sont celles dont a besoin q et que p possède. Il s'agit donc d'une intersection d'ensembles paramétriques.

À partir de ces différents ensembles on peut décrire un code SPMD de haut niveau qui les énumère pour générer les communications ou bien effectuer les calculs. L'idée du code est de faire d'abord les communications nécessaires, puis les calculs.

1.1.4 Raffinement

La section 2.4 précise le comment et non pas seulement le quoi. Le but est de décrire les solutions envisagées pour l'énumération des différents ensembles, et aussi de montrer comment ces solutions suggèrent des techniques d'allocation et d'adressage en mémoire pour les données locales comme pour les données temporaires.

Énumération

Elle est basée sur une méthode générale de parcours de Z-polytopes. Cependant cette méthode n'exploite pas de manière particulière les égalités qui sont nombreuses dans notre modélisation, ce qui peut conduire à des codes très inefficaces. Pour réduire ce problème, un changement de repère est proposé, qui s'appuie sur le calcul de formes de HERMITE. L'idée est de se ramener à un polytope dense en les exploitant, tout en maîtrisant l'ordre des variables du parcours pour utiliser au mieux le cache.

Résolution symbolique

Lorsque toute une dimension est accédée au moyen d'une section de tableau, la méthode d'énumération précédente peut être calculée de manière symbolique. On obtient alors de manière explicite et dense les solutions décrivant les éléments des tableaux locaux à un processeur. L'intérêt de cette résolution manuelle est double. D'une part, ce cas particulier est le cas fondamental traité par d'autres travaux, il est donc important de montrer ce qui peut être fait pour comparaison. D'autre part, il sert de base à l'allocation et l'adressage en mémoire des tableaux.

Allocation

La résolution symbolique permet de densifier l'ensemble des points. Pour un processeur donné, donc en fixant l'identité du processeur, cet ensemble paramétrique est l'ensemble des éléments locaux. La vue densifiée de cet ensemble est un candidat idéal pour l'allocation en mémoire contiguë, de manière à réduire l'espace perdu. Cette solution a notamment trois avantages : d'abord, puisque décidée en conjonction avec la technique d'énumération, elle favorise donc le fonctionnement du cache en parcourant les données dans le sens de stockage ; ensuite, elle est compatible avec l'analyse des recouvrements ; enfin elle intègre naturellement les calculs d'adresses locales au processus d'énumération.

Temporaires

Certaines données non locales doivent être temporairement stockées et référencées localement pour effectuer un calcul. Si les recouvrements n'offrent pas de solution satisfaisante, alors il est suggéré d'allouer en fonction de l'espace d'itération et de la dimension des accès. Si l'accès est plein, on peut par exemple allouer les données locales comme les données définies par le nid de boucles. Si l'accès n'est pas plein, des transformations supplémentaires permettent de se ramener à un cas plein (en dimensions) et dense (pas de trous réguliers).

La génération de code doit jouer avec ces différents espaces d'adressage et d'itération définis pour chaque tableau référencé dans le nid de boucles parallèles.

1.1.5 Conclusion

La section 2.5 présente quelques exemples de codes générés concernant l'adressage, les calculs et les communications. Les travaux d'autres équipes sur ces problèmes sont présentés avant de conclure.

1.2 Compilation des communications liées aux E/S

Le chapitre 3 étudie la compilation des communications liées aux entrées-sorties (E/S). Il a été publié [61]. Le premier problème est de déterminer l'ensemble des données qui font l'objet de ces opérations. Ensuite, il faut générer les communications qui amèneront ces données à leur lieu d'utilisation. Dans un premier temps, le modèle un hôte pour les E/S et plusieurs nœuds pour les calculs est utilisé. Ce modèle est représentatif par exemple d'un réseau de stations de travail qui dépendent d'un même serveur pour le stockage permanent des données. On généralise ensuite aux entrées-sorties pour un modèle de machine parallèle.

1.2.1 Formalisation du problème

La section 3.2 présente les analyses et algorithmes utilisés pour décrire mathématiquement les données qui font l'objet d'entrées-sorties. Il faut (1) décrire les éléments référencés, (2) décider du placement dans le code des communications et (3) modéliser le placement des données.

Les données concernées par les entrées-sorties sont caractérisées de manière paramétrique et affine grâce à l'analyse des régions. Cette analyse donne une sur-approximation des éléments de tableau référencés par une section de code, lus (région `read`) ou écrits (région `write`). Elle fournit également une information sur l'exactitude du résultat (approximation `must` si exacte et `may` si la région est un sur-ensemble des éléments référencés).

La seconde question concerne le placement dans le code des mouvements de données. Celui-ci est choisi au plus haut niveau possible, en remontant par exemple les boucles ne contenant que des instructions d'entrées-sorties. Les communications sont ainsi factorisées. Les directives HPF et les déclarations sont modélisées comme précédemment, sous forme de systèmes d'équations et d'inéquations affines.

À partir de cette modélisation affine des données concernées, il faut générer le code de communication entre l'hôte et les nœuds. Pour ce faire le système est d'abord nettoyé des variables inutiles (insérées par la construction systématique du système) par projection. L'algorithme de Ancourt et Irigoin [7] est ensuite appliqué pour produire un système dont les solutions sont directement énumérables avec des boucles. L'énumération des éléments paramétrée par les processeurs concernés, l'énumération de ces processeurs et une condition qui détecte une éventuelle entrée-sortie vide sont séparées. Dans le cas le plus général de placement, qui comporte de la duplication éventuellement partielle, les processeurs sont partitionnés en groupe partageant les mêmes données. L'un deux devra être choisi pour les envois, et tous devront recevoir les mises à jour.

1.2.2 Génération de code

La section 3.3 discute de code à générer dans différents cas. Quatre cas sont à étudier, selon que les données concernées sont réparties ou non, et selon qu'elles sont lues ou écrites.

Le principe est de collecter si nécessaire sur l'hôte les données lorsque celles-ci sont référencées en lecture (`PRINT`), et au contraire de faire des mises à jour à partir de l'hôte lorsque celles-ci sont définies (`READ`) par l'opération d'entrée-sortie. Cependant il faut veiller à gérer correctement l'approximation due à l'analyse des régions. En effet, on est éventuellement contraint d'envoyer un sur-ensemble des données lors d'une mise à jour, ce qui peut écraser à tort des valeurs qui n'ont pas été redéfinies. Dans ce cas précis, la solution proposée est de collecter a priori les données avant l'opération, pour assurer que la mise à jour se fasse avec les valeurs souhaitées, redéfinies ou initiales. Ces communications supplémentaires sont évitées lorsque le compilateur sait que la description est exacte.

Le code de communication généré pour les différents cas exploite simplement de manière complémentaire entre l'hôte et les nœuds l'énumération des solutions du système représentant les différents éléments de tableau concernés par l'entrée-sortie et leur placement. L'hôte utilise la description des processeurs pour les énumérer, tandis que les processeurs l'utilisent pour se sélectionner ou non. Il est montré que les messages sont bien générés dans le même ordre de part et d'autre, et que l'envoi et la réception sont bien symétriques par construction, ce qui assure la correction. La taille des buffers nécessaire est bornée par la taille de l'ensemble des données concernées.

1.2.3 Extensions

La section 3.4 décrit des extensions et améliorations possibles, pour produire du code plus efficace et aussi pour utiliser des entrées-sorties parallèles.

Tout d'abord on note que le code généré par l'implantation des algorithmes au sein de notre prototype de compilateur HPF (HPFC, présenté au chapitre 3) incorpore déjà des optimisations : les variables inutiles ont été éliminées, les conditions simplifiées pour ne garder que le nécessaire, et le code parcourt les données dans le sens du cache. Cependant d'autres améliorations sont encore possibles. Le code peut bénéficier de nombreuses optimisations standard comme le déplacement du code invariant, la réduction de force, mais aussi de transformations plus globales comme la modification du polyèdre initial pour le rendre dense.

Les travaux présentés sont ensuite étendus pour un cas particulier d'entrées-sorties parallèles. L'idée est d'affecter les processeurs non plus seulement à un seul hôte, mais plutôt à un ensemble de processeurs d'entrées-sorties. Il est suggéré de pouvoir déclarer de tels processeurs et de leur affecter ensuite les processeurs de calculs au moyen d'une directive de type distribution. Il faut reconnaître qu'une telle extension est un peu trop restrictive et contrainte par rapport à la généralité des cas envisageables. Cependant cette description se rapproche de machines existantes, comme la PARAGON, et les techniques décrites sont directement généralisables à ce cas.

La conclusion présente les travaux d'autres groupes dans ce domaine, essentiellement basés sur l'utilisation de librairies qui doivent gérer les différentes distributions.

1.3 Compilation des replacements par copies

Le chapitre 4 décrit une technique de compilation pour gérer les replacements au moyen de copies de tableaux. Le mot replacement est employé ici comme traduction de *remapping*. Il désigne dans la spécification de HPF la modification dynamique du placement d'un tableau. Cette modification peut être due soit aux directives exécutables `realign` ou `redistribute`, soit aux placements prescriptifs des arguments muets d'une routine lorsque celle-ci est appelée et que les arguments réels doivent s'y conformer.

L'idée est de traduire un code avec des placements dynamiques en un code purement statique. Les replacements d'un tableau sont assurés par des recopies entre différentes versions du tableaux placées statiquement. Cette technique permet également de supprimer automatiquement de nombreux replacements inutiles, soit parce que un tableau n'est jamais référencé avec le nouveau placement, soit parce que les valeurs d'une copie du tableau avec le placement désiré sont encore vivantes. Ces techniques de compilation et d'optimisation corrigent, complètent et étendent celles présentées par Hall *et al.* [121].

La technique présentée nécessite quelques contraintes supplémentaires sur le langage HPF par rapport à sa définition, pour garantir que le compilateur puisse connaître le placement des données de manière statique. Dans ce cas, une réécriture systématique du HPF avec replacements en une version HPF avec placements statiques est possible. Il faut noter que ces contraintes simplificatrices n'empêcheraient le portage d'aucune des applications réelles que nous avons rencontrées et qui bénéficieraient de replacements.

1.3.1 Survol de l'approche suivie

La section 4.2 détaille l'approche suivie, en discutant précisément les restrictions nécessaires imposées sur le langage, ainsi que la gestion des appels de procédure avec des tableaux répartis en paramètres.

Pour que la traduction des replacements en simples copies soit possible, il faut que le compilateur connaisse le placement requis aux différents points de redistribution. Pour cela, il faut que les replacements soient aussi contraints que les placements statiques, il ne peuvent donc plus dépendre de valeurs disponibles à l'exécution seulement. Pour substituer aux différentes références une seule copie de tableau, il faut proscrire les cas ambigus de placements liés au flot de contrôle. Enfin il faut que les placements requis par les appelés pour leurs arguments soient connus. Il faut donc que les interfaces décrivant ce placement soient obligatoires (c'est déjà le cas de langages comme C++, Pascal, ADA, il semble donc que les utilisateurs puissent se faire à ce type de contraintes, surtout si elles sont justifiées par la possibilité d'avoir de meilleures techniques de compilation et de meilleures performances).

Pour les appels de sous-routines, nous suivons l'approche proposée dans [121], qui charge l'appelant de se conformer au placement attendu par l'appelé. Cette approche permet beaucoup plus d'optimisations car plus d'information est disponible. Pour les placements d'arguments descriptifs, le compilateur doit vérifier que le placement est bien conforme. Dans le cas prescriptif, une recopie vers un tableau conforme au placement attendu de l'argument sera générée. Enfin le cas transcriptif est plus simple et plus compliqué à la fois. Du point de vue de l'appelant, l'appelé est sensé gérer n'importe quel placement, donc a priori pas de problème. Cependant, du côté de l'appelé, la compilation du code est complexe et le code généré sera plutôt inefficace car aucune information n'est donnée au compilateur qui doit tout déléguer à l'exécutif. La gestion des références sera potentiellement très coûteuse, et notre idée du langage et de sa compilation est plutôt de proscrire de tels placements au profit d'autres extensions discutées à la partie conception, comme la directive `assume` (chapitre II.4).

1.3.2 Graphe des replacements

La section 4.3 présente le graphe des replacements et sa construction. Ce graphe est un sous-graphe du graphe de contrôle, qui ne garde que les sommets contenant des replacements, ainsi que des sommets additionels pour représenter le placement initial, l'appel de la routine et son retour. Les arcs dénotent un chemin possible dans le graphe de contrôle avec un même tableau replacé aux deux sommets. Les sommets sont étiquetés avec les copies entrantes et sortantes de chaque tableau dynamique, et avec des informations sur l'utilisation des données, qui permettront de tracer les copies de tableau vivantes.

L'algorithme de construction du graphe consiste à propager des copies avec les placements initiaux pour chaque tableau à partir du point d'entrée dans la routine. La propagation est arrêtée par les replacements rencontrés qui affectent la copie. Une nouvelle copie avec le nouveau placement devra être propagée à partir du sommet en question. Au fur et à mesure de la propagation des copies, les références aux tableaux sont mises à jour avec la copie à référencer, et les sommets, arcs et étiquettes nécessaires sont ajoutés au graphe des replacements.

1.3.3 Optimisations sur ce graphe

La section 4.4 présente les optimisations effectuées sur le graphe des replacements. D'abord, les replacements dont la copie sortante n'est jamais référencée sont retirés. Le retrait de ces copies inutiles change cependant les copies entrantes, qui peuvent atteindre les différents sommets du graphe des replacements. Elle sont donc remises à jour à partir des copies effectivement utilisées et en suivant les chemins sans replacement intermédiaire.

La seconde optimisation envisagée est inhérente à la technique par recopies entre tableaux. L'idée est que lorqu'une copie est effectuée et que le nouveau tableau n'est référencé qu'en lecture, il est possible de garder la copie initiale qui est encore valable et peut être utilisée en cas de besoin sans mettre à jour ses valeurs. De plus, de telles situations peuvent apparaître selon le chemin suivi dans le flot de contrôle à l'exécution. Notre technique délègue donc à l'exécutif la gestion de cette information, mais le compilateur réduit l'ensemble des copies à garder à celles qui peuvent s'avérer utile, de manière à réduire les coûts mémoire de la conservation des copies de tableau.

Enfin des optimisations suggérées dans [121] sont abordées. Il s'agit d'une part de la détection des tableaux dont les valeurs ne seront pas réutilisées, et dont on peut éviter les communications liées aux replacements. Une directive (`kill`) ajoutée permet à notre compilateur de disposer de cette information. D'autre part, les replacements invariants dans un corps de boucles peuvent être extraits et mis à l'extérieur de la boucle. Nous soulignons alors qu'il faut que la boucle soit bien exécutée une fois pour que cette modification soit à coup sûr une amélioration, et proposons un code optimisé différent pour éviter ce genre de problème.

1.3.4 Implications sur l'exécutif

La section 4.5 discute des implications sur l'exécutif des différentes techniques décrites. L'exécutif doit maintenir à jour quelques informations : le placement courant des tableaux, et, pour un tableau donné, l'ensemble des copies dont les valeurs sont à jour et qui n'ont donc pas besoin d'être recopiés en cas de replacement.

Nous présentons ensuite l'algorithme qui génère pour chaque sommet de replacement du graphe de contrôle le code qui effectuera à l'exécution les recopies requises. Il se base sur le graphe optimisé des replacements. Le code généré maintient à jour les structures de données décrivant l'état des tableaux et de leurs copies.

Ces techniques permettent de transformer, sous réserve du respect de contraintes raisonnables par le programme considéré, un code HPF avec replacements en un code à placement statique avec des recopies entre tableaux. Elles permettent également de nombreuses optimisations qui évitent des replacements inutiles, détectés comme tels à la compilation ou même à l'exécution.

1.4 Compilation optimale des replacements

Ce chapitre est issu d'un travail en commun effectué avec Corinne Ancourt [68]. La contribution essentielle est la description d'une technique de compilation des communications liées à un replacement (réalignement ou redistribution). Cette technique gère tous les cas de placements HPF, y compris les alignements non unitaires, les distributions les plus générales, mais aussi la duplication partielle des données dont on tire bénéfice en utilisant des communications de un à plusieurs *(broadcast)* et en équilibrant la charge. La formalisation adoptée permet de démontrer des résultats d'optimalité tant en ce qui concerne la latence que la bande passante : les coûts de communication sont donc minimisés.

1.4.1 Motivation et autres travaux

La section 5.1 introduit et motive ce travail. Elle discute les autres travaux effectués sur cette question.

Les replacements sont très utiles pour toute une classe d'applications dont le placement optimal des données n'est pas constant tout au long des calculs. On trouve des exemples de traitement du signal (FFT, radar), mais aussi dans les techniques d'algèbre linéaire (ADI, résolutions de systèmes par ScaLapack ...). Les réalignements sont envisagés comme technique de compilation pour le langage HPF. Enfin les outils qui suggèrent de manière automatique des placements fonctionnent également par phases et aboutissent naturellement à des replacements entre ces phases. Tous ces exemples montrent à l'évidence l'intérêt de la dynamicité du placement et de sa compilation la plus efficace possible.

De nombreux travaux ont donc été consacrés à ce problème particulier. Il faut cependant noter que toute technique se proposant de compiler une affectation générale entre deux tableaux doit gérer des communications similaires, et est donc candidate pour les redistributions. Cependant, aucune de ces techniques ne gère le cas le plus général, mais au contraire chacune fait des hypothèses simplificatrices qui en réduisent la portée, le plus souvent en négligeant les alignements et la duplication. Notre technique est donc la seule à incorporer toute la problématique dans sa généralité, en incluant non seulement l'adressage local des tableaux sur les nœuds, mais encore le partage de charge ou de surcharge des nœuds.

Notre technique compile les communications en déterminant tout d'abord les couples de placements source et cible pour un tableau donné qui peuvent faire l'objet à l'exécution d'un replacement. En déterminant ces couples à l'aide d'une formulation classique basée sur les flots de données, quelques optimisations sont faites qui retirent les replacements inutiles. Ensuite, le compilateur génère un code de communication SPMD pour chaque couple de placement. La pertinence de la technique est montrée de manière mathématique dans le cas général. On minimise les coûts de communications en nombre de messages mais aussi en quantité de données échangées. La preuve est rendue possible par la formalisation adoptée qui est basée sur l'algèbre linéaire en nombres entiers.

1.4.2 Formalisation dans le cadre algébrique affine

La section 5.2 décrit la formalisation du problème sous forme de système d'équations et d'inéquations affines. Pour ce qui est des placements source et cible et de l'incorporation de l'adressage local, cette formalisation est la même que celle présentée dans les chapitres précédents. Les solutions du système d'équations obtenues décrivent les éléments de tableau et leurs placements sur les processeurs sources et cibles. On cherche donc à énumérer ces solutions pour générer les messages nécessaires entre processeurs, et effectuer les communications requises par le replacement.

Cependant, du fait de la duplication, il n'y a pas forcément une solution unique pour chaque élément de tableau. On ne peut pas affecter une seule source à un processeur cible. On veut donc profiter de cette duplication pour générer un meilleur code si possible. Du côté des processeurs cibles, la duplication regroupe des processeurs attendant *exactement* les mêmes données, ce qui suggère donc de leur envoyer les mêmes messages à l'aide de communications générales de type *broadcast*. Du côté des processeurs sources, la duplication regroupe des processeurs disposant des mêmes données, l'idée naturelle est que chacun de ces processeurs s'occupe de processeurs cibles distincts. Cela permet d'équilibrer la charge de l'envoi des messages.

Pratiquement, pour les diffusions, il suffit de séparer les équations portant sur les dimensions de duplication de l'arrangement de processeurs cibles. Pour le partage de charge, une simple équation supplémentaire liant dimensions cibles distribuées et dimensions sources dupliquées suffit à produire le résultat escompté. L'équation revient à distribuer de manière cyclique les processeurs cibles sur les processeurs sources disponibles. On obtient finalement un système d'équations dont les solutions entières matérialisent les communications souhaitées, y compris l'adressage local des données sur les processeurs et le partage de charge. Il reste alors à générer le code des processeurs qui énumérera ces solutions et réalisera les communications.

1.4.3 Génération du code SPMD

La section 5.3 présente le code généré à partir du système d'équations. Le système précedent est projecté sur les processeurs pour déterminer un ensemble des couples de processeurs qui peuvent vouloir communiquer. Cette projection n'étant pas nécessairement exacte, on obtient un sur-ensemble des processeurs pouvant communiquer. Cependant ce sur-ensemble reste généralement petit par rapport a l'ensemble de *tous* les couples processeurs possibles. Les couples sont énumérés de manière symétrique par le code d'envoi et le code de réception. Les sources se sélectionnent et énumèrent leurs cibles, alors que les cibles se sélectionnent et énumèrent leurs sources. Pour chaque couple de processeurs, les éléments à envoyer ou recevoir sont ensuite parcourus afin de fabriquer le message correspondant.

Lorsque les données sont disponibles localement, une recopie locale est effectuée. Le code produit est donc composé de deux parties. La première est l'envoi. La seconde regroupe la réception et la recopie. Cette recopie éventuelle des données disponibles localement est retardée de manière à accélérer la fabrication et l'envoi des messages.

Quelques gardes et astuces sont utilisées dans le code produit. D'abord, les messages vides ne sont pas envoyés ... le problème est alors de ne pas les recevoir! Pour ce faire, la réception éventuelle d'un message est retardée au moment ou un élément doit en être extrait, garantissant ainsi qu'il n'est pas vide et aura donc bien été envoyé. De plus, les messages ne sont envoyés qu'aux processeurs qui ne sont pas des *jumeaux* du processeur courant, *i.e.* des processeurs possédant les mêmes données du fait de la duplication. Ces gardes sont effectuées à l'exécution, parce que HPF ne permet aucune hypothèse sur la manière dont les processeurs HPF de différents arrangements de processeurs sont placés sur les processeurs réels. Enfin nous discutons quelques optimisations supplémentaires pour éviter les problèmes de surcharge des nœuds (plusieurs processeurs cherchent à envoyer un message à un même processeur cible).

1.4.4 Optimalité et expériences

La section 5.4 présente les résultats théoriques et pratiques de notre technique de compilation.

Tout d'abord la méthode est optimale, tant en ce qui concerne les coûts de latence que de bande passante. La démonstration est détaillée en annexe. Elle s'appuie sur des propriétés intrinsèques des placements permis par HPF, particulièrement sur l'orthogonalité entre dimensions distribuées et dimensions dupliquées. Elle utilise le fait que la description à base d'équations linéaires est exacte. Enfin elle requiert les astuces de programmation et les gardes introduites dans le code pour éviter l'envoi de messages vides et ne pas communiquer avec des processeurs jumeaux. La correction du code peut également être montrée en raisonnant sur la structure du code produit, en vérifiant que les variables sont proprement instanciées.

Notre technique est implantée au sein de notre prototype de compilateur HPF. Du code basé sur PVM est produit. Un certain nombre d'expériences ont été effectuées sur la ferme d'Alpha du LIFL (Laboratoire d'Informatique Fondamentale de Lilles). Elles ont consisté à mesurer les temps d'exécution de transpositions de matrice, pour des arrangements de processeurs et distributions différents. Les meilleurs temps ont été conservés : les temps moyens n'étaient pas significatifs car la machine n'était pas dédiée à nos expériences. Les résultats obtenus sont encourageants, en particulier vue l'implantation à haut niveau de notre technique : les performances sont de 20 à 30% meilleures que celles du compilateur HPF de DEC. Ces tests ont également montrés que certaines optimisations et transformations de codes, autant particulières que usuelles, étaient indispensables pour obtenir des codes d'énumération rapides.

1.4.5 Annexe

La section 5.5 détaille les points qui n'ont été qu'abordés dans le corps de l'article, par manque de place. Les notations et la formalisation sont d'abord revues, en soulignant les propriétés utiles à la preuve d'optimalité. Ensuite, le code généré est revu attentivement. Les résultats d'optimalité sont ensuite énoncés et prouvés. Enfin les conditions expérimentales et les mesures brutes sont détaillées.

Chapitre 2

A Linear Algebra Framework for Static HPF Code Distribution

Corinne ANCOURT, Fabien COELHO, François IRIGOIN et Ronan KERYELL

Une version préliminaire de ce chapitre a fait l'objet d'une communication au Workshop *CPC'93 à Delft [5]. Il doit paraître sous cette forme dans* Scientific Programming *[6].*

Résumé

Fortran HPF a été développé pour permettre la programmation à parallélisme de données des machines à mémoire répartie. Le programmeur peut utiliser l'espace mémoire uniforme qui lui est familier pour la programmation séquentielle et spécifie la répartition des données par des directives. Le compilateur utilise ces directives pour allouer des tableaux dans les mémoires locales, pour attribuer les itérations aux processeurs élémentaires et pour déplacer les données nécessaires entre processeurs. Nous montrons que l'algèbre linéaire fournit un cadre adéquat pour encoder les directives HPF et pour synthétiser du code réparti pour des boucles parallèles. La mémoire est utilisée à bon escient, les bornes de boucles sont ajustées et les communications sont minimales. L'utilisation systématique de l'algèbre linéaire rend possible les preuves de correction et d'optimalité du schéma de compilation proposé. Le code généré incorpore les optimisations usuelles que sont l'élimination des gardes, la vectorisation et l'agrégation des messages ainsi que l'analyse des recouvrements.

Abstract

HIGH PERFORMANCE FORTRAN (HPF) was developed to support data parallel programming for SIMD and MIMD machines with distributed memory. The programmer is provided with a familiar uniform logical address space and specifies the data distribution by directives. The compiler then exploits these directives to allocate arrays in the local memories, to assign computations to elementary processors and to migrate data between processors when required. We show here that linear algebra is a powerful framework to encode HPF directives and to synthesize distributed code with space-efficient array allocation, tight loop bounds and vectorized communications for `INDEPENDENT` loops. The generated code includes traditional optimizations such as guard elimination, message vectorization and aggregation, overlap analysis... The systematic use of an affine framework makes it possible to prove the compilation scheme correct.

2.1 Introduction

Distributed memory multiprocessors can be used efficiently if each local memory contains the right pieces of data, if local computations use local data and if missing pieces of data are quickly moved at the right time between processors. Macro packages and libraries are available to ease the programmer's burden but the level of details still required transforms the simplest algorithm, *e.g.* a matrix multiply, into hundreds of lines of code. This fact decreases programmer productivity and jeopardizes portability, as well as the economical survival of distributed memory parallel machines.

Manufacturers and research laboratories, led by Digital and Rice University, decided in 1991 to shift part of the burden onto compilers by providing the programmer a uniform address space to allocate objects and a (mainly) implicit way to express parallelism. Numerous research projects [105, 131, 241] and a few commercial products had shown that this goal could be achieved and the High Performance Fortran Forum was set up to select the most useful functionalities and to standardize the syntax. The initial definition of the new language, HPF, was frozen in May 1993, and corrections were added in November 1994 [96]. Prototype compilers incorporating some HPF features are available [38, 39, 56, 241, 260, 21]. Commercial compilers from APR [177, 178], DEC [195, 28], IBM [116] and PGI [80, 188] are also being developed or are already available. These compilers implement part or all of the HPF Subset, which only allows static distribution of data and prohibits dynamic redistributions.

This paper deals with this HPF static subset and shows how changes of basis and affine constraints can be used to relate the global memory and computation spaces seen by the programmer to the local memory and computation spaces allocated to each elementary processor. These relations, which depend on HPF directives added by the programmer, are used to allocate local parts of global arrays and temporary copies which are necessary when non-local data is used by local computations. These constraints are also used in combination with the *owner-computes rule* to decide which computations are local to a processor, and to derive loop bounds. Finally they are used to generate send and receive statements required to access non-local data.

These three steps, local memory allocation, local iteration enumeration and data communication, are put together as a general compilation scheme for parallel loops, known as `INDEPENDENT` in HPF, with affine bounds and subscript expressions. HPF's `FORALL` statements or constructs, as well as a possible future `ON` extension to advise the compiler about the distribution of iterations onto the processors, can be translated into a set of independent loops by introducing a temporary array mapped as required to store the intermediate results. These translations are briefly outlined in Figures 2.1 and 2.2. The resulting code is a pair of loops which can be compiled by our scheme, following the *owner-computes rule*, if the `ON` clause is put into the affine framework. The `FORALL` translation requires a temporary array due to its SIMD-like semantics. However, if the assigned array is not referenced in the *rhs*, the `FORALL` loop is independent and should be tagged as such to fit directly our scheme. Such *necessary* temporary arrays are not expected to cost much, both on the compilation and execution point of view: The allocated memory is reusable (it may be allocated on the stack), and the copy assignment on local data should be quite fast.

```
! non-independent
!   A in the rhs may
!   induce RW dependences...
FORALL (i=1:n, j=1:m, MASK(i,j))
  A(i,j) = f(A, ...)
```

```
! array TMP declared and mapped as A
! initial copy of A into TMP
!   because of potential RW dependences
INDEPENDENT(j,i)
do j=1, m
  do i=1, n
    TMP(i,j) = A(i,j)
  enddo
enddo
! possible synchro...
INDEPENDENT(j,i)
do j=1, m
  do i=1, n
    if (MASK(i,j)) A(i,j) = f(TMP, ...)
  enddo
enddo
```

Figure 2.1: Masked `FORALL` to `INDEPENDENT` loops

```
INDEPENDENT(j,i), ON(...)
do j=1, m
  do i=1, n
    A(i,j) = f(...)
  enddo
enddo
```

```
! array TMP(iteration space)
!   mapped as ON(iteration space)
INDEPENDENT(j,i)
do j=1, m
  do i=1, n
    TMP(i,j) = f(...)
  enddo
enddo
! possible synchro...
INDEPENDENT(j,i)
do j=1, m
  do i=1, n
    A(i,j) = TMP(i,j)
  enddo
enddo
```

Figure 2.2: `ON` to `INDEPENDENT` loops

This compilation scheme directly generates optimized code which includes techniques such as guard elimination [105], message vectorization and aggregation [131, 241]. It is compatible with overlap analysis [105]. There are no restrictions neither on the kind of distribution (general cyclic distributions are handled), nor on the rank of array references (the dimension of the referenced space: for instance rank of `A(i,i)` is 1). The memory allocation part, whether based on overlap extensions, or dealing with temporary arrays needed to store both remote and local elements, is independent of parallel loops and can always be used. The relations between the global programmer space and the local processor spaces can also be used to translate sequential loops with a run-time resolution mechanism or with some optimizations. The reader is assumed knowledgeable in HPF directives [96] and optimization techniques for HPF [105, 241].

The paper is organized as follow. Section 2.2 shows how HPF directives can be expressed as affine constraints and normalized to simplify the compilation process and its description. Section 2.3 presents an overview of the compilation scheme and introduces the basic sets *Own*, *Compute*, *Send* and *Receive* that are used to allocate local parts of HPF arrays and temporaries, to enumerate local iterations and to generate data exchanges between processors. Section 2.4 refines these sets to minimize the amount of memory space allocated, to reduce the number of loops whenever possible and to improve the communication pattern. This is achieved by using different coordinates to enumerate the same sets. Examples are shown in Section 2.5 and the method is compared with previous work in Section 2.6.

2.2 HPF directives

The basic idea of this work was to show that the HPF compilation problem can be put into a linear form, including both equalities and inequalities, then to show how to use polyhedron manipulation and scanning techniques to compile an HPF program from this linear form. The linear representations of the array and HPF declarations, the data mapping and the loop nest accesses are presented in this section.

HPF specifies data mappings in two steps. First, the array elements are *aligned* with a *template*, which is an abstract grid used as a convenient way to relate different arrays together. Each array element is assigned to at least one template cell thru the `ALIGN` directive. Second, the *template* is *distributed* onto the *processors*, which is an array of virtual processors. Each template cell is assigned to one and only one processor thru the `DISTRIBUTE` directive. The *template* and *processors* are declared with the `TEMPLATE` and `PROCESSORS` directives respectively.

Elements of arrays aligned on the same template cell are allocated on the same elementary processor. Expressions using these elements can be evaluated locally, without inter-processor communications. Thus the alignment step mainly depends on the algorithm. The template elements are packed in blocks to reduce communication and scheduling overheads without increasing load imbalance too much. The block sizes depend on the target machine, while the load imbalance stems from the algorithm. *Templates* can be bypassed by aligning an array on another array, and by distributing array directly on processors. This does not increase the expressiveness of the language but implies additional check on HPF declarations. Templates are systematically used in this paper to simplify algorithm descriptions. Our framework deals with both stages and could easily tackle direct alignment and direct distribution. The next sections show that any HPF directive can be expressed as a set of affine constraints.

2.2.1 Notations

Throughout this paper, a lower case letter as v denotes a vector of integers, which may be variables and constants. $v_i, (i \geq 1)$ the ith element or variable of vector v. Subscript 0, as in v_0, denotes a constant integer vector. As a convention, a denotes the variables which describe the elements of an array, t is used for *templates* and p for *processors*. An upper case letter as A denotes a constant integer matrix. Constants are implicitly expanded to the required number of dimensions. For instance 1 may denote a vector of 1. $|A|$ denotes the determinant of matrix A.

2.2.2 Declarations

The data arrays, the *templates* and the *processors* are declared as Cartesian grids in HPF. If a is the vector of variables describing the dimensions of array `A(3:8,-2:5,7)`, then the following linear constraints are induced on a:

$$3 \leq a_1 \leq 8, \quad -2 \leq a_2 \leq 5, \quad 1 \leq a_3 \leq 7$$

These may be translated into the matrix form $D_{\mathtt{A}} a \leq d$ where:

$$D_{\mathtt{A}} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{pmatrix}, \quad a = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}, \quad d = \begin{pmatrix} 8 \\ -3 \\ 5 \\ 2 \\ 7 \\ -1 \end{pmatrix}$$

Any valid array element must verify the linear constraints, *i.e.* $\mathtt{A}(a_1, a_2, a_3)$ is a valid array element if Equation (2.1) is verified by vector a. In the remaining of the paper it is assumed without loss of generality that the dimension lower bounds are equal to 0. This assumption simplifies the formula by deleting a constant offset at the origin. Thus the declaration constraints for `A(0:5,0:7,0:6)` can be written intuitively as

$$0 \leq a < D.1 \tag{2.1}$$

where

$$D.1 = \begin{pmatrix} 6 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 7 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 6 \\ 8 \\ 7 \end{pmatrix}$$

D is a diagonal matrix composed of the sizes of `A`. The rationale for storing the sizes in a diagonal matrix instead of a vector is that this form is needed for the distribution formula (see Section 2.2.4). Likewise the template declaration constraints are $0 \leq t < T.1$ and the processors $0 \leq p < P.1$.

2.2.3 Alignment

The `ALIGN` directive is used to specify the alignment relation between one object and one or many other objects through an affine expression. The alignment of an array `A` on a template `T` is an affine mapping from a to t. Alignments are specified dimension-wise with integer affine expressions as template subscript expressions. Each array index can be used at most once in a template subscript expression in any given alignment, and each subscript expression cannot contain more than one index [96]. Let us consider the following HPF alignment directive, where the first dimension of the array is collapsed and the most general alignment subscripts are used for the other dimensions:

```
align A(*,i,j) with T(2*j-1,-i+7)
```

it induces the following constraints between the dimensions of `A` and `T` represented respectively by vector a and t:

$$t_1 = 2a_3 - 1, \quad t_2 = -a_2 + 7$$

Thus the relation between `A` and `T` is given by $t = Aa + s_0$ where

$$t = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}, \quad A = \begin{pmatrix} 0 & 0 & 2 \\ 0 & -1 & 0 \end{pmatrix}, \quad a = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}, \quad s_0 = \begin{pmatrix} -1 \\ 7 \end{pmatrix}$$

A (for alignment) is a matrix with at most one non-zero integer element per column (dummy variables must appear at most once in the alignment subscripts) and per row (no more than one dummy variable in an alignment subscript), and s_0 is a constant vector that denotes the constant shift associated to the alignment. Note that since the first dimension of the array is collapsed the first column of A is null, and that the remaining columns constitute an anti-diagonal matrix since `i` and `j` are used in reverse order in the subscript expressions.

However this representation cannot express replication of elements on a dimension of the template, as allowed by HPF. The former relation is generalized by adding a projection matrix R (for replication) which selects the dimensions of the template which are actually aligned. Thus the following example:

```
align A(i) with T(2*i-1,*)
```

is translated into

$$Rt = Aa + s_0 \tag{2.2}$$

where $R = \begin{pmatrix} 1 & 0 \end{pmatrix}, \ t = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}, \ A = \begin{pmatrix} 2 \end{pmatrix}, \ a = \begin{pmatrix} a_1 \end{pmatrix}, \ s_0 = \begin{pmatrix} -1 \end{pmatrix}$

The same formalism can also deal with a chain of alignment relations (`A` is aligned to `B` which is aligned to `C`, etc.) by composing the alignments. To summarize, a "`*`" in an array reference induces a zero column in A and a "`*`" in a template reference a zero column in R. When no replication is specified, R is the identity matrix. The number of rows of R and A corresponds to the number of template dimensions that are *actually* aligned, thus leading to equations linking template and array dimensions. Template dimensions with no matching array dimensions are removed through R.

```
template T(0:99), T'(0:99)
processors P(0:4)
distribute T(block(20)), T'(cyclic(1)) onto P
```

Figure 2.3: Example 1

2.2.4 Distribution

Once optional alignments are defined in HPF, objects or aligned objects can be distributed on Cartesian processor arrangements, declared like arrays and templates, and represented by the inequality $0 \leq p < P.1$. Each dimension can be distributed by block, or in a cyclic way, on the processor set or even collapsed on the processors. Let us first consider the Examples 1 and 2 in Figures 2.3 and 2.5:

Example 1

In Example 1, the distributions of the templates creates a link between the templates elements t and the processors p, so that each processor owns some of the template elements. These links can be translated into linear formulae with the addition of variables. For the `block` distribution of template `T`, assuming the local offset ℓ within the size 20 block, $0 \leq \ell < 20$, then the formula simply is:

$$t = 20p + \ell$$

For a fixed processor p and the allowed range of offsets within a block ℓ, the formula gives the corresponding template elements that are mapped on that processor. There is no constant in the equality due to the assumption that template and processor dimensions start from 0. For the `cyclic` distribution of template `T'`, a cycle variable c counts the number of cycles over the processors for a given template element. Thus the formula is:

$$t' = 5c + p$$

The cycle number c generates an initial offset on the template for each cycle over the processors. Then the p translation associates the processor's owned template element for that cycle. The general `cyclic(`n`)` multi-dimensional case necessitates both cycle c and local offset ℓ variable vectors, and can be formulated with a matrix expression to deal with all dimensions at once.

General case

HPF allows a different block size for each dimension. The extents (n in `BLOCK(`n`)` or `CYCLIC(`n`)`) are stored in a diagonal matrix, C. P is a square matrix with the size of the processor dimensions on the diagonal. Such a distribution is not linear according to its definition [96] but may be written as a linear relation between the processor coordinate p, the template coordinate t and two additional variables, ℓ and c:

$$\Pi t = CPc + Cp + \ell \tag{2.3}$$

$c \downarrow$ / $l \rightarrow$	$p=0$				$p=1$				$p=2$				$p=3$			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

```
chpf$ processors P(0:3)
chpf$ template T(0:127)
chpf$ distribute T(cyclic(4)) onto P
```

Figure 2.4: Example of a template distribution (from [52]).

```
template T(0:99,0:99,0:99)
processors P(0:9,0:9)
distribute T(*,cyclic(4),block(13)) onto P
```

Figure 2.5: Example 2

Vector ℓ represents the offset within one block in one processor and vector c represents the number of wraparound (the sizes of which are CP) that must be performed to allocate blocks cyclically on processors as shown on Figure 2.4 for the example in Figure 2.7.

The projection matrix Π is needed when some dimensions are collapsed on processors, that means that all the elements on the dimension are on a same processor. These dimensions are thus orthogonal to the processor parallelism and can be eliminated. The usual modulo and integer division operators dealing with the block size are replaced by predefined constraints on p and additional constraints on ℓ. Vector c is implicitly constrained by array declarations and/or (depending on the replication) by the `TEMPLATE` declaration.

Specifying `BLOCK(`n`)` in a distribution instead of `CYCLIC(`n`)` is equivalent but tells the compiler that the distribution will not wrap around and the Equation (2.3) can be simplified by zeroing the c coordinate for this dimension. This additional equation reduces the number of free variables and, hence, removes a loop in the generated code. `CYCLIC` is equivalent to `CYCLIC(1)` which is translated into an additional equation: $\ell = 0$. `BLOCK` without parameter is equivalent to `BLOCK(`$\lceil \frac{e_t}{e_p} \rceil$`)` where e_t and e_p are the template and processor extents in the matching dimensions. Since block distributions do not cycle around the processors, $c = 0$ can be added.

The variables are bounded by:

$$0 \leq p < P.1, \quad 0 \leq t < T.1, \quad 0 \leq \ell < C.1 \tag{2.4}$$

Example 2

Let us consider the second, more general example, in Figure 2.5. These directives can be translated into the following matrix form:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} t = \begin{pmatrix} 40 & 0 \\ 0 & 130 \end{pmatrix} c + \begin{pmatrix} 4 & 0 \\ 0 & 13 \end{pmatrix} p + \ell, \quad \begin{pmatrix} 0 \\ 0 \end{pmatrix} \leq \ell < \begin{pmatrix} 4 \\ 13 \end{pmatrix}$$

where

$$t = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}, \quad p = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix}, \quad c = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}, \quad \ell = \begin{pmatrix} \ell_1 \\ \ell_2 \end{pmatrix}, \quad c_2 = 0$$

and

$$\Pi = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad C = \begin{pmatrix} 4 & 0 \\ 0 & 13 \end{pmatrix}, \quad P = \begin{pmatrix} 10 & 0 \\ 0 & 10 \end{pmatrix}$$

In this example, the distribution is of type `BLOCK` in the last dimension, thus the added $c_2 = 0$. If a dimension of a template is collapsed on the processors, the corresponding coordinate is useless because even if it is used to distribute the dimension of an array, this dimension is collapsed on the processors. Thus it can be discarded from the equations. It means that it can be assumed that $\Pi = I$ in Equation (2.3) if these useless dimensions are removed by a normalization process.

2.2.5 Iteration domains and subscript functions

Although iteration domains and subscript functions do not concern directly the placement and the distribution of the data in HPF, they must be considered for the code generation. Since loops are assumed `INDEPENDENT` with affine bounds, they can be represented by a set of inequalities on the iteration vectors i: The original enumeration order is not kept by this representation, but the independence of the loop means that the result of the computation does not depend on this very order. The array reference links the iteration vector to the array dimensions.

```
INDEPENDENT(i2,i1)
do j=1, n1, 3
  do i1=j+2, n2+1
    A(2*i1+1, n1-j+i1) = ...
  enddo
enddo
```

```
! normalized (stride 1) version
do i2=0, (n1-1)/3
  do i1=3*i2+3, n2+1
    A(2*i1+1, n1-3*i2+i1-1) = ...
  enddo
enddo
```

Figure 2.6: Loop example

Let us consider for instance the example in Figure 2.6. The iteration domain can be translated into a parametric (n_1 and n_2 may not be known at compile time) form, where the constant vector is a function over the parameters. Thus the set of constraints for the loop nest iterations, with the additional change of variable $j = 3i_2 + 1$ to normalize the loop [259], is:

$$\begin{pmatrix} 0 & -1 \\ 0 & 3 \\ 1 & 0 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \leq \begin{pmatrix} 0 \\ n_1 - 1 \\ n_2 + 1 \\ -3 \end{pmatrix}$$

```
      real A(0:42)
!HPF$ template T(0:127)
!HPF$ processors P(0:3)
!HPF$ align A(i) with T(3*i)
!HPF$ distribute T(cyclic(4)) onto P
      A(0:U:3) = ...
```

Figure 2.7: Example in Chatterjee *et al.* [52]

Moreover the array reference can also be translated into a set of linear equalities:

$$a = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} + \begin{pmatrix} 1 \\ n_1 - 1 \end{pmatrix}$$

In the general case, a set of constraints is derived for the iteration domain of the loop nest (Equation (2.5), where n stands for a vector of parameters for the loop nest) and for the array references (Equation (2.6)).

$$Li \leq b_0(n) \tag{2.5}$$

$$a = Si + a_0(n) \tag{2.6}$$

2.2.6 Putting it all together

Let us now consider the example in Figure 2.7. According to the previous Sections, all the declaration of distributions, alignments, arrays, processors and templates, iterations and references can be rewritten as

$$\begin{array}{rcl} 0 \leq & \ell & < C.1, \\ 0 \leq & a & < D.1, \\ 0 \leq & p & < P.1, \\ 0 \leq & t & < T.1, \\ Rt & = & Aa + s_0, \\ t & = & CPc + Cp + \ell, \\ Li & \leq & b_0(n), \\ a & = & Si + a_0(n) \end{array}$$

where

$$D = (43),\ T = (128),\ A = (3),\ s_0 = (0),\ R = (1),\ C = (4),\ P = (4),$$

$$L = \begin{pmatrix} 3 \\ -3 \end{pmatrix},\ b_0(n) = \begin{pmatrix} \mathtt{U} \\ 0 \end{pmatrix},\ S = (3),\ a_0(n) = (0),$$

The array mapping of `A` is represented by the "$\circ$" or "$\bullet$" on Figure 2.8 and the written elements (if $\mathtt{U} = 42$) of `A` with "$\bullet$". As can be seen on this 2D representation of a 3D space, the mapping and the access are done according to a regular pattern, namely a lattice. This is exploited to translate the compilation problem in a linear algebra framework.

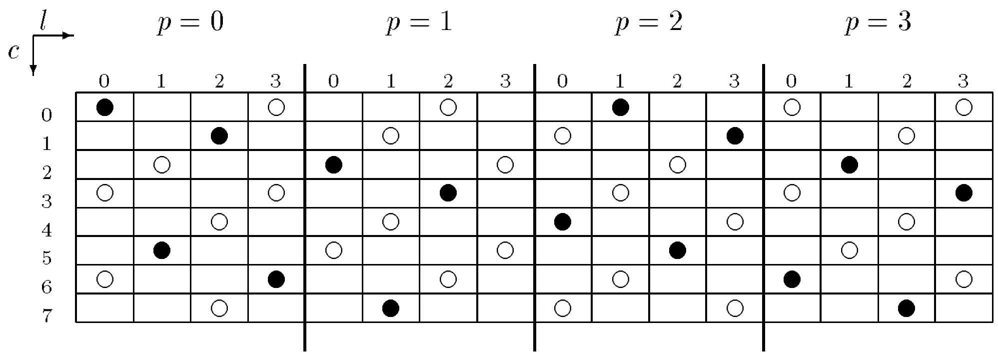


Figure 2.8: Example of an array mapping and writing (from [52]).

2.3 Overview of the compilation scheme

The compilation scheme is based on the HPF declarations, the *owner computes rule* and the SPMD paradigm, and deals with `INDEPENDENT` loops. Loop bound expressions and array subscript expressions are assumed affine. Such a parallel loop independently assigns elements of some left hand side `X` with some right hand side expression f on variables `Y`, `Z`, etc. (Figure 2.9). The loop is normalized to use unit steps as explained in the previous section. Since the loop is independent, $S_{\mathtt{X}}$ must be injective on the iteration domain for an independent parallel loop to be deterministic. The n models external runtime parameters that may be involved in some subexpressions, namely the loop bounds and the reference shifts.

The HPF standard does not specify the relationship between the HPF processors and the actual physical processors. Thus, efficiently compiling loop nests involving arrays distributed onto different processor sets without further specification is beyond the scope of this paper. But if the processor virtualization process can be represented by affine functions, which would be probably the case, we can encompass these different processor sets in our framework by remaining on the physical processors. In the following we assume for clarity that the interfering distributed arrays in a loop nest are distributed on the same processors set.

```
INDEPENDENT(i)
forall(Li ≤ b0(n))
    X(SX i + aX0(n)) = f(Y( SY i + aY0(n)), Z(SZ i + aZ0(n)))
```

Figure 2.9: Generic loop

2.3.1 Own Set

The subset of `X` that is allocated on processor p according to HPF directives must be mapped onto an array of the output program. Let $Own_{\mathtt{X}}(p)$ be this subset of `X` and `X'` the local array mapping this subset. Each element of $Own_{\mathtt{X}}(p)$ must be mapped to one element of `X'` but some elements of `X'` may not be used. Finding a good mapping involves a tradeoff between the memory space usage and the access function complexity. This is studied in Section 2.4.3. Subset $Own_{\mathtt{X}}(p)$ is derived from HPF declarations, expressed in an affine formalism (see Section 2.2), as:

$$\begin{aligned} Own_{\mathtt{X}}(p) = \{a \mid \exists t, \exists c, \exists \ell, \ s.t. \quad & R_{\mathtt{X}} t = A_{\mathtt{X}} a + s_{\mathtt{X}0} \\ & \wedge\ t = C_{\mathtt{X}} P c + C_{\mathtt{X}} p + \ell_{\mathtt{X}} \\ & \wedge\ 0 \leq a < D_{\mathtt{X}}.1 \\ & \wedge\ 0 \leq p < P.1 \\ & \wedge\ 0 \leq \ell < C_{\mathtt{X}}.1 \\ & \wedge\ 0 \leq t < T_{\mathtt{X}}.1 \} \end{aligned}$$

Subset $Own_{\mathtt{X}}(p)$ can be mapped onto a Fortran array by projection on c and ℓ, or on any equivalent set of variables, *i.e.* up to an injective mapping. Although Own is not a dense polyhedron as defined here with only variable a (the set may not represent contiguous integer points of array indexes, there may be holes), it can be seen as such in the higher dimensional (a, c, ℓ) space, that is with the addition for variables that enable skipping over the regular holes. Thus this view is used in the following, although our interest it just to represent the array elements a. Note that for a given distributed dimension a_i there is one and only one corresponding (p_j, c_j, ℓ_j) triple.

2.3.2 Compute set

Using the *owner computes rule*, the set of iterations local to p, $Compute(p)$, is directly derived from the previous set, $Own_{\mathtt{X}}(p)$. The iterations to be computed by a given processor are those of the loop nest for which the *lhs* are owned by the processor:

$$Compute(p) = \{i \mid S_{\mathtt{X}} i + a_{\mathtt{X}0}(n) \in Own_{\mathtt{X}}(p) \wedge Li \leq b_0(n)\}$$

Note that $Compute(p)$ and $Own_{\mathtt{X}}(p)$ are equivalent sets if the reference is direct ($S_{\mathtt{X}} = I$, $a_{\mathtt{X}0} = 0$) and if the iteration and array spaces conform. This is used in Section 2.4.3 to study the allocation of local arrays as a special case of local iterations. According to this definition of $Compute$, when `X` is replicated, its new values are computed on all processors having a copy. Depending on a tradeoff between communication and computation speed, the optimal choice may be to broadcast the data computed once in parallel rather than computing each value locally.

2.3.3 View set

The subset of `Y` (or `Z`,...) used by the loop that compute `X` is induced by the set of local iterations:

$$View_{\mathtt{Y}}(p) = \{a \mid \exists i \in Compute(p) \ s.t. \ \ a = S_{\mathtt{Y}} i + a_{\mathtt{Y}0}(n)\}$$

Note that unlike $S_{\mathtt{X}}$, matrix $S_{\mathtt{Y}}$ is not constrained and cannot always be inverted.

If the intersection of this set with $Own_{\mathtt{Y}}(p)$ is non-empty, some elements needed by processor p for the computation are fortunately on the same processor p. Then, in order to simplify the access to $View_{\mathtt{Y}}(p)$ without having to care about dealing differently with remote and local elements in the computation part, the local copy $\mathtt{Y}'$ may be enlarged so as to include its neighborhood, including $View_{\mathtt{Y}}(p)$. The neighborhood is usually considered not too large when it is bounded by a numerical *small* constant, which is typically the case if X and Y are identically aligned and accessed at least on one dimension up to a translation, such as encountered in numerical finite difference schemes. This optimization is known as *overlap analysis* [105]. Once remote values are copied into the overlapping Y′, all elements of $View_{\mathtt{Y}}(p)$ can be accessed uniformly in Y′ with no overhead. However such an allocation extension may be considered as very rough and expensive in some cases (*e.g.* a matrix multiplication), because of the induced memory consumption. A reusable temporary array might be preferred, and locally available array elements must be copied. Such tradeoffs to be considered in the decision process are not discussed in this paper, but present the techniques for implementing both solutions.

When Y (or X, or Z,...) is referenced many times in the input loop or in the input program, these references must be clustered according to their connection in the dependence graph [8]. Input dependencies are taken into account as well as usual ones (flow-, anti- and output-dependencies). If two references are independent, they access two distant area in the array and two different local copies should be allocated to reduce the total amount of memory allocated: a unique copy would be as large as the convex hull of these distant regions. If the two references are dependent, only one local copy should be allocated to avoid any consistency problem between copies. If Y is a distributed array, its local elements must be taken into account as a special reference and be accessed with (p, c, ℓ) instead of i.

The definition of *View* is thus altered to take into account array *regions*. These regions are the result of a precise program analysis which is presented in [240, 137, 15, 18, 17, 76, 75, 73]. An array region is a set of array elements described by equalities and inequalities defining a convex polyhedron. This polyhedron may be parameterized by program variables. Each array dimension is described by a variable. The equations due to subscript expression $S_{\mathtt{Y}}$ are replaced by the array region, a parametric polyhedral subset of Y which can be automatically computed. For instance, the set of references to Y performed in the loop body of:

```
do i2 = 1, N
   do i1 = 1, N
      X(i1,i2) = Y(i1,i2) + Y(i1-1,i2) + Y(i1,i2-1)
   enddo
enddo
```

is automatically summarized by the parametric region on $a_{\mathtt{Y}} = (y_1, y_2)$ for the Y reference in the loop body of the example above as:

$$\mathtt{Y}(y_1,y_2):\{(y_1,y_2) \mid y_1 \leq i_1 \ \wedge \ y_2 \leq i_2 \ \wedge \ i_1 + i_2 \leq y_1 + y_2 + 1\}$$

Subset $View_{\mathtt{Y}}$ is still polyhedral and array bounds can be derived by projection, up to a change of basis. If regions are not precise enough because convex hulls are used to summarize multiple references, it is possible to use additional parameters to exactly express a set of references with regular holes [7]. This might be useful for red-black SOR.

As mentioned before, if Y is a distributed array and its region includes local elements, it might be desired to simply extend the local allocation so as to simplify the addressing and to allow a uniform way of accessing these array elements, for a given processor p_Y and cycle c_Y. The required extension is computed simply by mixing the distribution and alignment *equations* to the $View_Y$ description. For the example above, assuming that `X` and `Y` are aligned and thus distributed the same way, the overlap is expressed on the block offset ℓ_Y with:

$$\begin{aligned} R_Y t &= A_Y a_Y + s_{Y0} \\ t &= CPc_Y + Cp_Y + \ell_Y \\ a_Y &\in View_Y(p) \end{aligned}$$

Vectors c_Y and p_Y are constrained as usual by the loop, the processor declaration and the template declaration of `X` but vector ℓ_Y is no more constrained in the block size (C). It is indirectly constrained by a_Y and the `Y` region. This leads to a $-1 \leq \ell_Y$ lower bound instead of the initial $0 \leq \ell_Y$, expressing the size 1 overlap on the left.

2.3.4 Send and Receive Sets

The set of elements that are owned by p and used by other processors and the set of elements that are used by p but owned by other processors are both intersections of previously defined sets:

$$\begin{aligned} Send_Y(p,p') &= Own_Y(p) \cap View_Y(p') \\ Receive_Y(p,p') &= View_Y(p) \cap Own_Y(p') \end{aligned}$$

These two sets cannot always be used to derive the data exchange code. Firstly, a processor p does not need to exchange data with itself. This cannot be expressed directly as an affine constraint and must be added as a guard in the output code[1]. Secondly, replicated arrays would lead to useless communication: each owner would try to send data to each viewer. An additional constraint should be added to restrict the sets of communications to needed ones. Different techniques can be used to address this issue: (1) replication allows broadcasts and/or load-balance, what is simply translated into linear constraints as described in [68]. (2) The assignment of owners to viewers can also be optimized in order to reduce the distance between communicating processors. For instance, the cost function could be the minimal Manhattan distance[2] between p and p' or the lexicographically minimal vector[3] $p' - p$ if the interconnection network is a $|p|$-dimension grid[4]. A combined cost function might even be better by taking advantage of the Manhattan distance to minimize the number of hops and of the lexicographical minimum to insure uniqueness. These problems can be cast as linear parametric problems and solved [86].

When no replication occurs, elementary data communication implied by $Send_Y$ and $Receive_Y$ can be parametrically enumerated in basis (p, u), where u is a basis

[1] This could be avoided by exchanging data first with processors p' such that $p' < p$ and then with processors such that $p' > p$, using the lexicographic order. But this additional optimization is not likely to decrease the execution time much, because the loop over p' is an outermost loop.

[2] The Manhattan norm of a vector is the sum of the absolute values of its coordinates, *i.e.* the l_1 norm.

[3] *i.e.* the closest on the first dimension, and then the closest on the second dimension, and so on, dimension per dimension, till one processor is determined.

[4] For a vector v, let $|v|$ denotes its dimension.

```
real X'((c,ℓ) ∈ Own_X(p)),
     Y'((c,ℓ) ∈ View_Y(p)),
     Z'((c,ℓ) ∈ View_Z(p))
forall(U ∈ {Y',Z',...})
       forall((p,p'), p ≠ p', Send_U(p,p') ≠ ∅ )
              forall((ℓ,c) ∈ Send_U(p,p'))
                     send(p', U(ℓ,c))
forall(U ∈ {Y',Z',...})
       forall((p,p'), p ≠ p', Receive_U(p,p') ≠ ∅ )
              forall((ℓ,c) ∈ Receive_U(p,p'))
                     U(ℓ,c) = receive(p')
if Compute(p) ≠ ∅
   forall((ℓ,c) ∈ Compute(p))
         X'(S_X'(ℓ,c)) = f(Y'(S_Y'(ℓ,c)),
                           Z'(S_Z'(ℓ,c)),...)
```

Figure 2.10: The output SPMD code.

for Y′, the local part of Y, the allocation and addressing of which are discussed in Section 2.4.3. *Send* and *Receive* are polyhedral sets and algorithms in [7] can be used. If the last component of u is allocated contiguously, vector messages can be generated.

2.3.5 Output SPMD code

The generic output SPMD code, parametric on the local processor p, is shown in Figure 2.10. U represents nay local array generated according to the dependence graph. Communications between processors can be executed in parallel when asynchronous send/receive calls and/or parallel hardware is available as in Transputer-based machines or in the PARAGON. The guards as $Send_{\mathtt{U}}$ can be omitted if unused HPF processors are not able to free physical processors for other useful tasks. The loop bound correctness ensures that no spurious iteration or communication occur. The parallel loop on U can be exchanged with the parallel inner loop on p' included in the `forall`. This other ordering makes message aggregation possible.

This code presented in Figure 2.10 is quite different from the one suggested in [164, 163]. Firstly, the set of local iterations $Compute(p)$ is no longer expressed in the i frame but in the (c,ℓ) frame: There is no *actual* need to know about the initial index values in the original code. Secondly, no difference is made between local and non-local iteration computations, at the price of possible communication and computation overlaps, but at the profit of an homogeneous addressing: Only one reference and addressing should be generated for each original reference in the computation expression, thus avoiding costly code duplications or runtime guards to deal with each reference of an expression that may or may not be locally available at every iteration. The temporary array management and addressing issues are discussed further in the next sections. Thirdly, non-local iterations computed as a set difference is not likely to produce an easily manageable set for code generation, since it should generally be non convex. Thirdly, the parametric description of the communications allows to enumerate a subset of active processors.

Additional changes of basis or changes of coordinates must be performed to reduce the space allocated in local memory and to generate a minimal number of more efficient scanning loops. Each change must exactly preserve integer points, whether they represent array elements or iterations. This is detailed in the next section.

2.4 Refinement

The pseudo-code shown in Figure 2.10 is still far from Fortran. Additional changes of coordinates are needed to generate proper Fortran declarations and loops.

2.4.1 Enumeration of iterations

A general method to enumerate local iterations is described below. It is based on (1) solving HPF and loop Equations, both equalities (2.2, 2.3, 2.6) and inequalities (2.4, 2.5), on (2) searching a *good* lattice basis to scan the local iterations in an appropriate order using p as parameter since each processor knows its own identity and (3) on using linear transformations to switch from the user visible frame to the local frame.

In this section, a change of frame is computed based on the available equalities to find a dense polyhedron that can be scanned efficiently with standard techniques. The change of frame computation uses two HERMITE forms to preserve the order of the (ℓ, c) variables which are related to the allocation scheme, for benefiting from cache effects.

Simplifying formalism

The subscript function S and loop bounds are affine. All object declarations are normalized with 0 as lower bound. The HPF array mapping is also normalized: $\Pi = I$. Under these assumptions and according to Section 2.2, HPF declarations and Fortran references are represented by the following set of equations:

$$\left.\begin{array}{rrcl} \text{alignment (2.2)} & Rt & = & Aa + s_0 \\ \text{distribution (2.3)} & t & = & CPc + Cp + \ell \\ \text{affine reference (2.6)} & a & = & Si + a_0(n) \end{array}\right\} \quad (2.7)$$

where p is the processor vector, ℓ the local offset vector, c the cycle vector, a the accessed element i the iteration and n the parameters.

Problem description

Let x be the following set of variables:

$$x = (\ell, c, a, t, i)^t \quad (2.8)$$

They are needed to describe array references and iterations. The order of the ℓ and c variables chosen here should be reflected by the local allocation in order to benefit from cache effects. This order suits best cyclic distributions, as discussed in [247], because there should be more cycles (along c) than block elements (along ℓ) in such cases. Putting the block offset after the cycle number for a given dimension would lead to larger inner loops for *more block* distributions. Note that the derivations discussed in this section are independent of this order. Equation (2.7) can be rewritten as

$$Fx = f_0(n, p) \tag{2.9}$$

with

$$F = \begin{pmatrix} 0 & 0 & A & -R & 0 \\ I & CP & 0 & -I & 0 \\ 0 & 0 & -I & 0 & S \end{pmatrix} \text{ and } f_0(n,p) = \begin{pmatrix} s_0 \\ -Cp \\ a_0(n) \end{pmatrix} \tag{2.10}$$

For instance in the Chatterjee *et al.* example in Figure 2.7, Equation 2.10 is:

$$\begin{pmatrix} 0 & 0 & 1 & -1 & 0 \\ 1 & 16 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 3 \end{pmatrix} \begin{pmatrix} \ell \\ c \\ a \\ t \\ i \end{pmatrix} = \begin{pmatrix} 0 \\ -4p \\ 0 \end{pmatrix}$$

The set of interest is the lattice $\mathcal{F} = \{x | Fx = f_0(n, p)\}$. The goal of this section is to find a parametric solution for each component of x. This allows each processor p to scan its part of $\mathcal{F}$ with minimal control overhead.

It is important to note that F is of full row-rank. The rows contain distinct variables that insure their independence one from the other: Firstly, the alignment equations are composed of independent equalities (they differ from a variables); This is also the case for the distribution and reference equations because of the ℓ and t variables. Secondly the equalities composing the distribution equations are the only to contain ℓ variables, thus they are independent from the alignment and reference. Thirdly, the alignment and reference equations are separated by the template variables thru R. Since all equalities are independent, F is a full row-rank. Note that HPF restrictions about alignment are not exploited, and need not be. Our technique can equally handle skewed alignments.

If additional equalities are taken into account, such as those arising from simple block ($c_i = 0$) or cyclic ($\ell_i = 0$) distributions, they can be used to remove the variables from the equation and do not actually change this property. Other degenerate cases may arise (for instance, there is only one processor on a given dimension ...), but they are not actually discussed here. Our general scheme can directly take advantage of such extra information to optimize the generated code by including it as additional equalities and inequalities.

Parametric solution of equations

Lattice $\mathcal{F}$ is implicitly defined but a parametric definition is needed to enumerate its elements for a given processor p. There are two kinds of parameters that must be set apart. First the constant unknown parameters n and local processor id p the value of which are known at runtime on each processor. Second, the parameters we are interested in, that have to be enumerated or instantiated on each processor to scan the integer solutions to the HPF equations, namely the variables in vector x.

An HERMITE form [225] of integer matrix F is used to find the parameters. This form associates to F (an $n \times m$ matrix with $m \geq n$) three matrices H, P and Q, such that $H = PFQ$. P is a permutation (a square $n \times n$ matrix), H an $n \times m$ lower triangular integer matrix and Q an $m \times m$ unimodular change of basis. Since F is of full row-rank, no permutation is needed: $P = I$ and $H = FQ$ (a). By definition, H is a lower triangular matrix, and thus can be decomposed as $H = (H_L\ 0)$, where H_L is an $n \times n$ integer triangular square matrix. We know that $|H_L| \in \{-1, 1\}$. Indeed, H_L is of full rank (as F) and the column combinations performed by the HERMITE form computation puts unit coefficients on H_L diagonal. This is insured since independent unit coefficients appear in each row of F. Thus H_L is an integer triangular unimodular matrix, and has an integral inverse.

Now we can use Q as a change of basis between new variables v and x, with $v = Q^{-1}x$ (b). Vector v can also be decomposed like H in two components: $v = (v_0, v')$, where $|v_0|$ is the rank of H. Using (a) and (b) Equation (2.9) can be rewritten as:

$$Fx = FQQ^{-1}x = Hv = (H_L 0)(v_0, v') = H_L v_0 = f_0(n, p)$$

v_0 is a parametric solution at the origin which depends of the runtime value of the n and p parameters. Thus we have $v_0(n, p) = H_L^{-1} f_0(n, p)$. By construction, H does not constrain v' and Q can also be decomposed like v as $Q = (Q_0\ F')$. Lattice $\mathcal{F}$ can be expressed in a parametric linear way:

$$x = Qv = (Q_0\ F')(v_0(n, p), v') = Q_0 v_0(n, p) + F'v'$$

and with $x_0(n, p) = Q_0 v_0(n, p) = Q_0 H_L^{-1} f_0(n, p)$:

$$\mathcal{F} = \{x | \exists v' \ \ s.t. \ \ x = x_0(n, p) + F'v'\} \tag{2.11}$$

We have switched from an implicit (2.9) description of a lattice on x to an explicit (2.11) one through F' and v'. Note that F' is of full column-rank.

Cache-friendly order

As mentionned earlier, the allocation is based somehow on the (ℓ, c) variables. The order used for the enumeration should reflect as much as possible the one used for the allocation, so as to benefit from memory locality. Equation (2.11) is a parametric definition of x . However the new variables v' are not necessarily ordered as the variables we are interested in. The aim of this paragraph is to reorder the variables as desired, by computing a new transformation based on the HERMITE form of F'. Let $H' = P'F'Q'$ be this form. Let Q' define a new basis:

$$\begin{aligned} u &= Q'^{-1}v' && (2.12) \\ x - x_0(n, p) &= F'v' = P'^{-1}H'Q'^{-1}v' = P'^{-1}H'u && (2.13) \end{aligned}$$

If $P' = I$, the new generating system of $\mathcal{F}$ is based on a triangular transformation between x and u (2.13). Since H' is a lower triangular integer matrix, the variables of u and of x simply correspond one to the other. Knowing this correspondance allows to order the variable u components to preserve locality of accesses. If $P' \neq I$, the variables are shuffled and some cache effect may be lost. However we have never encountered such an example, but have not succeeded in proving that it could not occur.

Code generation

Variable u defining x in lattice $\mathcal{F}$ and polyhedron $\mathcal{K}$ are easy to scan with `DO` loops. The constraints defining polyhedron $\mathcal{K}$ are coming from declarations, HPF directives and normalized loop bounds. They are:

$$0 \leq \ell < C.1, \quad 0 \leq t < T.1, \quad 0 \leq a < D.1, \quad Li \leq b_0(n)$$

Let $Kx \leq k_0(n)$ be these constraints on x. Using (2.13) the constraints on u can be written $K(x_0(n,p) + P'^{-1}H'u) \leq k_0(n)$, that is $K'u \leq k_0'(n,p)$, where $K' = KP'^{-a}H'$ and $k_0'(n,p) = k_0(n) - Kx_0(n,p)$.

Algorithms presented in [7] or others [86, 71, 148, 2, 49, 48, 173, 169, 149, 253] can be used to generate the loop nest enumerating the local iterations. When S is of rank $|a|$, optimal code is generated because no projections are required. Otherwise, the quality of the control overhead depends on the accuracy of integer projections [210] but the correctness does not.

Correctness

The correctness of this enumeration scheme stems from (1) the exact solution of integer equations using the HERMITE form to generate a parametric enumeration, from (2) the unimodularity of the transformation used to obtain a *triangular* enumeration, and from (3) the independent parallelism of the loop which allows any enumeration order.

2.4.2 Symbolic solution

The previous method can be applied in a symbolic way, if the dimensions are not coupled and thus can be dealt with independently, as array sections in [52, 117, 230]. Equations (2.7) then become for a given dimension:

$$\left.\begin{array}{rl} \text{alignment} & t = \alpha a + t_0 \\ \text{distribution} & t = \pi\gamma c + \gamma p + \ell \\ \text{affine reference} & a = \sigma i + a_0 \end{array}\right\} \qquad (2.14)$$

where π is the number of processors (a diagonal coefficient of P), and γ the block size (*i.e.* a diagonal coefficient in C). In order to simplify the symbolic solution, variables a and t are eliminated. The matrix form of the system is then $f_0 = (\alpha a_0 + t_0 - \gamma p)$, $x = (\ell, c, i)$ and $F = \begin{pmatrix} 1 & \pi\gamma & -\alpha\sigma \end{pmatrix}$.

The HERMITE form is $H = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} = PFQ$, with $P = I$ and:

$$Q = \begin{pmatrix} 1 & -\pi\gamma & \alpha\sigma \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{array}{l}
\forall p \in [0 \ldots \pi - 1] \\
\texttt{do } u_1 = \frac{\alpha a_0 + t_0 - \gamma p + \gamma + g - 2}{g}, \frac{\alpha a_0 + t_0 - \gamma p}{g} \\
\quad \texttt{do } u_2 = \frac{-\omega u_1 + \frac{\pi\gamma}{g} - 1}{\frac{\pi\gamma}{g}}, \frac{\beta - 1 - \omega u_1}{\frac{\pi\gamma}{g}} \\
\qquad x = H'u + x_0(p)
\end{array}$$

Figure 2.11: Generated code

Let g, μ and ω be such that g is $\gcd(\pi\gamma, \alpha\sigma)$ and $g = \pi\gamma\mu - \alpha\sigma\omega$ is the Bezout identity. The Hermite form H' of the two rightmost columns of Q noted F' ($H' = P'F'Q'$) is such that $x - x_0 = H'u$ with:

$$H' = \begin{pmatrix} -g & 0 \\ \mu & \frac{\alpha\sigma}{g} \\ \omega & \frac{\pi\gamma}{g} \end{pmatrix}, \; x_0 = \begin{pmatrix} \alpha a_0 + t_0 - \gamma p \\ 0 \\ 0 \end{pmatrix}, \; P' = I \text{ and } Q' = \begin{pmatrix} \mu & \frac{\alpha\sigma}{g} \\ \omega & \frac{\pi\gamma}{g} \end{pmatrix}$$

This links the two unconstrained variables u to the elements x of the original lattice $\mathcal{F}$. Variables a and t can be retrieved using Equations (2.14).

The translation of constraints on x to u gives a way to generate a loop nest to scan the polyhedron. Under the assumption $\alpha > 0$ and $\sigma > 0$, assuming that loop bounds are rectangular[5] and using the constraints in K:

$$0 \leq \ell < \gamma, \quad 0 \leq a < s, \quad 0 \leq i < \beta$$

the constraints on x (the a one is redundant if the code is correct) are:

$$\begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \ell \\ c \\ i \end{pmatrix} \leq \begin{pmatrix} 0 \\ \gamma - 1 \\ 0 \\ \beta - 1 \end{pmatrix}$$

and can be translated as constraints on u:

$$\begin{pmatrix} g & 0 \\ -g & 0 \\ -\omega & -\frac{\pi\gamma}{g} \\ \omega & \frac{\pi\gamma}{g} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \leq \begin{pmatrix} \alpha a_0 + t_0 - \gamma p \\ -\alpha a_0 - t_0 + \gamma p + \gamma - 1 \\ 0 \\ \beta - 1 \end{pmatrix}$$

The resulting generic SPMD code for an array section is shown in Figure 2.11. As expected, the loop nest is parameterized by p, the processor identity. Integer divisions with positive remainders are used in the loop bounds.

2.4.3 HPF array allocation

The previous two sections can also be used to allocate local parts of HPF distributed arrays. A loop nest referencing a whole array through an identity subscript function ($S = I, a_0 = 0$) serves as a basis for the allocation. The dense

[5] This assumption is, of course, not necessary for the general algorithm described in Section 2.4.1, but met by array sections.

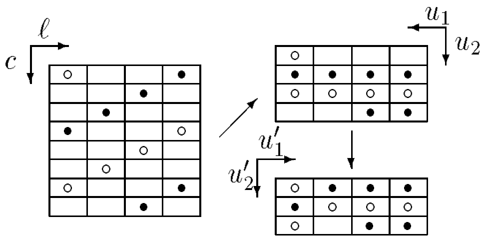


Figure 2.12: Packing of elements

polyhedron obtained by the changes of bases for the enumeration purpose can be used to store the required elements, since local iterations and local elements are strictly equivalent. Thus the constraints on local iterations can be reused as constraints on local elements. We identify simply the iteration space and array space to reuse our previous results developed to enumerate iterations for managing memory allocation and addressing of distributed arrays.

However, Fortran array declarations are based on Cartesian sets and are not as general as Fortran loop bounds. General methods have been proposed to allocate polyhedra [250]. For or particular case, it is possible to add another change of frame to fit Fortran array declaration constraints better and to reduce the amount of allocated memory at the expense of the access function complexity.

The local array elements are packed onto local memories dimension by dimension. The geometric intuition of the packing scheme for the [52] example is shown in Figure 2.12. The basic idea is to remove the regular holes due to the alignment stride by allocating the dense u space on each processor. The "$\circ$" and "$\bullet$" are just used to support the geometrical intuition of the change of frame. The first grid is the part of the template local to the processor, in the cycle and offset coordinate system. The second grid shows the same grid through transformation from x to u. The third one is the final packed form, which could be used if no space must be wasted, but at the expense of a complex access function.

An array dimension can be collapsed or distributed. If the dimension is collapsed no packing is needed, so the initial declaration is preserved. If the dimension is aligned, there are three corresponding coordinates (p, ℓ, c). For every processor p, local (ℓ, c) pairs have to be packed onto a smaller area. This is done by first packing up the elements along the columns, then by removing the empty ones. Of course, a dual method is to pack first along the rows, then removing the empty ones. This last method is less efficient for the example on Figure 2.12 since it would require 16 (8×2) elements instead of 12 (3×4). The two packing schemes can be chosen according to this criterion. Other issues of interest are the induced effects for the cache behavior and the enumeration costs. Formulae are derived below to perform these packing schemes.

Packing of the symbolic solution

Let us consider the result of the above symbolic solution, when the subscript expression is the identity $(\sigma = 1, a_0 = 0)$. The equation between u and the free variables of x is obtained by selecting the triangular part of H', i.e. its first rows.

If $H'' = (I\,0)H'$ is the selected sub-matrix, we have $x' - x'_0 = H''u$, *i.e.*:

$$\begin{pmatrix} \ell - t_0 + \gamma p \\ c \end{pmatrix} = \begin{pmatrix} -g & 0 \\ \mu & \frac{\alpha}{g} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$$

Variables σ and a_0 were substituted by their values in the initial definition of H'.

Variables (u_1, u_2) could be used to define an efficient packing, since holes are removed by the first change of basis (Figure 2.12). In order to match simply and closely the ℓ, c space, the sign of u_1 (linked to ℓ) can be changed, and the vector should be shifted so that the first local array element is mapped near $(0, 0)$.

Some space may be wasted at the beginning and end of the allocated space. For a contrived example (with very few cycles) the wasted space can represent an arbitrary amount on each dimension. Let us assume that two third of the space is wasted for a given dimension. Thus the memory actually used for an array with 3 of these dimensions is 1/27 and 26/27 of the allocated memory is wasted ... If such a case occurs, the allocation may be skewed to match a rectangle as closely as possible. This may be done if space limitation is at the main issue and if more complex, non-affine access functions are acceptable. The improved and more costly scheme is described in the next section.

Allocation basis

Let M be the positive diagonal integer matrix composed of the absolute value of the diagonal coefficients of H''.

$$u' = \mathrm{alloc}(u) = \lfloor M^{-1}(x' - x'_0) \rfloor = \lfloor M^{-1}H''u \rfloor \tag{2.15}$$

M provides the right parameters to perform the proposed packing scheme. To every u a vector u' is associated through Formula 2.15. This formula introduces an integer division. Let's show why u' is correct and induces a better mapping of array elements on local memories than u. Since H'' is lower triangular, Formula 2.15 can be rewritten:

$$\forall i \in [1 \ldots |u|], u'_i = \frac{h_{i,i}}{|h_{i,i}|}u_i + \frac{\sum_{j=1}^{i-1} h_{i,j}u_j}{|h_{i,i}|} \tag{2.16}$$

Function $\mathrm{alloc}(u)$ is bijective: $\mathrm{alloc}(u)$ is injective: if u^a and u^b are different vectors, and i is the first dimension for which they differ, Formula 2.16 shows that u'^a_i and u'^b_i will also differ. The function is also surjective, since the property allows to construct a vector that matches any u' by induction on i.

Array declaration

Two of the three components of x', namely p and ℓ, are explicitly bounded in K. Implicit bounds for the third component, c, are obtained by projecting K on c. These three pairs of bounds, divided by M, are used to declare the local part of the HPF array. Figure 2.13 shows the resulting declaration for the local part of the array, in the general symbolic case, for one dimension of the array. Bounds $\min(c)$ and $\max(c)$ are computed by FOURIER projection.

$$0 \leq p \leq \pi - 1, \;\; 0 \leq \ell \leq \gamma - 1$$
$$\left\lfloor \frac{t_0}{\gamma\pi} \right\rfloor \leq c \leq \left\lfloor \frac{\alpha(s-1)+t_0}{\gamma\pi} \right\rfloor$$

$$\texttt{array A'(0:}\left\lceil \frac{\max(c)-\min(c)+1}{\frac{\alpha}{g}} \right\rceil\texttt{,0:}\left\lceil \frac{\gamma}{g} \right\rceil\texttt{)}$$

Figure 2.13: Local new declaration

The packing scheme induced by u' is better than the one induced by u because there are no non-diagonal coefficients between u' and x' that would introduce a waste of space, and u' is as packed as u because contiguity is preserved by Formula 2.16. The row packing scheme would have been obtained by choosing (c, ℓ) instead of (ℓ, c) for x'. Different choices can be made for each dimension. The access function requires an integer division for the reduced memory allocation. Techniques have been suggested to handle divisions by invariant integers efficiently [111] that could help reduce this cost. Also, because contiguity is preserved, only one division per column is required to compute the base location (addresses can be managed incrementally). These packing schemes define two parameters (u'_1, u'_2) to map one element of one dimension of the HPF array to the processor's local part. The local array declaration can be linearized with the u'_3 dimension first, if the Fortran limit of 7 dimensions is exceeded.

2.4.4 Properties

The proposed iteration enumeration and packing scheme has several interesting properties. It is compatible with efficient cache exploitation and overlap analysis. Moreover, some improvements can statically enhance the generated code.

According to the access function, the iteration enumeration order and the packing scheme in Figure 2.11 can be reversed via loop u_2 direction in order that accesses to the local array are contiguous. Thus the local cache and/or prefetch mechanisms, if any, are efficiently used.

The packing scheme is also compatible with overlap analysis techniques [105]. Local array declarations are extended to provide space for border elements that are owned by neighbor processors, and to simplify accesses to non-local elements. The overlap is induced by relaxing constraints on ℓ, which is transformed through the scheme as relaxed constraints on u'_2. This allows overlaps to be simply considered by the scheme. Moreover a translated access in the original loop leading to an overlap is transformed into a translated access in the local SPMD generated code.

For example using the HPF array mapping of Figure 2.7:

```
align B with A
A(1:42) = B(0:41)
```

has a $View_{\texttt{B}}$ area represented in grey on Figure 2.14 in the unpacked template space and the local packed array space. The local array $\texttt{B}'$ can be extended by overlap to contain the grey area. Thus, constraint $0 \leq \ell \leq 3$ in *Own* becomes $-3 \leq \ell \leq 3$, expressing the size 3 overlap on the left.

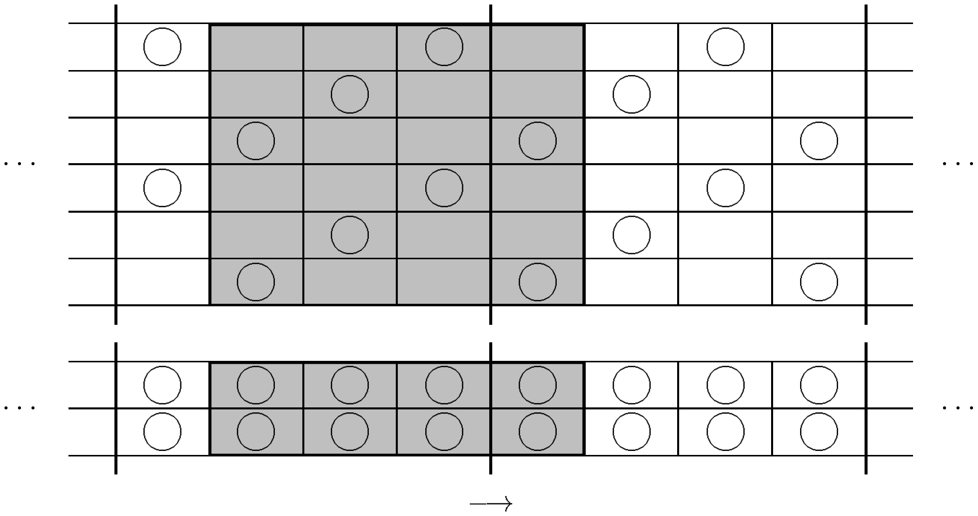


Figure 2.14: Overlap before and after packing.

$$\alpha = 3, t_0 = 0, \sigma = 3, a_0 = 0, \gamma = 4, \pi = 4, \beta = 15$$

$\forall p \in [0 \ldots 3]$
do u'_1 = 0, 3
    $lb_2 = \frac{16p+4u'_1+8}{9}, ub_2 = \frac{16p+4u'_1+7}{9}$
    $lb'_2 = \frac{9lb_2-4u'_1-16p}{3}$
    do u'_2 = lb'_2, $lb'_2 + (ub_2 - lb_2)$
        ...

Figure 2.15: Optimized code.

The generic code proposed in Figure 2.11 can be greatly improved in many cases. Integer division may be simplified, or performed efficiently with shifts, or even removed by strength reduction. Node splitting and loop invariant code motion should be used to reduce the control overhead. Constraints may also be simplified, for instance if the concerned elements just match a cycle. Moreover, it is possible to generate the loop nest directly on u', when u is not used in the loop body. For the main example in [52], such transformations produce the code shown in Figure 2.15.

In the general solution scheme (Section 2.4.1) the cycle variables c were put after the local offsets ℓ. The induced inner loop nest is then on c. It may be interesting to exchange ℓ and c in x' when the block size is larger than the number of cycles: the loop with the larger range would then be the inner one. This may be useful when elementary processors have some vector capability.

2.4.5 Allocation of temporaries

Temporary space must be allocated to hold non-local array elements accessed by local iterations. For each loop L and array X, this set is inferred from $Compute_L(p)$ and from subscript function $S_{\mathtt{X}}$. For instance, references to array Y in Figure 2.18 require local copies. Because of Y's distribution, overlap analysis is fine but another algorithm is necessary for other cases.

Let us consider the generic loop in Figure 2.9 and assume that local elements of Y cannot efficiently be stored using overlap analysis techniques. First of all, if all or most of local elements of the lhs reference are defined by the loop nest, and if $S_{\mathtt{Y}}$ has the same rank as $S_{\mathtt{X}}$, temporaries can be stored as X. If furthermore the access function $S_{\mathtt{X}}$ uses one and only one index in each dimension, the resulting access pattern is still HPF-like, so the result of Section 2.4.3 can be used to allocate the temporary array. Otherwise, another multi-stage change of coordinates is required to allocate a minimal area.

The general idea of the temporary allocation scheme is first to reduce the number of necessary dimensions for the temporary array via an HERMITE transformation, then to use this new basis for declaration, or the compressed form to reduce further the allocated space.

The set of local iterations, $Compute(p)$, is now defined by a new basis and new constraints, such that $x - x_0 = P'^{-1}H'u$ (2.13) and $K'u \leq k'$. Some rows of $P'^{-1}H'$ define iteration vector i, which is part of x (2.8). Let H'_i be this submatrix: $i = H'_i u$.

Let $H_{\mathtt{Y}} = P_{\mathtt{Y}} S_{\mathtt{Y}} Q_{\mathtt{Y}}$ be $S_{\mathtt{Y}}$'s HERMITE form. Let $v = Q_{\mathtt{Y}}^{-1} i$ be the new parameters, then $S_{\mathtt{Y}} i = P_{\mathtt{Y}}^{-1} H_{\mathtt{Y}} v$. Vector v can be decomposed as (v', v'') where $v' = \Pi v$ contributes to the computation of i and v'' belongs to $H_{\mathtt{Y}}$'s kernel. If $H_{\mathtt{Y}L}$ is a selection of the non-zero columns of $H_{\mathtt{Y}} = (H_{\mathtt{Y}L}\ 0)$, then we have:

$$\begin{aligned} a_{\mathtt{Y}} - a_{\mathtt{Y}0} &= S_{\mathtt{Y}} i = P_{\mathtt{Y}}^{-1} H_{\mathtt{Y}L}\, v' \\ v' &= \Pi Q_{\mathtt{Y}}^{-1}\, i \end{aligned}$$

and by substituting i:

$$v' = \Pi Q_{\mathtt{Y}}^{-1} H'_i u$$

input: $\{y = f.w, Ww \leq 0\}$
output: $\{y' = f'.w\}$

initial system: $f^0 = f$
for $i = 1 \ldots (|w| - 1)$
  $g_i = \gcd(\{f_j^{i-1}, j > i\})$
  $T_i = \{t | t = \sum_{j \leq i} f_j^{i-1} w_j \wedge W w_j \leq 0\}$
  $s_i = \max_{t \in T_i} t \; - \; \min_{t \in T_i} t$
  if $(s_i \geq g_i)$
    then $f^i = f^{i-1}$
    else $\forall j \leq i, f_j^i = f_j^{i-1}$ and $\forall j > i, f_j^i = \frac{f_j^{i-1}}{g_i} s_i$
end for
$f' = f^{|w|-1}$

Figure 2.16: Heuristic to reduce allocated space

It is sometime possible to use x' instead of v'. For instance, if the alignments and the subscript expressions are translations, i is an affine function of x' and v' simply depends on x'. The allocation algorithm is based on u but can also be applied when x' is used instead.

Since the local allocation does not depend on the processor identity, the first $|p|$ components of u should not appear in the access function. This is achieved by decomposing u as (u', u'') with $|u'| = |p|$ and $\Pi Q_{\Upsilon}^{-1} H_i'$ in (F_P, F_{Υ}) such that:

$$v' = F_P u' + F_{\Upsilon} u''$$

and the local part v'' is introduced as:

$$v'' = F_{\Upsilon} u''$$

Then the first solution is to compute the amount of memory to allocate by projecting constraints K onto v''. This is always correct but may lead to a waste of space because periodic patterns of accesses are not recognized and *holes* are allocated. In order to reduce the amount of allocated memory, a heuristic, based on large coefficients in F_{Υ} and constraints K, is suggested.

Each dimension, i.e. component of v'', is treated independently. Let F_{Υ}^i be a line of F_{Υ} and y the corresponding component of v'': $y = fv''$. A change of basis G (not related to the previous alloc operation) is applied to v'' to reduce f to a simpler equivalent linear form f' where each coefficient appears only once and where coefficients are sorted by increasing order. For instance:

$$f = \begin{pmatrix} 16 & 3 & -4 & -16 & 0 \end{pmatrix}$$

is replaced by:

$$\begin{aligned} f &= f'G \\ G &= \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 \end{pmatrix} \\ f' &= \begin{pmatrix} 3 & 4 & 16 \end{pmatrix} \end{aligned}$$

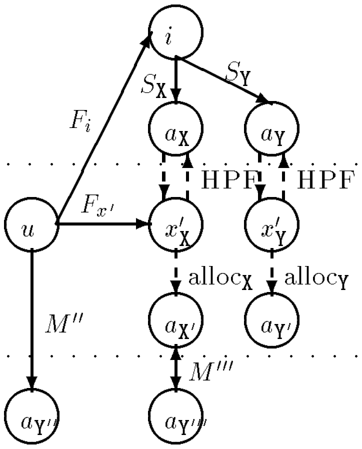


Figure 2.17: Relationships between frames

and v'' is replaced by $w = Gv''$. Constraints K on u are rewritten as constraints W on w by removing the constraints due to processors and by using projections and G.

Linear form f is then processed by the heuristic shown in Figure 2.16. It reduces the extent by $\frac{s_i}{g_i}$ at each step, if the constraints W show that there is a hole. A hole exists when g_i is larger than the extent of the partial linear form being built, under constraints W. Linearity of access to temporary elements is preserved.

This scheme is correct if and only if a unique location y' is associated to each y. Further insight on this problem, the minimal covering of a set by interval congruences, can be found in [110, 187].

2.4.6 Data movements

The relationships between the bases and frames defined in the previous sections are shown in Figure 2.17. Three areas are distinguished. The top one contains user level bases for iterations, i, and array elements, $a_{\mathtt{X}}$, $a_{\mathtt{Y}}$,... The middle area contains the bases and frames used by the compiler to enumerate local iterations, u, and to allocate local parts of HPF arrays, $a'_{\mathtt{X}}$, $a'_{\mathtt{Y}}$,... as well as the universal bases, $x'_{\mathtt{X}}$ and $x'_{\mathtt{Y}}$, used to define the lattices of interest, $\mathcal{F}$ and $\mathcal{F}'$. The bottom area shows new bases used to allocate temporary arrays, $a''_{\mathtt{Y}}$ and $a'''_{\mathtt{Y}}$, as defined in Section 2.4.5.

Solid arrows denote affine functions between different spaces and dashed arrows denote possibly non-affine functions built with integer divide and modulo operators. A quick look at these arrows shows that u, which was defined in Section 2.4.1 is the central basis of the scheme, because every other coordinates can be derived from u, along one or more paths.

Two kinds of data exchanges must be generated: updates of overlap areas and initializations of temporary copies. Overlap areas are easier to handle because the local parts are mapped in the same way on each processor, using the same alloc function. Let us consider array `X` in Figure 2.17. The send and receive statements are enumerated from the same polyhedron, up to a permutation of p and p', with the same basis u. To each u corresponds only one element in the user space, $a_{\mathtt{X}}$, by construction[6] of u. To each $a_{\mathtt{X}}$ corresponds only one $a'_{\mathtt{X}}$ on each processor[7]. As a result, data exchanges controlled by loop on u enumerate the same element at the same iteration.

Because the allocation scheme is the same on each processor, the inner loop may be transformed into a vector message if the corresponding dimension is contiguous and/or the send/receive library supports constant but non-unit strides. Block copies of larger areas also are possible when alloc is affine, which is not the general case from a theoretical point of view but should very often happen in real applications which should display simple inter-processor communication patterns. Such more optimizable cases can also appear when replicating data over processors. On the other hand, overlaps may create regularly distributed areas in the memory allocated for an array that is not to be communicated, preventing such optimizations.

Temporary arrays like $\mathtt{Y}''$ and $\mathtt{Y}'''$ are more difficult to initialize because there is no such identity between their allocation function as local part of an HPF array, $\text{alloc}_{\mathtt{Y}}$, and their allocation functions as temporaries, $\text{t\_alloc}_{\mathtt{Y}''}$ and $\text{t\_alloc}_{\mathtt{Y}'''}$. However, basis u can still be used.

When the temporary is allocated exactly like the master array, e.g. $\mathtt{Y}'''$ and $\mathtt{X}'$, any enumeration of elements u in $\mathcal{U}$ enumerates every element $a'''_{\mathtt{Y}}$ once and only once because the input loop is assumed independent. On the receiver side, $a'''_{\mathtt{Y}}$ is directly derived from u. On the sender side, $a_{\mathtt{Y}}$ is computed as $S_{\mathtt{Y}}F_i u$ and a non-affine function is used to obtain $x'_{\mathtt{Y}}$ and $a'_{\mathtt{Y}}$. Vector messages can be used when every function is affine, because a constant stride is transformed into another constant stride, and when such transfers are supported by the target machine. Otherwise, calls to packing and unpacking routines can be generated.

When the temporary is allocated with its own allocation function M, it is more difficult to find a set of u enumerating exactly once its elements. This is the case for copy $\mathtt{Y}''$ in Figure 2.17. Elements of $\mathtt{Y}''$, $a''_{\mathtt{Y}}$, must be used to compute one related u among many possible one. This can be achieved by using a pseudo-inverse of the access matrix M:

$$u = M^t(MM^t)^{-1}a''_{\mathtt{Y}} \tag{2.17}$$

Vector u is the rational solution of equation $a''_{\mathtt{Y}} = Mu$ which has a minimum norm. It may well not belong to $\mathcal{U}$, the polyhedron of meaningful u which are linked to a user iteration. However, M was built to have the same kernel as $S_{\mathtt{Y}}F_i$. Thus the same element $a_{\mathtt{Y}}$ is accessed as it would be by a regular u. Since only linear transformations are applied, these steps can be performed with integer computations by multiplying Equation (2.17) with the determinant of MM^t and by dividing the results by the same quantity. The local address, $a'_{\mathtt{Y}}$, is then derived as above. This proves that sends and receives can be enumerated by scanning elements $a''_{\mathtt{Y}}$.

[6]As explained in Section 2.4.3, allocation is handled as a special case by choosing the identity for $S_{\mathtt{X}}$.

[7]Elements in overlap area are allocated more than once, but on different processors.

```
      implicit integer (a-z)
      real X(0:24,0:18), Y(0:24,0:18)
!HPF$ template T(0:80,0:18)
!HPF$ align X(I,J) with T(3*I,J)
!HPF$ align Y with X
!HPF$ processors P(0:3,0:2)
!HPF$ distribute T(cyclic(4),block) onto P

      read *, m, n

!HPF$ independent(I,J)
      do I = m, 3*m
         do J = 0, 2*I
            X(2*I,J) = Y(2*I,J) + Y(2*I+1,J) + I - J
         enddo
      enddo

!HPF$ independent(I)
      do I = 0, n
         Y(I,I) = I
      enddo
```

Figure 2.18: Code example.

2.5 Examples

The different algorithms presented in the previous section were used to distribute the contrived piece of code of Figure 2.18, using functions of a linear algebra library developed at École des mines de Paris.

This is an extension of the example in [52] showing that allocation of HPF arrays may be non-trivial. The reference to `X` in the first loop requires an allocation of `X'` and the computation of new loop bounds. It is not a simple case because the subscript function is not the identity and because a `cyclic(4)` and `block` distribution is specified. The two references to `Y` in the first loop are similar but they imply some data exchange between processors and the allocation of an overlap area in `Y'`. Furthermore, the values of the `I` and `J` are used in the computation, the iteration domain is non rectangular and is parameterized with `m`.

The second loop shows that the *Compute*(p) set may have fewer dimensions than the array referenced and that fewer loops have to be generated.

The output code is too long to be printed here. Interesting excerpts are shown in figures 2.19 and 2.20 and commented below.

2.5.1 HPF Declarations

HPF declarations are already normalized. Templates are all used thus $\Pi = I$. Other key parameters are:

$$P = \begin{pmatrix} 4 & 0 \\ 0 & 3 \end{pmatrix},\ T = \begin{pmatrix} 80 & 0 \\ 0 & 18 \end{pmatrix},\ C = \begin{pmatrix} 4 & 0 \\ 0 & 7 \end{pmatrix}$$

$$A_X = A_Y = \begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix},\ R_X = R_Y = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ t_{0X} = t_{0Y} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

```
      implicit integer (a-z)
c     Local declaration of X and Y(0:42,0:42):
      real X_hpf(0:6, 0:1, 0:4), Y_hpf(0:6, 0:2, 0:4)

      read *, m, n

c     For each processor P(p1,p2)

c     Send view_Y to compute_X:
      do dest_p1 = 0, 3
       do dest_p2 = 0, 2
        if (dest_p1.ne.p1.or.dest_p2.ne.p2) then
         do u6 = max(-15, -3 - 4*p1), min(0, -4*p1)
          do u7 = max(0, divide(15 - 5*u6, 16)),
     &          min(6, divide(24 - 5*u6, 16))
           do u8 = 7*p2, min(20, 7*p2)
            do u9 = 0, 4
             do u11 = max(0, divide(16*u7 + 5*u6,2), m, (2 + 2*dest_p1 + 8*u9)/3),
     &             min(12, divide(16*u7 + 5*u6,2), 3*m, (2*dest_p1 + 8*u9)/3)
              do u12 = max(0, divide(2*u11 - u8, 21), divide(7*dest_p2 - u8, 21)),
     &              min(0, divide(20 + 7*dest_p2 - u8, 21))
               w3 = 3*u7 + u6
               w5 = (-u6 - 4*p1)/3
               w6 = u8 - 7*p2
               send(P(dest_p1,dest_p2), Y_hpf(w6,w5,w3))
              enddo
             enddo
            enddo
           enddo
          enddo
         enddo
        endif
       enddo
      enddo

c     receive_Y
c     ...

c     compute_X:
      do u4 = 0, 4
       do u6 = max(0, m, (2 + 2*p1 + 8*u4)/3),
     &      min(12, 3*m, (1 + 2*p1 + 8*u4)/3)
        do u7 = max(0, 7*p2), min(18, 2*u6, 6 + 7*p2)
         w3 = u4
         w5 = (6*u6 - 16*u4 - 4*p1)/3
         w6 = u7 - 7*p2
         i = u6
         j = u7
         X_hpf(w6, w5, w3) = Y_hpf(w6, w5, w3)
     &        + i - j + Y_hpf(w6, w5 + 1, w3)
        enddo
       enddo
      enddo
```

Figure 2.19: Output code (excerpts, part 1).

```
c      compute_Y:
       do u4 = 0, 3
        do u5 = max(0, 7*p2, (2 + 4*p1 + 16*u4)/3),
     &      min(18, 6 + 7*p2, n, (3 + 4*p1 + 16*u4)/3)
         w3 = u4
         w5 = (3*u5 - 16*u4 - 4*p1)/3
         w6 = u5 - 7*p2
         i = u5
         Y_hpf(w6, w5, w3) = i
        enddo
       enddo

       end

c      Integer divide with positive remainder. Assumes j>0.
       integer function divide(i,j)
       integer i,j
       if(i.ge.0) then
          divide = i/j
       else
          divide = -(-i+j-1)/j
       endif
       end
```

Figure 2.20: Output code (excerpts, part 2).

2.5.2 Allocation of X′

Array `X` is referenced only once, as a lhs. Thus its local part is directly derived from the set $Own_{\mathtt{X}}(p)$ using the algorithm described in section 2.4.3.

$$Own_{\mathtt{X}}(p) = \{ \begin{array}{l} (x_1, x_2) \mid t_1 = 3x_1,\ t_2 = x_2 \\ t_1 = 16c_1 + 4p_1 + \ell_1,\ t_2 = 21c_2 + 7p_2 + \ell_2 \\ 0 \leq x_1 \leq 24,\ 0 \leq x_2 \leq 18,\ 0 \leq p_1 \leq 3,\ 0 \leq p_2 \leq 2 \\ 0 \leq \ell_1 \leq 3,\ 0 \leq \ell_2 \leq 6,\ 0 \leq t_1 \leq 80,\ 0 \leq t_2 \leq 18 \quad \} \end{array}$$

The equations are parametrically solved and x is replaced by u in the constraints according to the following change of basis:

$$\begin{array}{llll} u_1 = p_1, & u_2 = p_2, & u_3 = c_1, & 3u_5 - 16u_3 - 4u_1 = \ell_1 \\ u_4 = c_2, & u_5 = x_1, & u_6 = x_2, & u_6 - 21u_4 - 7u_2 = \ell_2 \end{array}$$

Then, Frame u is replaced by Frame u' (represented with the `w` variables in the Fortran code) to avoid non-diagonal terms in the constraints. Frame u' is linked to the user basis x by:

$$u'_1 = p_1, \quad u'_2 = p_2, \quad u'_3 = c_1, \quad u'_4 = c_2, \quad 3u'_5 = \ell_1, \quad u'_6 = \ell_2$$

and constraints K are projected on each component of u' to derive the declaration bounds of `X`′:

$$\begin{array}{lll} 0 \leq u'_1 \leq 3, & 0 \leq u'_2 \leq 2, & 0 \leq u'_3 \leq 4 \\ 0 \leq u'_4 \leq 0, & 0 \leq u'_5 \leq 1, & 0 \leq u'_6 \leq 6 \end{array}$$

Coordinates u'_1 and u'_2 are the processor coordinates. The other coordinates, u'_3 to u'_6, are the local array coordinates. Note that since we have a block distribution in the second dimension, $u'_4 = 0$. The Fortran declaration of `X`′ is shown in Figure 2.19 as `X_hpf`. 10×7 elements are allocated while 7×7 would be enough. The efficiency factor may be better in the (p, ℓ, c) basis. It would also be better with less contrived alignments and subscript expressions and/or larger blocks.

2.5.3 Sending Y′

Before the first loop nest is executed, each processor (p_1, p_2) must send some elements of `Y` to its right neighbor (p_1+1, p_2), with a wrap-around for the rightmost processors, $(3, p_2)$. The pairs of communicating processors are not accurately represented by convex constraints and the `dest_p1` and `dest_p2` loop is not as tight as it could. This suggests that central and edge processors should be handled differently when overlap is observed. A linearized view of the processors could also help to fix this issue.

The bounds for the inner loops are complicated but they could mostly be evaluated iteratively if node splitting and loop invariant code motion are used. Also, the test on `u7` can be simplified as a modulo operation. Finally `w5` and `w6` can also be evaluated iteratively.

2.5.4 Local iterations for the second loop

As for the first loop, the second loop cannot be represented as an array section assignment, which makes it harder to handle than the first one. The set of local iteration is defined by:

$$
\begin{array}{ll}
Compute_{\mathtt{Y}}(p) = \{ & (y_1, y_2) \mid y_1 = i,\ y_2 = i,\ t_1 = 3y_1,\ t_2 = y_2 \\
& t_1 = 16c_1 + 4p_1 + \ell_1,\ t_2 = 21c_2 + 7p_2 + \ell_2 \\
& 0 \le i \le n,\ 0 \le y_1 \le 24,\ 0 \le y_2 \le 18,\ 0 \le p_1 \le 3,\ 0 \le p_2 \le 2 \\
& 0 \le \ell_1 \le 3,\ 0 \le \ell_2 \le 6,\ 0 \le t_1 \le 80,\ 0 \le t_2 \le 18 \quad \}
\end{array}
$$

The loop bounds were retrieved by projection on u. It is probably useless to generate a guard on the processor identity because non-involved processor have nothing to do and because this does not delay the other ones. The guard may as well be hidden in the inner loop bounds. Experiments are needed to find out the best approach.

Note that an extra-loop is generated. `Y` diagonal can be enumerated with only two loops and three are generated. This is due to the use of an imprecise projection algorithm but does not endanger correctness [7]. Further work is needed in this area.

Integer divide

One implementation of the integer divide is finally shown. The divider is assumed strictly positive, as is the case in all call sites. It necessary because Fortran remainder is not positive for negative numbers. It was added to insure the semantics of the output code. Nevertheless, if we can prove the dividend is positive, we can use the Fortran division.

2.6 Related work

Techniques to generate distributed code from sequential or parallel code using a uniform memory space have been extensively studied since 1988 [45, 191, 261]. Techniques and prototypes have been developed based on Fortran [105, 106, 131, 38, 190, 260, 39, 41], C [12, 174, 10, 171, 11, 172] or others languages [220, 221, 164, 179, 163].

The most obvious, most general and safest technique is called run-time resolution [45, 191, 220]. Each instruction is guarded by a condition which is only true for processors that must execute it. Each memory address is checked before it is referenced to decide whether the address is local and the reference is executed, whether it is remote, and a receive is executed, or whether it is remotely accessed and a send is executed. This rewriting scheme is easy to implement [56] but very inefficient at run-time because guards, tests, sends and receives are pure overhead. Moreover every processor has to execute the whole control flow of the program, and even for parallel loop, communications may sequentialize the program at run-time [45].

Many optimization techniques have been introduced to handle specific cases. GERNDT introduced overlap analysis in [105] for block distributions. When local array parts are allocated with the necessary overlap and when parallel loops are translated, the instruction guards can very often be moved in the loop bounds and the send/receive statements are globally executed before the local iterations, *i.e.* the loop nest with run-time resolution is distributed into two loop nests, one for communications and one for computations. The communication loops can be rearranged to generate vector messages.

TSENG [241] presents lots of additional techniques (message aggregation and coalescing, message and vector message pipelining, computation replication, collective communication ...). He assumes affine loop bounds and array subscripts to perform most optimizations. He only handles `block` and `cyclic(1)` distributions and the alignment coefficient must be 1.

Recent publications tackle any alignment and distribution but often restrict references to array sections. Each dimension is independent of the others as was assumed in Section 2.4.2.

In PAALVAST *et al.* [200, 247] a technique based on the resolution of the associated Diophantine equations is presented. Row- and column-wise allocation and addressing schemes are discussed. BENKNER *et al.* [25, 24] present similar techniques.

CHATTERJEE *et al.* [52] developed a finite state machine approach to enumerate local elements. No memory space is wasted and local array elements are ordered by Fortran lexicographic order exactly like user array elements. They are sequentially accessed by `while` loops executing the FSM's, which may be a problem if vector units are available. Moreover, accesses to an auxiliary data structure, the FSM transition map, add to the overhead. Note that the code generated in Figure 2.11 may be used to compute the FSM. In fact the lower iteration of the innermost loop is computed by the algorithm that builds the FSM. KENNEDY *et al.* [152, 153, 133, 130] and others [238] have suggested improvements to this technique, essentially to compute faster at run-time the automaton transition map. Also multi-dimensional cases need many transition maps to be handled.

Papers by STICHNOTH *et al.* [230, 228] on the one hand and GUPTA *et al.* [117, 119, 147] on the other hand present two similar methods based on closed forms for this problem. They use array sections but compute some of the coefficients at run-time. GUPTA *et al.* solve the block distribution case and use processor virtualization to handle cyclic distributions. Arrays are densely allocated as in [52] and the initial order is preserved but no formulae are given. STICHNOTH uses the dual method for array allocation as in [56], that is blocks are first compressed, and the cycle number is used as a second argument.

In [4, 8] polyhedron-based techniques are presented to generate transfer code for machines with a distributed memory. In [2, 245] advanced analyses are used as an input to a code generation phase for distributed memory machines. Polyhedron scanning techniques are used for generating the code. Two family of techniques have been suggested for that purpose. First, FOURIER elimination based techniques [136, 7, 148, 2, 173, 169, 149, 253], and second, parametric integer programming based methods [86, 71, 49, 48]. In [19], a two-fold HERMITE transformation is also used to remove modulo indexing from a loop nest. First, variables are added to explicit the modulo computation, then the HERMITE computations are used to regenerate simply new loop bounds. While the aim is different, the transformations are very similar to those presented here.

2.7 Conclusion

The generation of efficient SPMD code from an HPF program is not a simple task and, up to now, many attempts have provided partial solutions and many techniques. A translation of HPF directives in affine constraints, avoiding integer division and modulo, was presented in Section 2.2 to provide a unique and powerful framework for HPF optimizations. Homogeneous notations are used to succinctly represent user specifications and a normalization step is then applied to reduce the number of objects used in the compilation scheme overviewed in Section 2.3. New algorithms are presented to (1) enumerate local iterations, to (2) allocate local parts of distributed arrays, to (3) generate send and receive blocks and to (4) allocate temporaries implied by rhs references. They are based on changes of basis and enumeration schemes. It is shown in Section 2.4 that problem (2) can be cast as special case of problem (1) by using an identity subscript function but that constraints on array bounds are more difficult to efficiently satisfy than loop bounds. Problem (3) is an extension of problem (1): a unique reference to an array is replaced by a set of references and the equation used to express the reference is replaced by a set of inequalities. Problem (4) is an extension of problem (2). The set of elements to allocate is no longer the image of a rectangle but the image of an iteration set which can have any polyhedral shape. This shows that all these problems are closely linked.

Although the usual affine assumptions are made for loop bounds and subscript expressions, our compilation scheme simultaneously lifts several restrictions: Array references are not restricted to array sections. General HPF alignments and distributions are supported, and the same algorithms also generate efficient codes for classical block distributions, similar to the ones produced by classical techniques. Memory allocation is almost 100 % efficient on large blocks and performs quite well on small ones when strange alignments are used. We believe that this slight memory waste is more than compensated by the stride-1 vector load, store, send and receive which can be performed on the copies and which are necessary for machines including vector units. These contiguous accesses also perform well with a cache. The overlap analysis and allocation is integrated to the basic allocation scheme. Finally, all the work for generating the computation and communication code is performed at compile-time and no auxiliary data structures are used at run-time.

Our scheme can also be extended to cope with processor virtualization if the virtualization scheme is expressed with affine constraints. Such a scheme could reuse HPF distribution to map HPF processors on physical processors.

Many partial optimization techniques are integrated in our direct synthesis approach: message vectorization, and aggregation [131], overlap analysis [105]. A new storage management scheme is also proposed. Moreover other optimizations techniques may be applied to the generated code such as vectorization [259], loop invariant code motion [1] and software pipelining [101, 249].

This technique uses algorithms, directly or indirectly, that may be costly, such as Fourier elimination or the simplex algorithm, which have exponential worst-case behaviors. They are used for array region analysis, in the set manipulations and in the code generation for polyhedron scanning. However our experience with such algorithms is that they remain practical for our purpose: Polyhedron-based techniques are widely implemented in the PIPS framework [138] where hpfc, our prototype HPF compiler, is developped. Firstly, for a given loop nest, the number of equalities and inequalities is quite low, typically a dozen or less. Moreover these systems tend to be composed of independent subsystems on a dimension per dimension basis, resulting in a more efficient practical behavior. Secondly efficient and highly tuned versions of such algorithms are available, for instance in the Omega library. Thirdly, potentially less precise but faster program analysis [44, 20, 127] can also be used in place of the region analysis.

Polyhedron-based techniques are already implemented in hpfc, our prototype HPF compiler [59] to deal with I/O communications in a host/nodes model [61] and also to deal with dynamic remappings [68] (`realign` and `redistribute` directives). For instance, the code generation times for arbitrary remappings are in 0.1–2s range. Future work includes the implementation of our scheme in hpfc, experiments, extensions to optimize sequential loops, to overlap communication and computation, and to handle indirections.

Acknowledgments

We are thankful to Béatrice Creusillet for her many questions, Pierre Jouvelot for his careful reading and many suggestions, William Pugh for helpful comments and debugging, William (Jingling) Xue for the many remarks which helped improve the paper and to the referees for their precise and constructive comments.

Chapitre 3

Compilation of I/O Communications for HPF

Ce chapitre a fait l'objet d'une communication à Frontiers'95 [61].

Résumé

Les machines massivement parallèles sont le plus souvent des machines à mémoire repartie. Ce principe facilite l'extensibilité, au prix de la facilité de programmation. De nouveaux langages, comme HPF, ont été introduits pour résoudre ce problème. L'utilisateur conseille le compilateur sur le placement des données et les calculs parallèles au moyen de directives. Cet article est consacré à la compilation des communications liées aux entrées-sorties pour HPF. Les données doivent être collectées sur le processeur hôte, ou mises à jour à partir de l'hôte, de manière efficace, avec des messages vectorisés, quel que soit le placement des données. Le problème est résolu en utilisant des techniques standards de parcours de polyèdres. La génération de code dans les différents cas est traitée. Puis on montre comment améliorer la méthode et comment l'étendre au cas des entrées-sorties parallèles. Ce travail suggère de nouvelles directives HPF pour la compilation des entrées-sorties parallèles.

Abstract

The MIMD Distributed Memory architecture is the choice architecture for massively parallel machines. It insures scalability, but at the expense of programming ease. New languages such as HPF were introduced to solve this problem: the user advises the compiler about data distribution and parallel computations through directives. This paper focuses on the compilation of I/O communications for HPF. Data must be efficiently collected to or updated from I/O nodes with vectorized messages, for any possible mapping. The problem is solved using standard polyhedron scanning techniques. The code generation issues to handle the different cases are addressed. Then the method is improved and extended to parallel I/Os. This work suggests new HPF directives for parallel I/Os.

3.1 Introduction

The supercomputing community is more and more interested in massively parallel architectures. Only these machines will meet the performances required to solve problems such as the *Grand Challenges*. The choice architecture is the MIMD Distributed Memory architecture: a set of loosely coupled processors linked by a network. Such a design insures scalability, at the expense of programming ease. Since the programmer is not given a global name space, distributed data addressing and low level communications are to be dealt with explicitely. Non portable codes are produced at great expense. This problem must be solved to enlarge the potential market for these machines. The SVM (Shared Virtual Memory) approach [180, 29] puts the burden on the hardware and operating system, which have to simulate a shared memory. The HPF Forum chose to put it on the language and compiler technology [95], following early academic and commercial experiments [131, 55, 241, 260, 38, 41]. The user is given a way to advise the compiler about data distributions and parallel computations, through a set of directives added to Fortran 90. The Forum did not address parallel I/O since no concensus was found.

However MPP machines have parallel I/O capabilities [31] that have to be used efficiently to run real applications [204, 91, 182]. For instance, the TMC's CM5 or the Intel's Paragon have so called I/O nodes attached to their fast network. They are used in parallel by applications requiring high I/O throughput. For networks of workstations, files are usually centralized on a server and accessed through NFS. Such a system can be seen as a parallel machine using PVM-like libraries [103]. These systems are often programmed with a host-node model. To allow nodes to access disk data, one solution is to provide the machine with parallel I/O which ensure I/O operation coherency when many nodes share the same data.

Another solution is to rely on program analyses to determine the data needed on each node, and to generate the communications from one or several processes which directly perform the I/O. This paper presents a technique for such an approach, which may be extended to parallel I/Os as shown in Section 3.4.2. It focuses on the compilation of I/O communications for HPF: data must be collected to or updated from one or several I/O nodes, for any possible mapping. An array may be mapped onto all processors (*i.e.* it is shared), or distributed, or even partially replicated onto some, depending on the mapping directives. Vectorized messages must be generated between the nodes that have data to input or output and the I/O nodes.

```
      program triangle
      real A(30,30)
chpf$ template T(68,30)
chpf$ align A(i,j) with T(2*i,j)
chpf$ processors P(4,2)
chpf$ distribute T(block,cyclic(5)) onto P
      read *, m
      do 1 j=3, m
         do 1 i=3, m-j+1
 1          print *, A(i,j)
      end
```

Figure 3.1: Example `triangle`

Let us look at the example in Figure 3.1: a 2D array `A` is mapped onto a

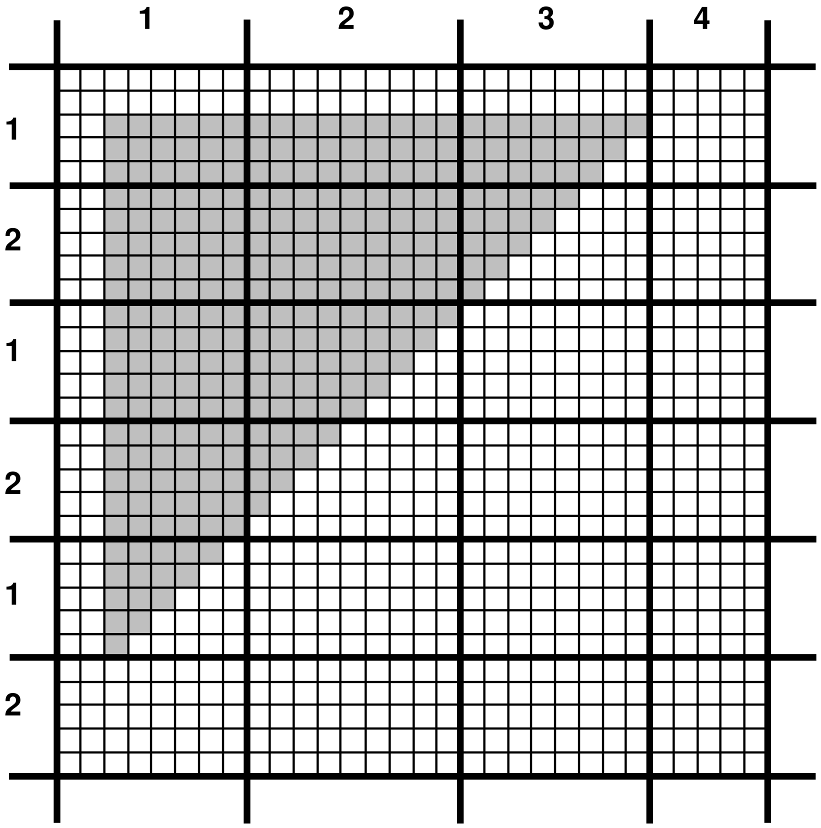


Figure 3.2: Accessed area of array `A` for $m = 27$

4×2 processor grid `P`. The I/O statement within the loop nest accesses the upper left of `A` using parameter `m` as shown in Figure 3.2. Several problems must be solved to compile this program: first, the loop nest must be considered as a whole to generate efficient messages. If the sole `print` instruction is considered, and since it outputs one array element at a time, there is no message vectorization opportunity. Second, the accessed array elements are only known at runtime (`m` dependence), so the method must be parametric. Third, the block distribution of the array first dimension is not regular, due to the stride 2 alignment and the odd size of the block: the array block size on the processor grid first dimension oscillates between 8 and 9 elements. Such issues must be managed by the solution. Finally, the generated code must deal with the array local addressing scheme chosen on the nodes to reduce the allocated memory in an efficient way, in order to avoid expensive address computations at runtime.

Section 3.2 gives the problem a mathematical formulation, and solves it using standard polyhedron scanning techniques. Section 3.3 describes the generated code to handle the different cases: collect or update of distributed or shared arrays. Section 3.4 discusses possible improvements and extensions of the technique to parallel I/O. This work suggests new HPF directives to help compile parallel I/O communications.

$$\texttt{A}(\alpha_1, \alpha_2)\texttt{-MUST-READ-} \{\alpha_1 + \alpha_2 \leq 1 + m, 3 \leq \alpha_1, 3 \leq \alpha_2\}$$

Figure 3.3: Region for the I/O loop nest

$$\begin{aligned}
\alpha'_1 &= ((2\alpha_1 - 1) \bmod 17) \div 2 + 1 \\
\alpha'_2 &= 5((\alpha_2 - 1) \div 10) + (\alpha_2 - 1) \bmod 5 + 1
\end{aligned}$$

Figure 3.4: Global $\texttt{A}(\alpha_1, \alpha_2)$ to local $\texttt{A}'(\alpha'_1, \alpha'_2)$

3.2 Problem expression

From an I/O statement, a distributed code must be generated. The first step is to formalize the problem in a mathematical way and to show how this problem is solved. This section addresses this point. The next step will be to show how the mathematical solution is embodied in a generated program, that is how to generate code from the mathematical solution. First, the basic analyses used are presented. Second, the communications required by an I/O statement are formulated in a linear framework. Third, the scheme for solving these equations is briefly outlined.

3.2.1 Analyses and assumptions

A precise description of the elements accessed by an I/O statement is needed. For each statement and each array accessed within this statement, a set of linear equalities and inequalities describing the region of accessed elements can be automatically derived from the program [239, 240, 15, 16]. A *region* is given an *action* and an *approximation*. The *action* is `READ` if the elements are read and `WRITE` if written. The *approximation* is `MUST` if all the elements are actually accessed and `MAY` if only a subset of these elements will probably be accessed. Figure 3.3 shows the *region* derived from the I/O loop nest of the running example. The α variables range over the accessed region on each dimension of the array. The linear inequalities select the upper left of the array, except borders.

Moreover the point where to insert the communications to achieve a possible message vectorization must be chosen. The running example may have been expressed with the implied-do syntax, however such loop nests can be dealt with. A simple bottom-up algorithm starting from the I/O instructions and walking through the abstract syntax tree of the program gives a possible level for the communications. For the running example, the I/O compilation technique will be applied to the whole loop nest.

3.2.2 Linear formulation

The declarations, the HPF directives and the local addressing scheme (Figure 3.4: each node allocates a 9×15 array `A'` to store its local part of distributed array `A`, and the displayed formulae allow to switch from global to local addresses) are translated into linear constraints, as suggested in [5]. Together with the *region*,

$$
\begin{array}{rl}
\texttt{A(}\alpha_1\texttt{,}\alpha_2\texttt{)} & 1 \le \alpha_1 \le 30, \quad 1 \le \alpha_2 \le 30 \\
\texttt{T(}\theta_1\texttt{,}\theta_2\texttt{)} & 1 \le \theta_1 \le 68, \quad 1 \le \theta_2 \le 30 \\
\texttt{P(}\psi_1\texttt{,}\psi_2\texttt{)} & 1 \le \psi_1 \le 4, \quad 1 \le \psi_2 \le 2 \\
\textit{region} & \left\{ \begin{array}{l} \alpha_1 + \alpha_2 \le 1 + m, \\ 3 \le \alpha_1, \quad 3 \le \alpha_2 \end{array} \right. \\
\texttt{align} & \theta_1 = 2\alpha_1, \quad \theta_2 = \alpha_2 \\
\texttt{distribute} & \left\{ \begin{array}{l} \theta_1 = 17\psi_1 + \delta_1 - 16, \\ 0 \le \delta_1 \le 16 \\ \theta_2 = 10\gamma_2 + 5\psi_2 + \delta_2 - 4, \\ 0 \le \delta_2 \le 4 \end{array} \right. \\
\texttt{A}'\texttt{(}\alpha'_1\texttt{,}\alpha'_2\texttt{)} & \left\{ \begin{array}{l} 1 \le \alpha'_1 \le 9, \quad 1 \le \alpha'_2 \le 15 \\ 2\alpha'_1 = \delta_1 - \eta_1 + 2, \quad 0 \le \eta_1 \le 1 \\ \alpha'_2 = 5\gamma_2 + \delta_2 + 1 \end{array} \right.
\end{array}
$$

Figure 3.5: Linear constraints for `triangle`

it gives a fully linear description of the communication problem, that is the enumeration of the elements to be sent and received. Figure 3.5 shows the linear constraints derived for the running example. The α variables (resp. θ, ψ) describe the dimensions of the array (resp. the template, the processors). The α' variables are used for the array local declaration. The γ and δ variables show the implicit linearity within the distribution: the δ's range within blocks, and the γ's over cycles. η_1 is an auxiliary variable.

The first three inequalities come from the declarations. The alignment constraints simply link the array (α) to the template (θ) variables, following the affine alignment subscript expressions. The distribution constraints introduce the block variables (δ), and the cyclic distribution on the second dimension requires a cycle variable (γ) which counts the cycles over the processors for a given template cell. Each distributed dimension is described by an equation that links the template and the processor dimensions together with the added variables. The local addressing scheme for the array is also translated into a set of linear constraints on α' variables. It enables the direct enumeration of the elements to exchange without expensive address computations at runtime. However what is really needed is the ability to switch from global to local addresses, so any other addressing scheme would fit into this technique. Moreover other addressing schemes [184] may be translated into a linear framework.

3.2.3 Solving the equations

The integer solutions to the previous set of equations must be enumerated to generate the communications. Any polyhedron scanning technique [86, 4, 7, 252, 71, 148, 48, 173, 169] can be used. The key issue is the control of the enumeration order to generate tight bounds and to reduce the control overhead. Here the word polyhedron denotes a set of constraints that defines a subspace of integer points. Different sets of constraints may define the same subspace, and some allow loop nests to be generated to scan the points. A *scannable* polyhedron for a given list of variables is such that its constraints are ordered in a triangular way which suits loop bound generation. One loop bound level must only depend on the outer

$$
\begin{array}{lll}
\mathcal{C} & (m) & 5 \leq m \\
\mathcal{P} \left\{ \begin{array}{l} \\ \\ \\ \end{array} \right. & \begin{array}{l} (\psi_1) \\ \\ (\psi_2) \end{array} & \begin{array}{l} 1 \leq \psi_1 \leq 4, \\ 17\psi_1 - 2m \leq 12 \\ 1 \leq \psi_2 \leq 2 \end{array} \\
\mathcal{E} \left\{ \begin{array}{l} \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \right. & \begin{array}{l} (\gamma_2) \\ (\alpha_2) \\ \\ (\alpha_1) \\ \\ \\ (\alpha'_1) \\ (\alpha'_2) \end{array} & \begin{array}{l} 1 \leq 2\gamma_2 + \psi_2 \leq 6 \\ 0 \leq 10\gamma_2 + 5\psi_2 - \alpha_2 \leq 4, \\ 3 \leq \alpha_2 \\ 0 \leq 17\psi_1 - 2\alpha_1 \leq 16, \\ 3 \leq \alpha_1 \leq 30, \\ \alpha_2 - m + \alpha_1 \leq 1 \\ 17 \leq 2\alpha'_1 - 2\alpha_1 + 17\psi_1 \leq 18 \\ \alpha'_2 = 5 + \alpha_2 - 5\psi_2 - 5\gamma_2 \end{array}
\end{array}
$$

Figure 3.6: Scannable polyhedra for `triangle`

levels, hence the triangular structure.

The technique used in the implementation is algorithm *row_echelon* [7]. It is a two-stage algorithm that takes as input a polyhedron and a list of variables, and generates a scannable polyhedron on these variables, the others being considered as parameters. The first stage builds the scannable polyhedron through successive projections, and the second stage improves the quality of the solution by removing redundant constraints while preserving the triangular structure. This algorithm generates a remainder polyhedron by separating the constraints that do not involve the scanning variables.

For the running example, this algorithm is applied twice. The first step addresses the actual element enumeration for a given processor. The processor identity is considered as a fixed parameter to enumerate the elements ($\mathcal{E}$ with γ_2, α and α' variables). The second step is applied on the remainder polyhedron of the first. It addresses the processor enumeration ($\mathcal{P}$ with ψ variables), to characterize the nodes that have a contribution in the I/O statement. The final remainder polyhedron ($\mathcal{C}$) is a condition to detect empty I/Os. The resulting polyhedra are shown in Figure 3.6: each line gives the variable and the linear constraints that define its lower and upper bounds. The algorithm insures that each variable has a lower and upper bounds which only depend on the outer variables and the parameters. Unnecessary variables (variables that are not needed to generate the communications) are eliminated when possible to simplify the loop nest and speed up the execution. θ, δ and η variables are removed, but γ_2 cannot. The order of the variables to enumerate the elements is an important issue which is discussed in Section 3.4.1.

3.3 Code generation

In the previous section, it has been shown how to formalize an HPF I/O statement into a linear framework, and how to solve the problem. The solution is based on standard polyhedron techniques. From a set of equalities and inequalities, it allows to build scannable polyhedra that suit loop bound generation. The method presented on the running example is very general, and any HPF I/O statement may be formalized and solved as described in Section 3.2. The only point which

	READ effect (print)	WRITE effect (read)
shared	host: I/O	host: I/O broadcast
distributed	collect host: I/O	if MAY collect host: I/O update

Figure 3.7: Generated code

is not yet addressed is the replication. In this section, the polyhedra generated in the very general case (replication included) are presented. Then the different code generation issues are discussed. First the macro code to handle the different cases is outlined. Second the particular problems linked to the actual generation of the communications are addressed, that is how data is to be collected to or to updated from the host.

3.3.1 General case polyhedra

In the general case, replication must be handled. Since several processors share the same view of the data, the same message will be sent on updates. For collects, only one processor will send the needed data. Thus four polyhedra are generated which define constraints on the variables between parentheses as described:

condition $\mathcal{C}$ (parameters): to guard empty I/Os.

primary owner $\mathcal{P}_1$ (ψ's): one of the owner processors for the array, which is arbitrarily chosen on the replication dimensions. The HPF replications occur regularly on specific dimensions of the processor grid. One processor is chosen by fixing the ψ variable on these dimensions. This subset describes the one processors that will send data on collects, if several are able to do it for some data, due to replication for instance.

other owners $\mathcal{P}_r$ (also ψ's): for a given primary owner in $\mathcal{P}_1$, all the processors which share the same view of the data. This is achieved by relaxing the previously fixed dimensions. This polyhedron synthetises the replication information. If there is no replication, then $\mathcal{P}_r = I$, thus the simpler $\mathcal{P}$ notation used in Figure 3.6 for $\mathcal{P}_1$. This polyhedron is used to send the message to all the processors which wait for the same data.

elements $\mathcal{E}$ (scanners): a processor identity (ψ) being considered as a parameter, it allows to enumerate the array elements of the I/O statement that are on this processor. The scanning variables are the variables introduced by the formalization, except the parameters and the ψ's.

HOST:
if $(\sigma \in \mathcal{C})$ – *non-empty I/O*
  – *get and unpack the messages*
  for $p_1 \in \mathcal{P}_1$
    receive from p_1
    – *enumerate the received elements*
    for $e \in \mathcal{E}(p_1)$
      unpack A(global(e))

SPMD node:
– *p is my id*
– *if non-empty I/O and I am in $\mathcal{P}_1$*
if $(\sigma \in \mathcal{C} \bigwedge p \in \mathcal{P}_1)$
  – *enumerate the elements to pack*
  for $e \in \mathcal{E}(p)$
    pack A′(local(e))
  send to host

Figure 3.8: Distributed collect

HOST:
if $(\sigma \in \mathcal{C})$ – *non-empty I/O*
  – *for each primary owner*
  for $p_1 \in \mathcal{P}_1$
    – *enumerate the elements to pack*
    for $e \in \mathcal{E}(p_1)$
      pack A(global(e))
    – *send the buffer to all owners*
    for $p \in \mathcal{P}_r(p_1)$
      send to p

SPMD node: – *p is my id*
– *if non-empty I/O and I am an owner*
if $(\sigma \in \mathcal{C} \bigwedge p \in \mathcal{P}_1 \times \mathcal{P}_r)$
  receive from host
  – *enumerate the elements to unpack*
  for $e \in \mathcal{E}(p)$
    unpack A′(local(e))

Figure 3.9: Distributed update

3.3.2 Generated code

The target programming model is a host-node architecture. The host processor runs a specific code and has to perform the I/O, and the nodes execute the computations. However the technique is not restricted to such a model, and the host processor could be one of the nodes with little adaptation. The generated codes for the distributed and shared arrays are shown in Figure 3.7. For the shared array case, it is assumed that each processor (host included) has its own copy of the array, so no communication is needed when the array is read. When it is written, the message is sent to all nodes: the accessed elements defined by the I/O statement are broadcast.

For the distributed array case, the whole array is allocated on the host, so the addressing on the host is the same as in the original code. However this copy is not kept coherent with the values on the nodes. It is used just as a temporary space to hold I/O data. The only issue is to move the needed values from the nodes to the host when the array is read, and to update the nodes from the host when the array is written. If the region's approximation is `MAY` and the array is defined within the statement (distributed row and `WRITE` effect column in Figure 3.7), a collect is performed prior to the I/O and the expected update. The rationale is that some of the described elements *may* not be defined by the statement, while the update generates a *full* copy of that region, hence overwritting carelessly the initial values stored on the nodes. This added collect insures that the elements that are not redefined by the I/O are updated latter on with their initial value.

```
// HOST
ARRAY A(30,30)
IF (m ≥ 5) THEN
 DO ψ1 = 1, min((12+2m)/17, 4)
 DO ψ2 = 1, 2
 pvmfrecv(P(ψ1, ψ2)...)
 DO γ2 = (2-ψ2)/2, (6-ψ2)/2
 DO α2 = max(-4 + 10γ2 + 5ψ2, 3),
 5ψ2 + 10γ2
 DO α1 = max((17ψ1-15)/2, 3),
 min(17ψ1/2, 30, -α2 + m + 1)
 DO α'1 = (18-17ψ1+2α1)/2, (18-17ψ1+2α1)/2
 α'2 = α2 - 5ψ2 + 5 - 5γ2
 pvmfunpack(A(α1, α2)...)
 ENDDOs
ENDIF
```

```
// SPMD node, (ψ1, ψ2) is my id in P
ARRAY A'(9,15)
IF (m ≥ 5 AND 17ψ1 ≤ 2m + 12) THEN
 pvmfinitsend(...)
 DO γ2 = (2-ψ2)/2, (6-ψ2)/2
 DO α2 = max(-4 + 10γ2 + 5ψ2, 3),
 5ψ2 + 10γ2
 DO α1 = max((17ψ1-15)/2, 3),
 min(17ψ1/2, 30, -α2 + m + 1)
 DO α'1 = (18-17ψ1+2α1)/2, (18-17ψ1+2α1)/2
 α'2 = α2 - 5ψ2 + 5 - 5γ2
 pvmfpack(A'(α'1, α'2)...)
 ENDDOs
 pvmfsend(host, ...)
ENDIF
```

Figure 3.10: Collect host/SPMD node codes

3.3.3 Scanning code

The next issue is the generation of the collects and updates, and how the polyhedra are used. For the host-node model, two codes are generated: one for the host, and one SPMD code for the nodes, parameterized by the processor identity. The two codes use the processor polyhedron ($\mathcal{P}$) differently. While the host scans all the nodes described by the polyhedron, the nodes check if they belong to the described set to decide whether they have to communicate with the host. On the host side, the polyhedron is used to enumerate the remote nodes, while on the node side it enables an SPMD code.

The distributed collect code is shown in Figure 3.8. First, each part checks whether an empty I/O is going to occur. If not, the host enumerates one of the owner of the data to be communicated through the $\mathcal{P}_1$ polyhedron. The messages are packed and sent from the contributing nodes, and symetrically received and unpacked on the host. The local and global functions used model the addressing scheme for a given point of the polyhedron which represents an array element. For the running example, it is a mere projection since both global (α) and local (α') indices are directly generated by the enumeration process ($\mathcal{E}$). The distributed update code is shown in Figure 3.9. The replication polyhedron $\mathcal{P}_r$ is used to send a copy of the same message to all the processors that share the same view of the array.

The correctness of the scheme requires that as many messages are sent as received, and that these messages are packed and unpacked in the same order. The balance of messages comes from the complementary conditions on the host and the nodes: the host scans the very node set that is used by the nodes to decide whether to communicate or not. The packing/unpacking synchronization is ensured since the same scanning loop ($\mathcal{E}$) is generated to enumerate the required elements on both sides. The fact that for the distributed update case the messages are packed for the primary owner and unpacked on all the owners is not a problem: the area to be enumerated is the same, so it cannot depend on the ψ variables of the replication dimensions. Thus the enumeration order is the same.

This technique is implemented in `hpfc`, a prototype HPF compiler [59, 56]. An excerpt of the automatically generated code for the running example is shown in Figure 3.10. The integer division with a positive remainder is used. The array declaration is statically reduced on the nodes. The SPMD code is parameterized by the processor identity (ψ_1, ψ_2) which is instanciated differently on each node. The inner loop nest to enumerate the elements is the same in both codes, as needed to ensure the packing/unpacking synchronization.

3.4 Extensions

The previous sections address the compilation of HPF I/O communications for a host-node model. From an I/O statement, it has been shown how to formalize and to solve the problem, then how to generate code from this mathematical solution. In this section, possible improvements are discussed first, then the technique is extended to handle parallel I/O communications.

3.4.1 Improvements

The implementation of the technique in `hpfc` already includes many basic optimizations. Let us look at the automatically generated code again. Non necessary variables (δ, η, θ) are removed from the original system through exact integer projections when legal, to reduce the polyhedron dimension. The condition checked by the node to decide whether it has to communicate is simplified: for instance constraints coming from the declarations on ψ's do not need to be checked. Moreover the variables used to scan the accessed elements are ordered so that the accesses are contiguous if possible in the memory, taking into account the row major allocation scheme of Fortran. Thus they are ordered by dimension first, then the cycle before the array variables, etc. However the generated code may still be improved.

First, Hermite transformations could have been systematically applied to minimize the polyhedron dimension, as suggested in [5]. The simplification scheme used in the implementation may keep unnecessary variables in some cases.

Second, the generated code would benefit from many standard optimizations such as strength reduction, invariant code motion, dag detection and constant propagation [1], which are performed by any classical compiler at no cost for `hpfc`. For instance, in Figure 3.10, a compiler may notice that α'_1 loop lower and upper bounds are equal, that α'_2 computation may be moved outside of the α_1 loop, or even that α'_1 and α'_2 computations are not necessary on the host.

Third, improving the analyses used by the technique would help enhance the generated code. For instance, the current implementation of the region analysis only computes a polyhedron around the accessed elements. Thus, it fails to detect accesses with a stride, and useless elements may be transmitted. Equalities may be added to the analysis to catch the underlying lattice which is lost in such cases. Another issue is to detect more `MUST` regions in general. Indeed, if the region approximation is `MAY` while all elements are accessed, then useless communications are generated when distributed arrays are defined.

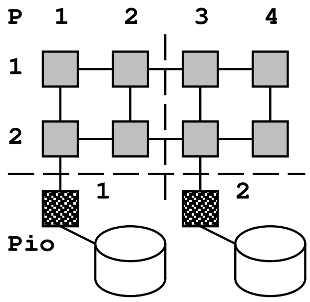


Figure 3.11: 2 I/O processors

$$\begin{array}{ll} \texttt{Pio}(\psi_{\texttt{io}}) & 1 \leq \psi_{\mathrm{io}} \leq 2 \\ \texttt{P}(\psi_1, \psi_2) & \left\{ \begin{array}{l} \psi_1 = 2\psi_{\mathrm{io}} - 1 + \delta_{\mathrm{io}}, \\ 0 \leq \delta_{\mathrm{io}} \leq 1 \end{array} \right. \end{array}$$

Figure 3.12: `Pio` to `P`

3.4.2 Parallel I/O

The technique was presented for a host/node architecture. It can be extended to parallel I/O. The basic idea of the method is to use standard polyhedron scanning techniques to generate vectorized messages. A linear description of the problem is needed. The reciprocal would be that if given a linear description of a parallel I/O problem, an efficient vectorized message passing code can be generated. So it is. Let us consider the running example again. Let us now assume that instead of a host/node architecture, we have 2 I/O processors which are regularily attached to the processor grid as depicted in Figure 3.11. This kind of arrangement is representative of I/O capabilities of some real world machines. However, there is no need for I/O nodes to be physically mapped as depicted. This regular mapping may only be a virtual one [72] which allows to attach groups of processors to I/O nodes.

Figure 3.12 suggests a possible linear formulation of the link between the I/O nodes and their attached processors. ψ_{io} ranges over the I/O nodes. An auxiliary dummy variable (δ_{io}) is used to scan the corresponding processors. This description looks like the one presented in Figure 3.5. The same scheme to enumerate the elements of interest (the contributing nodes, namely polyhedron $\mathcal{P}$) can be applied, with some external parameters (ψ_{io}, m) to be instantiated at runtime on the different nodes: the algorithm to build the scannable polyhedra is applied three times instead of two. The added level allows the I/O nodes ($\mathcal{P}_{\mathrm{io}}$) to scan their processors ($\mathcal{P}$), as the processors are allowed to scan their elements ($\mathcal{E}$).

The resulting scannable polyhedra are shown in Figure 3.13. They allow to generate the 2 SPMD codes of Figure 3.14. One is for the I/O nodes, and the other for the computation nodes. The inner loops are skipped, since they are the same as in Figure 3.10. The I/O nodes scan their attached nodes to collect the needed messages, and the computation nodes communicate with their attached I/O node instead of the host. To complete the code generation, a new addressing scheme

$$
\begin{array}{lll}
\mathcal{C} & (m) & 5 \leq m \\
\mathcal{P}_{\mathrm{io}} & \left\{ \begin{array}{l} (\psi_{\mathrm{io}}) \\ \\ \end{array} \right. & \begin{array}{l} 1 \leq \psi_{\mathrm{io}} \leq 2, \\ 17\psi_{\mathrm{io}} - m \leq 16 \end{array} \\
\mathcal{P} & \left\{ \begin{array}{l} (\psi_1) \\ \\ (\psi_2) \end{array} \right. & \begin{array}{l} 2\psi_{\mathrm{io}} - 1 \leq \psi_1 \leq 2\psi_{\mathrm{io}}, \\ 17\psi_1 - 2m \leq 12 \\ 1 \leq \psi_2 \leq 2 \end{array} \\
\mathcal{E} & \ldots &
\end{array}
$$

Figure 3.13: Parallel I/O polyhedra

```
// SPMD I/O node, ψio
// is my id in Pio
IF (m ≥ 5 AND 17ψio ≤ m + 16) THEN
 DO ψ1 = 2ψio − 1, min((12+2m)/17, 2ψio)
 DO ψ2 = 1, 2
 // receive from P(ψ1, ψ2)
 // and unpack
 ...
ENDIF
```

```
// SPMD node, (ψ1, ψ2) is my id in P
// ψio is my attached I/O node
IF (m ≥ 5 AND 17ψ1 ≤ 2m + 12) THEN
 // pack and send to Pio(ψio)
 ...
ENDIF
```

Figure 3.14: Parallel I/O collect

should be chosen on the I/O nodes, and a parallel file system should be used to respect the distributed data coherency.

This compilation scheme suggests new HPF directives to help compile parallel I/O statements. Indeed, what is needed is a linear link between the I/O nodes and the processing nodes. Moreover every computing node must be given a corresponding I/O node. The easiest way to achieve both linearity and total function is to rely on a distribution-like declaration, that is to distribute processors onto I/O nodes. A possible syntax is shown in Figure 3.15. It advises the compiler about the distribution of the I/Os onto the I/O nodes of the machine for data-parallel I/O. It is in the spirit of the hint approach that was investigated by the HPF Forum [94], that is to give some information to the compiler. However the record oriented file organization defined by the Fortran standard, which may also be mapped onto I/O nodes, is not directly addressed by this work, but such mappings may also be translated into linear constraints and compiled [5].

3.4.3 Related work

Other teams investigate the distributed memory multicomputers I/O issues, both on the language and runtime support point of view. Most works focus on the development of runtime libraries to handle parallel I/O in a convenient way for users. The suggested solutions focus more on general issues than on specific techniques to handle the required communications efficiently. They are designed for the SPMD programming model, and the allowed data distribution semantics is reduced with respect to the extended data mapping available in HPF.

In [30], a two-phase access strategy is advocated to handle parallel I/O efficiently. One phase performs the I/O, and the other redistribute the data as

```
chpf$ processors P(4,2)
chpf$ io_processors Pio(2)
chpf$ io_distribute P(block,*) onto Pio
```

Figure 3.15: Suggested syntax for parallel I/O

expected by the application. The technique presented in this paper would help the compilation of the communications involved by such a redistribution (between the I/O nodes and the computation nodes). This whatever the HPF mapping, as part of a dataparallel program to be compiled to a MIMD architecture.

In [100], the PETSc/Chameleon package is presented. It emphasizes portability and parallel I/O abstraction. [72] suggests to decluster explicitely the files, thus defining logical partitions to be dealt with separately. Their approach is investigated in the context of the Vesta file system. An interface is provided to help the user to perform parallel I/O. In [194], the PIOUS system is described. It is in the spirit of the client-server paradigm, and a database-like protocol insures the coherency of the concurrent accesses.

Moreover polyhedron scanning techniques have proven to be efficient and realistic methods for compilers to deal with general code transformations [252, 148] as well as distributed code generation [2, 171, 5]. In [2], a dataflow analysis is used to determine the communication sets. These sets are presented in a linear framework, which includes more parametrization and overlaps. The data mapping onto the processors is a superset of the HPF mapping. The local memory allocation scheme is very simplistic (no address translations) and cyclic distributions are handled through processor virtualization. [171] presents similar techniques in the context of the Pandore project, for commutative loop nests. The mapping semantics is a subset of the HPF mapping semantics. The Pandore local memory allocation is based on a page-like technique, managed by the compiler [173]. Both teh local addressing scheme and the cyclic distributions are integrated in [5] to compile HPF. Moreover equalities are used to improve the scanning loop nests, and temporary allocation issues are discussed.

3.5 Conclusion

Experiments were performed on a network of workstations and on a CM5 with the PVM3-based generated code for the host-node model. The performances are as good as what could have been expected on such systems. The control overhead to enumerate the required elements is not too high, especially for simple distributions. For network of workstations, the host should be the file server, otherwise the data must be transfered twice on the network.

We have presented a technique based on polyhedron scanning methods to compile parallel I/O communications for distributed memory multicomputers. This technique for the host-node architecture is implemented within hpfc, a prototype HPF compiler developed at CRI. This compiler is part of the PIPS automatic parallelizer [138]. From Fortran 77 code and static HPF mapping directives, it generates portable PVM 3-based code. It implements several optimizations such as message vectorization and overlap analysis on top of a runtime resolution compilation. Future work includes experiments, the implementation of advanced optimizations [5] and tests on real world codes.

Acknowledgements

I am thankful to Corinne Ancourt, Béatrice Creusillet, François Irigoin, Pierre Jouvelot, Kélita Le Nénaon, Alexis Platonoff and Xavier Redon for their comments and suggestions.

Chapitre 4

Compiling Dynamic Mappings with Array Copies

Ce chapitre est présenté en rapport interne EMP CRI A-292 [64].

Résumé

Les replacements de tableaux sont utiles à de nombreuses applications et sont intégrés dans le langage à parallélisme de données HPF qui vise les machines parallèles à mémoire répartie. Ce chapitre décrit des techniques pour gérer le placement dynamique avec des copies de tableaux : les replacements sont remplacés par des recopies entre tableaux placés statiquement. Il discute les restrictions de langage nécessaires pour appliquer une telle approche. Le graphe des replacements, qui capture l'information d'utilisation et les replacements nécessaires est présenté, ainsi que des optimisations appliquées sur ce graphe pour éviter des replacements inutiles et coûteux à l'exécution. De tels remplacements apparaissent quand un tableau n'est pas référencé après un replacement. De plus, une gestion dynamique des copies de tableaux dont les valeurs sont vivantes permet d'éviter encore d'autres cas de replacements. Finalement, la génération du code et les implications sur l'exécutif sont discutées. Ces techniques sont implantées dans notre prototype de compilateur HPF.

Abstract

Array remappings are useful to many applications on distributed memory parallel machines. They are integrated into High Performance Fortran, a Fortran 90-based data-parallel language. This paper describes techniques for handling dynamic mappings through simple array copies: array remappings are translated into copies between statically mapped distinct versions of the array. It discusses the language restrictions required to do so. The remapping graph which captures all remapping and liveness information is presented, as well as additional data-flow optimizations that can be performed on this graph, so as to avoid useless remappings at runtime. Such useless remappings appear for arrays that are not used after a remapping. Live array copies are also kept to avoid other flow-dependent useless remappings. Finally the code generation and runtime required by our scheme are discussed. These techniques are implemented in our prototype HPF compiler.

```
!hpf$ align with B:: A
!hpf$ distribute B(block,*)
      ...
! A is remapped
!hpf$ realign A(i,j) with B(j,i)
! A is remapped again
!hpf$ redistribute B(cyclic,*)
```

Figure 4.1: Possible direct `A` remapping

```
!hpf$ align with B:: C
!hpf$ distribute B(block,*)
      ...
! C is remapped
!hpf$ realign C(i,j) with B(j,i)
! C is remapped back to initial!
!hpf$ redistribute B(*,block)
```

Figure 4.2: useless `C` remappings

4.1 Introduction

Array remappings, *i.e.* the ability to change array mappings at runtime, are definitely useful to applications and kernels such as ADI [166], linear algebra solvers [209], 2D FFT [118] and signal processing [189] for efficient execution on distributed memory parallel computers. HPF [162] provides explicit remappings through `realign` and `redistribute` directives and implicit ones at subroutine calls and returns for array arguments. This paper discusses compiler handling of remappings and associated data flow optimizations.

4.1.1 Motivation

These features are costly at runtime because they imply communication. Moreover, even well written HPF programs may require useless remappings that can be avoided. In Figure 4.1 the change of both alignment and distribution of `A` requires two remappings while it could be remapped at once from `(block,*)` to `(*,cyclic)` rather than using the intermediate `(*,block)` mapping. In Figure 4.2 both `C` remappings are useless because the redistribution restores its initial mapping. In Figure 4.3 template `T` redistribution enforces the remapping of all five aligned arrays, although only two of them are used afterwards. In Figure 4.4 the consecutive calls to subroutine `foo` remap the argument on entry in and on exit from the routine, and both back and forth remappings could be avoided between the two calls; moreover, between calls to `foo` and `bla`, array `Y` is remapped from `(cyclic)` to `(block)` and then from `(block)` to `(cyclic(2))` while a direct remapping would be possible. All these examples do not arise from badly written programs, but from a normal use of HPF features. They demonstrate the need for compile-time data flow optimizations to avoid remappings at runtime.

4.1.2 Related work

Such optimizations, especially the interprocedural ones, are discussed in Hall *et al.* [121]. For handling subroutine calls, they show that the right approach is that the caller must comply to the callee requirements, that is the caller must insures that the arguments are properly mapped before the call. Our approach follows these lines. However, their analysis and optimization algorithms do not suit HPF for several technical reasons: first, they overlook the fact that in the general case several mappings can be associated at runtime to an array reference;

```
!hpf$ align with T :: &
!hpf$     A,B,C,D,E
!hpf$ distribute T(block)
      ... A B C D E ...
!hpf$ redistribute T(cyclic)
      ... A D ...
```

Figure 4.3: Aligned array remappings

```
      real Y(1000)
!hpf$ distribute Y(block)
      interface
        subroutine foo(X)
          real X(1000)
!hpf$ distribute X(cyclic)
        end subroutine
        subroutine bla(X)
          real X(1000)
!hpf$ distribute X(cyclic(2))
        end subroutine
      end interface
      ... Y ...
      call foo(Y)
      call foo(Y)
      call bla(Y)
      ... Y ...
```

Figure 4.4: Useless argument remappings

second, they do not address the practical handling of array references; third, the two-level mappings of HPF cannot be directly adapted to the simple algorithms described or suggested.

Let us focus on their work. *Reaching decompositions* are computed within procedures to describe possible mappings of array references:

> ([121] p. 525) To determine the decomposition of distributed arrays at each point in the program, the compiler calculates *reaching decompositions*. Locally, it is computed in the same manner as *reaching definitions*, with each decomposition treated as a “definition” [1].

Although it is indeed a dataflow problem, the computation of reaching mappings is not as simple as the reaching definitions problem because the compiler cannot directly know the mapping of arrays while looking at a remapping: whether an array is remapped or not may depend on the control flow, as shown in Figure 4.5(a); array `A` mapping is modified by the `redistribute` if the `realign` was executed before at runtime, otherwise `A` is aligned with template T_1 and get through T_2 redistribution unchanged; thus reaching mappings must be computed from the entry point of the program, propagating each initial mapping; however, unlike the reaching definition problem, new mappings (definitions) may appear in the graph as possible mappings reaching a remapping are changed by the directive.

Useless remappings are also removed by computing *live decompositions*:

> ([121] p. 529) We define *live decompositions* to be the set of data composition specifications that may reach some array reference aligned with the decomposition. [...] We calculate live decompositions by simply propagating uses backwards through the local control flow graph for each procedure [1].

```
!hpf$ template T1,T2
!hpf$ align with T1 :: A
      ... A ...
      if (...) then
!hpf$   realign with T2 :: A
        ... A ...
      endif
!hpf$ redistribute T2
      ... A ...
```

```
!hpf$ distribute A(block)
      ... A ...
      if (...) then
!hpf$   redistribute A(cyclic)
        ... A ...
      endif
      ...
      ! no reference to A
      ...
!hpf$ redistribute A(cyclic(10))
      ... A ...
```

Figure 4.5: Ambiguity of remappings (a) and (b)

The propagation is stopped by remappings that affects the given used array. However, here again, the ambiguity problem due to the two level mapping of HPF makes this fact not obvious to the compiler: the propagation should be stopped or not depending on the control flow followed by the program.

Finally optimizations are suggested for coalescing *live decompositions*, extracting loop-invariant mappings and using array kill analysis to avoid actually remapping dead arrays. Their approach assumes that a collection of library routines is available to remap data as needed.

In contrast to this approach, our scheme handles array references explicitely (by substituting the appropriate array copy), and thus enables more subsequent optimizations. We build a graph representation of the remappings ($\mathcal{G}_R$) which associates source to target copies, plus a precise use information (array may be read and/or written). We then repropagate live mappings to associate live source and target mapping couples after having removed dead remappings. This recomputation extracts the liveness analysis benefit and also performs the coalescing of live mappings in a single optimization on the small $\mathcal{G}_R$ graph. A second compile-time optimization allows to keep maybe-live array copies so as to avoid more useless remappings: the runtime management traces the status of live copies that may depend of the control flow. The drawback is the language restrictions needed for doing so in the general case. These issues are discussed in the following.

4.1.3 Outline

This paper focuses on compiling HPF remappings with array copies. Required language restrictions are discussed: basically, the compiler must statically know the mapping of array references, in order to apply this approach. Under a few constraints a program with dynamic mappings can be translated into a standard HPF program with copies between differently mapped arrays, as outlined in Figure 4.6: the redistribution of array A is translated into a copy from A_1 to A_2; the array references are updated to the appropriate array version.

By doing so, the benefit of remappings is limited to software engineering issues since it is equivalent to some static HPF program to be derived. It may also be argued that much expressiveness is lost by restricting the language. However it

```
!hpf$ distribute A(cyclic)
      ... A ...
!hpf$ redistribute A(block)
      ... A ...
```

```
      allocatable A1,A2
!hpf$ distribute A1(cyclic)
!hpf$ distribute A2(block)
      allocate A1
      ... A1 ...
! remapping
      allocate A2
      A2 = A1
      deallocate A1
! done
      ... A2 ...
```

Figure 4.6: Translation from dynamic to static mappings

must be noted that: (1) software engineering is an issue that deserves consideration; (2) the current status of the language definition is to drop remappings as a whole because they are considered too difficult to handle; (3) no real application we have encountered so far that would have benefit from remappings require the full expressiveness of arbitrary flow-dependent dynamic mappings. Thus it makes sense to suggest to keep the simple and interesting aspects of a feature and to delay further powerful extentions when both the need and compilation techniques are available.

Such an approach is also required if one wants to compile remappings rather than to rely on generic library functions. Indeed, for compiling a remapping into a message-passing SPMD code [68] both source and target mapping couples must be known. Then the compiler can take advantage of all available information to generate efficient code. The paper also addresses associated data flow optimizations enabled by such a scheme, by removing useless remappings and keeping track of live array copies.

Section 4.2 discusses the language restrictions and the handling of subroutine mappings. Section 4.3 describes the building of the remapping graph which extract remapping and liveness information from the control flow graph. Section 4.4 then presents data flow optimizations that can be performed on this graph. Finally Section 4.5 discusses runtime implications of our compilation and optimization techniques, before concluding.

4.2 Overview

This paper aims at compiling explicit and implicit HPF remappings by translating them into runtime array copies. We first discuss the language restrictions needed to apply this scheme. Then the subroutine argument mapping translation into explicit array copies is outlined.

4.2.1 Language restrictions

In order to be able to translate dynamic mappings into differently distributed array copies, enough information must be available to the compiler. Basically, the compiler must statically know about array mappings for each array reference. Thus the following constraints are needed:

1. dynamic mappings must be as constrained as static mappings;

   (thus they cannot depend on runtime parameters)

2. there must be no control flow mapping ambiguities for a reference;

   (however there may be such an ambiguity at a point in the program provided that the array is not referenced: in Figure 4.5(b) after the `endif` and before the final redistribution the compiler cannot know whether array `A` is distributed block or cyclic, but the mapping ambiguity is solved before any reference to `A`)

3. explicit mapping interfaces for subroutine arguments are required;

Constraint 1 insures that differently distributed array copies can be declared a static mapping. Thus array remappings cannot depend on runtime values as currently allowed by HPF. Constraint 2 is required so as to use only one possible new array for each reference. This constraint will be checked by the remapping graph construction algorithm which handles the substitution. Finally Constraint 3 must be met so that the caller knows about the callee requirements and is able to comply with them, as suggested in Section 4.2.2.

Under these constraints the compiler can switch all array references to different arrays depending on the array mapping at this point, and to insert array copies to update data when remappings are required. The expressiveness of array remappings is definitely reduced by these constraints, which restrict general dynamic array remappings to a static version. However it must be pointed out that no application or kernel we are aware of requires such expressiveness. Moreover although one can imagine some applications that would find some benefit from runtime dependent mappings, it is very hard to imagine codes that would both make sense from the application and performance point of view and that would present control flow-related mapping ambiguities.

4.2.2 Subroutine arguments handling

Subroutine arguments mappings will be handled as local remappings by the caller. Let us consider the different dummy arguments mappings the compiler may have to handle:

descriptive: when encountering a descriptive mapping for an array dummy argument at a call cite, the compiler will check that the current mapping of the actual array is compatible with this mapping.

prescriptive: the mapping is then enforced by the caller, by copying the array into an actual copy mapped as the corresponding dummy argument.

transcriptive: `inherit` tells that the callee can handle any mapping, thus the caller compilation is not constrained. However from the callee point of view handling inherited mapping array references is similar to handling remapping-related ambiguous references and is likely to be inefficient because all address computations must be delayed to run time.

Thus implicit remappings are translated into explicit ones at call site in the caller, provided that the dummy argument mapping is known to the caller through an appropriate explicit interface declaration. The intent attribute (`in`, `out` or `inout`) provides additionnal information about the effects of the call onto the array. It will be used to determine live copies.

Under the proposed scheme we plan to respect the intended semantics of HPF argument passing, that is the argument is the only information the callee obtain from the caller. Thus explicit remappings of arguments within the callee will only affect copies local to the subroutine. Under more advance calling conventions, it may be thought of passing live copies along the required copy, so as to avoid further useless remappings within the subroutine.

4.3 Remapping graph $\mathcal{G}_R$

This section defines and describes the construction of the remapping graph. This graph is a subgraph of the control flow graph which captures remapping information such as the source and target copies for each remapping of an array and how the array is used afterwards, that is a liveness information. Subsequent optimizations will be expressed on this small graph.

4.3.1 Definition and notation

In the following we will distinguish the abstract array and its possible instances with an associated mapping. Arrays are denoted by capital typewriter letters as $\mathtt{A}$. Mapped arrays are associated a subscript such as $\mathtt{A}_2$. Differently subscripted arrays refer to differently mapped instances. A zero-subscripted copy $\mathtt{A}_0$ designates the initial static mapping of the array. $C(\mathtt{A})$ denote the set of all possible mappings of abstract array $\mathtt{A}$. Through our compilation approach, each array and associated mapping will be given a new name, and copies between such arrays will model remappings.

If $\mathcal{G}$ is a graph then $\mathcal{V}(\mathcal{G})$ is its set of vertices and $\mathcal{E}(\mathcal{G})$ its set of edges. Successors of a vertex are designated by $\mathrm{succ}(v)$ and predecessors by $\mathrm{pred}(v)$.

Let us formally introduce the remapping graph $\mathcal{G}_R$. This graph is a subgraph of the control flow graph, and is expected to be much smaller in the general case. Our subsequent optimizations are expressed on this small graph instead of the control flow graph, avoiding many simple information propagations along paths.

vertices $\mathcal{V}(\mathcal{G}_R)$: the vertices are the remapping statements, whether explicit (`realign`, `redistribute`) or added to model implicit remappings at call sites; there is a subroutine entry point vertex v_0 and an exit point $v_{\mathtt{e}}$.

edges $\mathcal{E}(\mathcal{G}_R)$: an edge denotes a possible path in the control flow graph with the same array remapped at both vertices.

labels: to each vertex v in the remapping graph is associated $S(v)$, the set of remapped arrays. For each array $\mathtt{A} \in S(v)$ we have:

$L_{\mathtt{A}}(v)$: *one* (or none, noted $\perp$) leaving array copy, *i.e.* the copy which must be referenced after the remapping; note that in the general case HPF

```
!hpf$ distribute T(*,block)
!hpf$ align A(i,j) with T(i,j)
      if (...) then
!hpf$   realign A(i,j) with T(j,i)
      endif
!hpf$ redistribute T(block,*)
```

Figure 4.7: Several leaving mappings

$\mathtt{A}\ \{1,3\} \xrightarrow{\mathtt{R}} 2$

Figure 4.8: Label representation

allows several leaving mappings as depicted in Figure 4.7: array `A` is remapped at the `redistribute` to `(block,*)` or `(*,block)` depending on the execution of the `realign`.

We assume that no such cases occur to simplify our presentation.

$R_{\mathtt{A}}(v)$: the set of reaching copies for the array at this vertex.

In the general case with several leaving copies, distinct reaching copy sets must be associated to each possible leaving copy.

$U_{\mathtt{A}}(v)$: how the leaving copy might be used afterwards. It may be not referenced (`N`), fully redefined before any use (`D`), only read (`R`) or modified (`W`). The use information qualifiers supersede one another in the given order, *i.e.* once a qualifier is assigned it can only be updated to a stronger qualifier. The default value is `N`.

This provides a precise live information that will be used by the runtime and other optimizations to avoid remappings by detecting and keeping live copies. However it must be noted that this information is conservative, because abstracted at the high remapping graph level. The collected information can differ from the actual runtime effects on the subroutine: an array can be qualified as `W` from a point and not be actually modified.

Each edge is labelled with the arrays that are remapped from at the sink vertex when coming from the source vertex: $A(v, v')$. Note that

$$\mathtt{A} \in A(v, v') \Rightarrow \mathtt{A} \in S(v) \text{ and } \mathtt{A} \in S(v')$$

Figure 4.8 shows a label representation. Array `A` remapping links reaching copies $\{1, 3\}$ to the leaving mapping 2, the new copy being only read (`R`). The information means that the vertex is a remapping for array `A`. It may be reached with copies $\mathtt{A}_1$ and $\mathtt{A}_3$ and must be left with copy $\mathtt{A}_2$. As this copy will only be read, the compiler and runtime can keep the reaching copy values which are live, as they may be useful for instance if the array is to be remapped back to one of these distributions.

4.3.2 Construction algorithm

The remapping graph described above holds all the remapping and liveness information. The next issue is to build this graph. This section outlines the construction algorithm, which builds the remapping graph and updates the control graph to (1) switch array references to the appropriate copy, distributed as expressed by the program, (2) reflect implicit remappings of array arguments through explicit remappings and (3) check the conditions required for the correctness of our scheme.

The remapping graph construction algorithm starts by propagating the initial mapping copy of the array from the entry point of the subroutine. $\mathcal{G}_R$ construction algorithm pushes array versions along the control graph and extract a simpler graph to reflect the needed runtime copies to comply to the intended semantics of the program. Two added vertices (call $v_{\mathtt{c}}$ and exit $v_{\mathtt{e}}$) model the dummy arguments imported and/or exported values.

Here is such a data flow formulation of the construction algorithm. First, let us define the sets that will be computed by the dataflow algorithms in order to build $\mathcal{G}_R$:

REACHING(v): the set of arrays and associated mappings reaching vertex v; these arrays may be remapped at the vertex or left unchanged, thus going through the vertex.

LEAVING(v): the set of arrays and associated mappings leaving vertex v; one leaving mapping per array is assumed for simplifying the presentation.

REMAPPED(v): the set of arrays actually remapped at vertex v. (note that if several leaving array mappings are allowed, this information is associated to array and mapping couples instead of just considering arrays).

EFFECTSOF(v): the proper effect on distributed variables of vertex v, *i.e.* these variables and whether they are not referenced, fully redefined, defined or used; this is assume as an available information.

EFFECTSAFTER(v): the distributed variables and associated effects that may be encountered after v and before any remapping of these variables.

EFFECTSFROM(v): just the same, but including also the effects of v.

REMAPPEDAFTER(v): the distributed variables and associated remapping vertices that may be encountered directly (without intermediate remapping) after v.

REMAPPEDFROM(v): just the same, but including also v.

The following function computes the leaving mapping from a reaching mapping at a given vertex:

$\mathtt{A_j}$ = IMPACT($\mathtt{A_i}, v$): the resulting mapping of $\mathtt{A}$ after v when reached by $\mathtt{A_i}$; for all but remapping vertices, $\mathtt{A_i} = \mathtt{A_j}$, *i.e.* the mapping is not changed; realignments of $\mathtt{A}$ or redistributions of the template $\mathtt{A_i}$ is aligned with may give a new mapping. The impact of a call is null.

ARRAY($\mathtt{A_i}$)=$\mathtt{A}$: the function returns the array from and array and associated mapping.

operator $-$: to be understand as *but those concerning*, that is the operator is not necessarily used with sets of the same type.

intent	$U_{\mathtt{A}}(v_{\mathsf{c}})$	$U_{\mathtt{A}}(v_{\mathsf{e}})$
in	D	N
inout	D	W
out	N	W

Figure 4.9: Array argument use

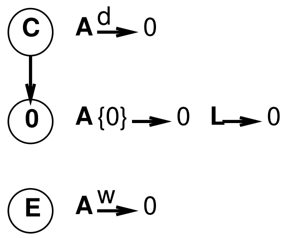


Figure 4.10: Initial $\mathcal{G}_R$

Now, here is the construction algorithm expressed as a set of data flow equations.

Input:

- control flow graph $\mathcal{G}_C$ with entry v_0 and exit v_{e} vertices
- the set of remapping vertices V_R, which includes v_0 and v_{e}.
- the proper effects of vertices on distributed variables EFFECTSOF(v) (the default for V_R is no effects)
- for any remapped array at a vertex, there is only one possible leaving mapping. This assumption simplifies the presentation, but could be removed by associating remapped information to array mappings instead of the array.

Updating $\mathcal{G}_C$ (arguments): first let us update $\mathcal{G}_C$ to model the desired mapping of arguments.

- Add call vertex v_{c} and an edge from v_{c} to v_0 in $\mathcal{G}_C$.

Reaching and Leaving mappings: are computed starting from the entry point in the program. Propagated mappings are modified by remapping statements as modeled by the IMPACT function, leading to new array versions to be propagated along $\mathcal{G}_C$. This propagation is a may forward dataflow problem.

initialization:

- REACHING $= \emptyset$
- LEAVING $= \emptyset$
- add all argument distributed variables and their associated mappings to LEAVING(v_{c}) and LEAVING(v_{e}).
- update EFFECTSOF(v_{c}) and EFFECTSOF(v_{e}) as suggested in Figure 4.9: If values are imported the array is annotated as defined before the entry point. If values are exported, it is annotated as used after exit. This models safely the caller context the callee must assume to comply to the intended semantics.
- add all local distributed variables and their associated initial mapping to LEAVING(v_0).

Figure 4.10 shows the initial remapping graph with an `inout` intended array argument `A` and a local array `L`.

$$\bigcirc\ \texttt{call foo(A)} \qquad \text{is translated into} \qquad \begin{array}{lcl} v_{\mathtt{b}} & \bigcirc & (\mathtt{A}\{i\} \xrightarrow{\mathtt{W}} k) \\ & \downarrow & \\ & \bigcirc & \texttt{call foo(}\mathtt{A}_{\mathtt{k}}\texttt{)} \\ & \downarrow & \\ v_{\mathtt{a}} & \bigcirc & (\mathtt{A}\{k\} \rightarrow i) \end{array}$$

Figure 4.11: Call with a prescriptive `inout`-intended argument

propagation:

- the array mappings reaching a vertex are those leaving its predecessors.
  $\textsc{Reaching}(v) = \bigcup_{v' \in \mathrm{pred}(v)} \textsc{Leaving}(v')$
- the array mappings leaving a vertex are updated with the impact of the statement on the array mappings reaching this vertex.
  $\textsc{Leaving}(v) = \textsc{Leaving}(v) \cup \bigcup_{a \in \textsc{Reaching}(v)} \textsc{impact}(a, v)$

Updating references for all vertices $v \in \mathcal{V}(\mathcal{G}_C) - V_R$ so that $\textsc{EffectsOf}(v)$ on is not `N`:

- if $|\{m \in \textsc{Leaving}(v), \textsc{array}(m) = \mathtt{A}\}| > 1$ then issues an error, because there is more than one mapping for a given array
- else substitute the references with the corresponding array copy.
- note that there may be none if some piece of code is dead.

Remapped arrays are directly extracted from REACHING; they are those transformed by IMPACT.

$\textsc{Remapped}(v) = \bigcup_{m \in \textsc{Reaching}(v), m \neq \textsc{impact}(m,v)} \textsc{array}(m)$

Updating $\mathcal{G}_C$ (calls):

- calls with distributed arguments are managed as shown in Figure 4.11:
  $\mathrm{pred}(v_{\mathtt{b}}) = \mathrm{pred}(v), \mathrm{succ}(v_{\mathtt{b}}) = \{v\}, \mathrm{pred}(v_{\mathtt{a}}) = \{v\}, \mathrm{succ}(v_{\mathtt{a}}) = \mathrm{succ}(v), \mathrm{pred}(v) = \{v_{\mathtt{b}}\}, \mathrm{succ}(v) = \{v_{\mathtt{a}}\}$
  $\textsc{Remapped}(v_{\mathtt{b}}) = \{\mathtt{A}\}$
- V_R is updated accordingly: $V_R = V_R \cup \{v_{\mathtt{b}}, v_{\mathtt{a}}\}$

Summarizing effects: this phase summarizes the use information after remapping statements, and up to any other remapping statement. Hence it captures what may be done with the array mapping considered.

This phase is based proper effects that are directly extracted from the source code for direct references, or through intent declarations in subroutine explicit interfaces. Depending on the intent attribute to a subroutine argument the corresponding effect is described in Figure 4.12.

Remapping statements but $v_{\mathtt{c}}$ and $v_{\mathtt{e}}$ have no proper effects:

$$\forall v \in V_R - \{v_{\mathtt{c}}, v_{\mathtt{e}}\}, \textsc{EffectsOf}(v) = \emptyset$$

intent	effect
`in`	`R`
`inout`	`W`
`out`	`D`

Figure 4.12: Intent effect

This is a may backwards dataflow problem.

initialization: no effects!

- $\textsc{EffectsAfter} = \emptyset$
- $\textsc{EffectsFrom} = \emptyset$

propagation:

- the effects leaving a vertex are those from its successors.
  $\textsc{EffectsAfter}(v) = \bigcup_{v' \in \mathrm{succ}(v)} \textsc{EffectsFrom}(v')$
- the effects from a vertex are those leaving the vertex and proper to the vertex, but remapped arrays.
  $\textsc{EffectsFrom}(v) = (\textsc{EffectsAfter}(v) \cup \textsc{EffectsOf}(v)) - \textsc{Remapped}(v)$

Computing $\mathcal{G}_R$ edges: as we expect few remappings to appear within a typical subroutine, we designed the remapping graph over the control graph with accelerating edges that will be used to propagate remapping information and optimization quickly. This phase propagates for once remapping statements (array and vertex couples) so that each remapping statement will know it possible successors for a given array.

This is a may backwards dataflow problem.

initialization:

- $\textsc{RemappedAfter} = \emptyset$
- initial mapping vertex couples are defined for remapping statement vertices and arrays remapped at this very vertex.
  $\textsc{RemappedFrom}(v) = \bigcup_{a \in \textsc{Remapped}(v)} \{(a, v)\}$

propagation:

- the remapping statements after a vertex are those from its successors.
  $\textsc{RemappedAfter}(v) = \bigcup_{v' \in \mathrm{succ}(v)} \textsc{RemappedFrom}(v')$
- the remapping statements from a vertex are updated with those after the vertex, but those actually remapped at the vertex.
  $\textsc{RemappedFrom}(v) = \textsc{RemappedFrom}(v) \cup (\textsc{RemappedAfter}(v) - \textsc{Remapped}(v))$

Generating $\mathcal{G}_R$: from these sets we can derive the remapping graph:

- V_R are $\mathcal{G}_R$ vertices
- edges and labels are deduced from $\textsc{RemappedAfter}$

- $S()$, $R()$ and $L()$ from REMAPPED, REACHING and LEAVING
- $U()$ from EFFECTSAFTER

All the computations are simple standard data flow problems, but the reaching and leaving mapping propagation: indeed, the IMPACT function may create new array mappings to be propagated from the vertex. The worst case complexity of the propagation and remapping graph algorithm describe above can be computed. Let us denote n is the number of vertices in $\mathcal{G}_C$, s the maximum number of predecessors or successors of a vertex in $\mathcal{G}_C$, m the number of remapping statements (including the entry and exit points), p the number of distributed arrays. With the simplifying assumption that only one mapping may leave a remapping vertex, then the maximum number of mappings to propagate is mp. Each of these may have to be propagated through at most n vertices with a smp worst case complexity for a basic implementation of the union operations. Thus we can bound the worst case complexity of the propagation to $\mathcal{O}(nsm^2p^2)$.

4.3.3 Example

Let us focus on the example in Figure 4.13. The small subroutine contains four remappings, thus with the added call, entry and exit vertices there are seven vertices in the corresponding remapping graph. There are three arrays, two of which are local and the last one which is passed as an argument to the subroutine. The sequential loop structure with two remappings is typical of ADI. The ... notation stands for any piece of code with the specified effects on the variables.

Figure 4.14 shows the resulting remapping graph and the associated liveness information for each array represented above the arrow. The rationale for the 1 to E and 2 to E edges is that the loop nest may have no iteration at runtime, thus the remappings within the array may be skipped. Since all arrays are aligned together, they are all affected by the remapping statements. Four different versions of each array might be needed with respect to the required different mapping. However, the liveness analysis shows that some instances are never referenced such as $\texttt{B}_3$ and $\texttt{C}_1$.

4.4 Data flow optimizations

The remapping graph $\mathcal{G}_R$ constructed above abstracts all the liveness and remapping information extracted from the control flow graph and the required dynamic mapping specifications. Following [121] we plan to exploit as much as possible this information to remove useless remappings that can be detected at compile time, or even some that may occur under particular runtime conditions. These optimizations on $\mathcal{G}_R$ are expressed as standard data flow problems [157, 150, 1].

4.4.1 Removing useless remappings

Leaving copies that are not live appear in $\mathcal{G}_R$ with the `N` (not used) label. It means that although some remapping on an array was required by the user, this array is not referenced afterwards in its new mapping. Thus the copy update is not needed and can be skipped. However, by doing so, the set of copies that may

```
  subroutine remap(A,m)                    → C
    parameter(n=1000)
    intent(inout):: A
    real, dimension(n,n):: A,B,C
!hpf$ align with A:: B,C
!hpf$ distribute * A(block,*)              → 0
    ... B written, A read
    if (...B read) then
!hpf$ redistribute A(cyclic,*)             → 1
      ... A p written, A B read
    else
!hpf$ redistribute A(block,block)          → 2
      ... p written, A read
    endif
    do i=1, m+p
!hpf$ redistribute A(block,*)              → 3
      ... C written, A read
!hpf$ redistribute A(*,block)              → 4
      ... A written, A C read
    enddo
  end subroutine remap                     → E
```

Figure 4.13: Code example

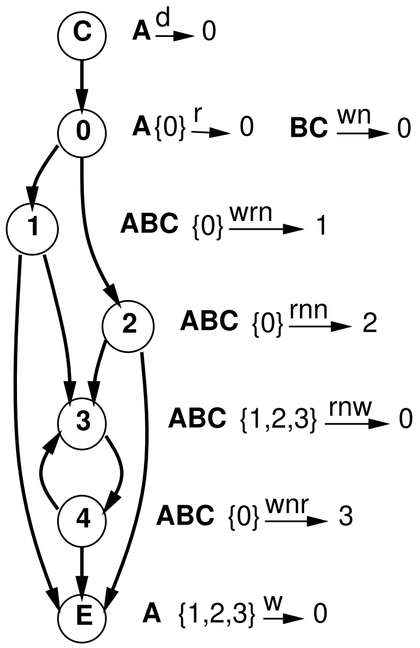


Figure 4.14: Remapping graph

reach latter vertices is changed. Indeed, the whole set of reaching mappings must be recomputed. It is required to update this set because we plan a compilation of remappings, thus the compiler must know all possible source and target mapping couples that may occur at run time. This recomputation is a may forward standard data-flow problem.

This optimization is performed as follows:

remove useless remappings:

simply by deleting the leaving mapping of such arrays.

$$\forall v \in \mathcal{V}(\mathcal{G}_R), \forall \mathtt{A} \in S(v), U_{\mathtt{A}}(v) = \mathtt{N} \Rightarrow L_{\mathtt{A}}(v) = \perp$$

recompute reaching mappings:

initialization: use 1-step reaching mappings

$$\forall v \in \mathcal{V}(\mathcal{G}_R), \forall \mathtt{A} \in S(v), R_{\mathtt{A}}(v) = \bigcup_{\substack{v' \in \mathrm{pred}(v) \\ \mathtt{A} \in A(v',v), U_{\mathtt{A}}(v') \neq \mathtt{N}}} L_{\mathtt{A}}(v')$$

Reaching mappings at a vertex are initialized as the leaving mappings of its predecessors which are actually referenced.

propagation: optimizing function

$$\forall v \in \mathcal{V}(\mathcal{G}_R), \forall \mathtt{A} \in S(v), R_{\mathtt{A}}(v) = R_{\mathtt{A}}(v) \cup \bigcup_{\substack{v' \in \mathrm{pred}(v) \\ \mathtt{A} \in A(v',v), U_{\mathtt{A}}(v') = \mathtt{N}}} R_{\mathtt{A}}(v')$$

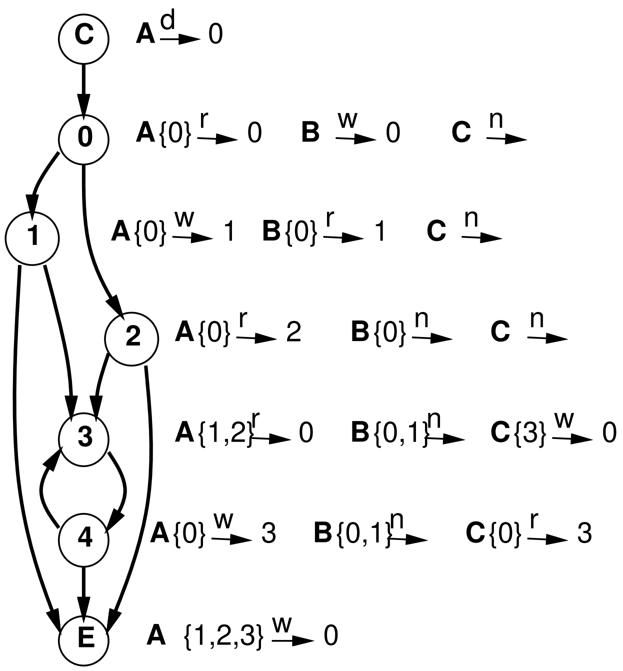


Figure 4.15: Example after optimization

> The function propagates reaching mappings along paths on which the array is not referenced, computing the transitive closure of mappings on those paths.

The iterative resolution of the optimizing function is increasing and bounded, thus it converges.

Let us assume $\mathcal{O}(1)$ set element put, get and membering functions. Let m be the number of vertices in $\mathcal{G}_R$, p the number of distributed arrays, q the maximum number of different mappings for an array and r the maximum number of predecessors for a vertex. Then the worst case time complexity of the optimization, for a simple iterative implementation, is $\mathcal{O}(m^2pqr)$. Note that m, q and r are expected to be very small.

This optimization is correct and the result is optimal:

Theorem 1 *The computed remappings (from new reaching to remaining leaving) are those and only those that are needed (according to the static information provided by the data flow graph):*

$$\forall v \in \mathcal{V}(\mathcal{G}_R), \forall \mathtt{A} \in S(v), U_{\mathtt{A}}(v), \forall a \in R_{\mathtt{A}}(v),$$
$$\exists v' \textit{ and a path from } v' \textit{ to } v \textit{ in } \mathcal{G}_R,$$
$$\textit{so that } a \in L_{\mathtt{A}}(v') \textit{ and } \mathtt{A} \textit{ is not used on the path.}$$

Proof sketch: construction of the path by induction on the solution of the data flow problem. Note that the path in $\mathcal{G}_R$ reflects an underlying path in the control flow graph with no use and no remapping of the array.

```
!hpf$ distribute A(block)             → 0
      ... A read
      if (...) then
!hpf$   redistribute A(cyclic)        → 1
        ... A written
      else
!hpf$   redistribute A(cyclic(2))     → 2
        ... A read
      endif
!hpf$ redistribute A(block)           → 3
      ... A read
      end
```

Figure 4.16: Flow dependent live copy

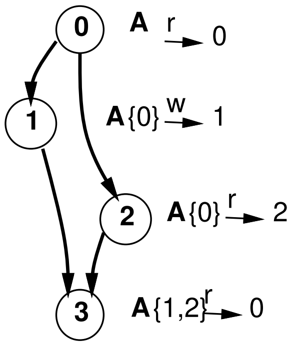


Figure 4.17: Corresponding $\mathcal{G}_R$

Figure 4.15 displays the remapping graph of our example after optimization. From the graph it can be deduced that array `A` may be used with all possible mappings $\{0, 1, 2, 3\}$, but array `B` is only used with $\{0, 1\}$ and array `C` with $\{0, 3\}$. Array `C` is not live but within the loop nest, thus its instantiation can be delayed, and may never occur if the loop body is never executed. Array `B` is only used at the beginning of the program, hence all copies can be deleted before the loop. The generation of the code from this graph is detailed in Section 4.5.

4.4.2 Dynamic live copies

Through the remapping graph construction algorithm, array references in the control graph $\mathcal{G}_C$ were updated to an array version with a statically known mapping. The remapping graph holds the information necessary to organize the copies between these versions in order to respect the intended semantics of the program. The first idea is to allocate the leaving array version when required, to perform the copy and to deallocate the reaching version afterwards.

However, some copies could be kept so as to avoid useless remappings when copying back to one of these copies if the array was only read in between. The remapping graph holds the necessary information for such a technique. Let us consider the example in Figure 4.16 and its corresponding remapping graph in Figure 4.17. Array `A` is remapped differently in the branches of the condition. It may be only modified in the `then` branch. Thus, depending on the execution path in the program, array copy $\mathtt{A}_0$ may reach remapping statement 3 live or not. In order to catch such cases, the liveness management is delayed until run time: dead copies will be deleted (or mark as dead) at the remapping statements.

For the example above, when executing remapping statement 1 the runtime knows that the array may be modified, thus can tag copy 0 as dead, but when executing remapping statement 2 the array is only read afterwards hence the copy can be kept as live. At remapping statement 3 the runtime can then check for the liveness tag associated to array version 0 and avoid a copy and the associated communication if live. Thus simple runtime data structures can keep track of live copies and avoid remappings, depending on the execution path of the program.

Keeping array copies so as to avoid remappings is a nice but expensive optimization, because of the required memory. Thus it would be interesting to keep only copies that may be used latter on. In the example above it is useless to keep copies $\mathtt{A}_1$ or $\mathtt{A}_2$ after remapping statement 3 because the array will never be remapped to one of these distribution. Determining at each vertex the set of copies that may be live and used latter on is a may backward standard data flow problem: leaving copies must be propagated backward on paths where they are only read. Let $M_{\mathtt{A}}(v)$ be the set of copies that may be live after v.

initialization: directly useful mappings

$$\forall v \in \mathcal{V}(\mathcal{G}_R), \forall \mathtt{A} \in S(v), M_{\mathtt{A}}(v) = L_{\mathtt{A}}(v)$$

propagation: optimizing function

$$\forall v \in \mathcal{V}(\mathcal{G}_R),\ \forall \mathtt{A} \in S(v),\ U_{\mathtt{A}}(v) \in \{\mathtt{N}, \mathtt{R}\}, M_{\mathtt{A}}(v) = M_{\mathtt{A}}(v) \cup \bigcup_{\substack{v' \in \mathrm{succ(v)} \\ \mathtt{A} \in A(v,v')}} M_{\mathtt{A}}(v')$$

Copies that may be of interest are propagated towards predecessors if the array is not modified (neither `W` nor `D`).

At a remapping statement v the runtime should only keep the live copies described in $M(v)$. Moreover, as the liveness information is managed dynamically, the runtime may decide to free some live copies because not enough memory is available. This would simply lead to a copy based update of the values of the array instead of the communication-free reuse of the values if the copy had been kept.

Such a scheme makes an implicit choice of granularity for managing live copies, by staying at the remapping statement level. One could imagine a very precise runtime tagging of modified arrays. However, where to put the needed tag updates so as to avoid any significant runtime overhead is not obvious. At the remapping graph level, communication is expected to occur at vertices, thus the runtime implication of the liveness management should be very small compared to these costs. Moreover it is likely that the remappings have been carefully placed in the program.

4.4.3 Other optimizations

Further optimization can be thought of, as discussed in [121].

Array kill analysis, for instance based on array regions [77], tells whether the values of an array are dead at a given point in the program. This semantical analysis can be used to avoid remapping communication of values that will never be reused. Array regions can also describe a subset of values which are live, thus the remapping communication could be restricted to these values, reducing communication costs further. However such compile-time intensive advanced semantical analyses are not the common lot of commercial compilers. Our prototype HPF compiler includes a `kill` directive for the user to provide this information. The directive simply creates a remapping vertex with the `D` use information, which will require no communication as discussed in Section 4.5.

Another idea is to move remappings around in the control flow graph, especially for moving them out of loops. However some pitfalls must be avoided. From the

```
!hpf$ distribute A(block)
      ... A
      do i=1, t
!hpf$   redistribute A(cyclic)
        ... A
!hpf$   redistribute A(block)
      enddo
      ... A
```

Figure 4.18: Loop invariant remappings

```
!hpf$ distribute A(block)
      ... A
      do i=1, t
!hpf$   redistribute A(cyclic)
        ... A
      enddo
!hpf$ redistribute A(block)
      ... A
```

Figure 4.19: Optimized version

code in Figure 4.18 we suggest to move the remappings as shown in Figure 4.19. This differs from [121]: the initial remapping is not moved out of the loop, because if `t` < 1 this would induce a useless remapping. The remapping from `block` to `cyclic` will only occur at the first iteration of the loop. At others, the runtime will notice that the array is already mapped as required just by an inexpensive check of its status.

4.5 Runtime issues

The remapping graph information describying array versions reaching and leaving remapping vertices must be embedded into the program through actual copies in order to fullfil the semantical requirements for the correctness of the execution. Moreover some optimizations described in the previous sections rely on the runtime to be performed. This section presents the runtime implications of our scheme.

4.5.1 Runtime status information

First, some data structure must be managed at run time to store the needed information: namely, the current status of the array (which array version is the current one and may be referenced); the live copies; the reaching status at a call cite.

The current status of an array can be simply described by a descriptor holding the version number, for instance `status(A)` for array `A`. By testing this status, the runtime will be able to notice which version of an array reaches a remapping statement. This status will have to be updated each time the array is remapped.

Secondly, the runtime must be able to test whether a version of a given array is live at a point. This is a boolean information to be attached to each array version. It will be denoted `live`($\mathtt{A}_x$) for liveness of array $\mathtt{A}_x$. The liveness information will be updated at each remapping vertex, depending of the latter use of the copies from this vertex.

If interpreted strongly, Constraint 2 (p. 164) may imply that arrays as call arguments are considered as references and thus should not bare any ambiguity, such as the one depicted in Figure 4.20. However, since an explicit remapping of the array is inserted, the ambiguity is solved before the call, hence there is no need to forbid such cases. The issue is to restore the appropriate reaching mapping on return from the call. This can be achieved by saving the current status of the array

```
      interface
        subroutine foo(X)
!hpf$     distribute X(block)
        end subroutine
      end interface
!hpf$ distribute A(cyclic)
      ... A
      if (...) then
!hpf$   redistribute A(cyclic(2))
        ... A
      endif
! A is cyclic or cyclic(2)
! foo requires a remapping
      call foo(A)
```

Figure 4.20: Subroutine calls

```
! save the reaching status
      reaching(A)=status(A)
!hpf$ redistribute A(block)
      call foo(A)
! restore the reaching mapping
      if (reaching(A)=1) then
!hpf$   redistribute A(cyclic)
      elif (reaching(A)=2) then
!hpf$   redistribute A(cyclic(2))
      endif
```

Figure 4.21: Mapping restored

that reached the call as suggested in Figure 4.21. Variable `reaching(A)` holds the information. The saved status is then used to restore the initial mapping after the call.

4.5.2 Copy code generation

The algorithm for generating the copy update code and live management from the remapping graph is outlined in Figure 4.23. Copy allocation and deallocation are inserted in the control flow graph to perform the required remappings. The algorithm relies on the sets computed at the $\mathcal{G}_R$ optimization phase to insert only required copies and to keep only live copies that may be useful while others are cleaned.

The first loop inserts the runtime data structure management initialization at the entry point in the subroutine. All copies are denoted as not live, and no copy receives an *a priori* instantiation as the current copy. The rationale for doing so is to delay this instantiation to the actual use of the array, which is not necessarily with the initial mapping, and will not be necessarily used, as Array `C` in Figure 4.13.

The second loop nest extracts from the remapping graph the required copy, for all vertex and all remapped arrays, if there is some leaving mapping for this array at this point. The generated code first checks whether the status is different from the required leaving status, in order to skip the following communication if the copy is already ok. If not, it is first allocated. If the array is not to be fully redefined afterwards, a copy from the reaching copy to the desired leaving copy if performed if the values are not live. Then the current status of the array and its associated liveness information are updated.

Copies that were live before but that are not live any more are cleaned, *i.e.* both freed and marked as dead. Finally a full cleaning of all local arrays is inserted at the exit vertices. Figure 4.22 shows some example of generated copy code for the remapping vertex depicted in Figure 4.8, assuming that all copies may be useful for latter remappings.

It must be noted that dead arrays (`D`) do not require any actual array copy,

```
if (status(A)≠2) then
  allocate A2 if needed
  if (not live(A2)) then
    if (status(A)=1) A2=A1
    if (status(A)=3) A2=A3
    live(A2)=true
  endif
  status(A)=2
endif
```

Figure 4.22: Code for Figure 4.8

thus none is generated, hence this will result in no communication at run time. Moreover, there is no initial mapping imposed from entry in the subroutine. If an array is remapped before any use, it will be instantiated at the first remapping statement encountered at runtime with a non empty leaving copy, which may be the entry vertex v_0 or any other. Finally, care must be taken not to free the array dummy argument copy, since it is referenced by the caller. Thus the `allocate` and `free` must be only applied to local variables.

Another benefit from this dynamic live mapping management is that the runtime can decide to free a live copy if not enough memory is available on the machine to keep it, and to change the corresponding liveness status. If required latter on, the copy will be regenerated, *i.e.* both allocated and properly initialized with communication. Since the generated code does not assume that any live copy must reach a point in the program, but rather decided at remapping statements what can be done, the code for the communication will be available.

4.6 Conclusion

Most of the techniques described in this paper are implemented in our prototype HPF compiler. The standard statically mapped HPF code generated is then compiled, with a special code generation phase for handling remapping communication due to the explicit array copies. The described techniques, especially the runtime live management, could be extended to interprocedural copies with some conventions at call cites.

Acknowledgment

I am thankful to Corinne Ancourt, Béatrice Creusillet, François Irigoin and Pierre Jouvelot for their helpful comments and suggestions.

```
for A ∈ S(v_0)
   append to v_0 "status(A)=⊥"
   for a ∈ C(A)
      append to v_0 "live(A_a)=false"
   end for
end for
for v ∈ V(G_R) − {v_c}
   for A ∈ S(v)
      if (L_A(v) ≠⊥) then
         append to v "if (status(A)≠ L_A(v)) then"
         append to v "allocate A_{L_A(v)} if needed"
         append to v "if (not live(A_{L_A(v)})) then"
         if (U_A(v) ≠ D) then
            for a ∈ R_A(v) − {L_A(v)}
               append to v "if (status(A)=a) A_{L_A(v)}=A_a"
            end for
         end if
         append to v "live(A_{L_A(v)})=true"
         append to v "endif"
         append to v "status(A)=L_A(v)"
         append to v "endif"
      end if
      for a ∈ C(A) − M_A(v)
         append to v "if (live(A_a)) then"
         append to v "  free A_a  if needed"
         append to v "  live(A_a)=false"
         append to v "endif"
      end for
   end for
end for
for all A
   for a ∈ C(A)
      append to v_e "if (live(A_a) and needed) free A_a"
   end for
end for all
```

Figure 4.23: Copy code generation algorithm

Chapitre 5

Optimal Compilation of HPF Remappings

Fabien Coelho et Corinne Ancourt

Ce chapitre doit paraître sans l'annexe dans JPDC [68].

Résumé

Les applications dont les motifs d'accès aux tableaux varient au cours de l'exécution du programme sur une machine parallèle à mémoire répartie nécessitent de changer dynamiquement le placement des données. HPF (High Performance Fortran) permet ces déplacements, éventuellement sur des données partiellement répliquées, de manière explicite avec les directives `realign` et `redistribute`, et de manière implicite lors des appels et retours de procédures. Cependant ces possibilités du langage sont laissées hors de la nouvelle version du langage en cours de définition. Cet article présente une nouvelle technique de compilation pour gérer ces déplacements, en ciblant une architecture parallèle à passage de messages. La première phase retire tous les déplacements inutiles qui apparaissent naturellement dans les procédures. La seconde genère le code de déplacement, et profite de la réplication éventuelle pour réduire le temps de recomposition du tableau. Un nombre minimum de messages ne contenant que les données nécessaires est envoyé sur le réseau. Ces techniques sont implantées dans HPFC, notre prototype de compilateur HPF. Des résultats expérimentaux sur une ferme d'Alphas (DEC) sont également présentés.

Abstract

Applications with varying array access patterns require to dynamically change array mappings on distributed-memory parallel machines. HPF (High Performance Fortran) provides such remappings, on data that can be replicated, explicitly through `realign` and `redistribute` directives, and implicitly at procedure calls and returns. However such features are left out of HPF 2.0 for efficiency reasons. This paper presents a new technique for compiling HPF remappings onto message-passing parallel architectures. Firstly,

useless remappings that appear naturally in procedures are removed. Secondly, the SPMD generated code takes advantage of replication to shorten the remapping time. Communication is proved optimal: a minimal number of messages, containing only the required data, is sent over the network. The technique is fully implemented in HPFC, our prototype HPF compiler and was experimented on a DEC Alpha farm.

5.1 Introduction

Many applications and kernels, such as ADI (Alternating Direction Integration), linear algebra solvers [209], 2D FFT [118] (Fast Fourier Transform) and signal processing [189], require different array mappings at different computation phases for efficient execution on distributed-memory parallel machines (*e.g.* CRAY T3D, IBM SP2, DEC Alpha farm). Data replication, sometimes partial, is used to share data between processors. Data remapping and replication are often combined: a parallel matrix multiplication accesses a whole row and column of data to compute each single target element, hence the need to remap data with some replication for parallel execution. Moreover, automatic data layout tools [166, 54] suggest data remappings between computation phases. Finally HPF compilers generate realignment-based communication [145]. Thus handling data remappings efficiently is a key issue for high performance computing.

High Performance Fortran [95, 96, 162] (HPF), a Fortran 90-based data-parallel language, targets distributed-memory parallel architectures. Standard directives are provided to specify array mappings that may involve some replication. These mappings are changed dynamically, explicitly with *executable* directives (`realign`, `redistribute`) and implicitly at procedure calls and returns for *prescriptive* argument mappings. These useful features are perceived as difficult to compile efficiently and thus are left out of currently discussed HPF 2.0 [97]. If not supported, or even not well supported, applications requiring them will not be ported to HPF. The key issues to be addressed are the reduction of runtime overheads induced by remappings and the management of the rich variety of HPF mappings.

5.1.1 Related work

Any technique that handles HPF array assignments can be used to compile remappings: the induced communication is the one of an array assignment `A=B`, where `B` is mapped as the source and `A` as the target. Such techniques are based on finite state machines [53, 133, 152], closed forms [117, 229, 119], Diophantine equations [200, 25, 247] or polyhedra [2, 5, 245, 233]. However *none* of these techniques considers load-balancing and broadcasts. Also issues such as handling different processor sets, multidimensional distributions, communication generation and local addresses are not always clearly and efficiently managed in these papers, thus dedicated optimized techniques have been proposed. Runtime library support [234, 235] is suggested for simple cases involving neither shape changing (same distributed dimensions), nor alignment or replication; multidimensional remappings are decomposed into 1D remappings; general cyclic distributions of non multiple block sizes are also decomposed, hence resulting in several remappings at runtime. *Ad hoc* descriptors called *pitfalls* are devised [212], but alignment, replication and shape changing are not considered either. A polyhedron-based approach is outlined [246], for realignments with a fixed general cyclic distribution onto a 1D processor array; the alignments, unlike HPF, involve arbitrary affine functions. An optimal technique for a constrained case is also suggested [144], assuming that the HPF to real processor mapping can be changed; alignment and replication are not considered.

5.1.2 Contributions

In contrast to previous approaches, this paper handles *all* HPF mapping issues including arbitrary alignments and distributions that involves partial replication, as well as the local addressing scheme. Load balancing, broadcast and node contention issues are also discussed. Thanks to the mathematical formulation on which the technique is grounded and to simple programming tricks, the generated communication is proved optimal.

The technique consists of compiling communication, by generating a proper message-passing SPMD code. Firstly, the candidate source to target mappings that may occur at run-time are computed as a simple may forward data-flow problem. By doing so, useless remappings are removed automatically by detecting the paths on which remapped arrays are not referenced. Such useless remappings arise naturally in programs: the change of both alignment and distribution of an array requires a `realign` and a `redistribute`, resulting in two remappings if no special care is taken; a template redistribution (Even when no templates are used [258] array alignment generates the problem) induces the remapping of all aligned arrays, even if they are not all referenced afterwards; at an interprocedural level, two consecutive subroutine calls may require the same remapping for an array, resulting in two useless remappings on return from the first subroutine and on entry in the second; if two different mappings are required, it is interesting to remap data directly rather than using the intermediate original mapping. Such examples do not arise from badly written programs, but from a normal use of HPF features, thus they demonstrate the need for compile time data flow optimizations to avoid useless costly remappings at runtime.

Once mapping couples defining remappings are determined, a linear algebra based description of the problem is built (Section 5.2). This description incorporates all HPF mapping issues (alignment strides, general distributions, partial replication). It is completed with global to local addressing constraints and a load balancing equation so as to benefit from replication. From this representation, standard polyhedron manipulation techniques allow to generate a SPMD code (Section 5.3). The conclusion presents experiments performed on a DEC Alpha farm as well as optimality and correctness results (Section 5.4).

5.2 Linear formalization

Let us consider the example in Figure 5.1 (left). This example is deliberately contrived, and designed to show all the capabilities of our algorithm. Real application remappings should not present all these difficulties at once, but they should frequently include some of them. Vector `A` is remapped from a block distribution onto 3D processor grid `Ps` to a general cyclic distribution onto 2D processor grid `Pt`

`      parameter (n=20)`	$n = 20,$
`      dimension A(n)`	$1 \leq \alpha_1 \leq n,$
`!hpf$ template T(2,n,2)`	$1 \leq \theta_1 \leq 2,\ 1 \leq \theta_2 \leq n\ , 1 \leq \theta_3 \leq 2,$
`!hpf$ align A(i) with T(*,i,*)`	$\theta_2 = \alpha_1,$
`!hpf$ processors Ps(2,2,2)`	$1 \leq \psi_1 \leq 2,\ 1 \leq \psi_2 \leq 2,\ 1 \leq \psi_3 \leq 2,$
`!hpf$ distribute onto Ps :: &`	$0 \leq \delta_2 \leq 9,$
`!hpf$     T(block,block,block)`	$\theta_2 = 10\psi_2 + \delta_2 - 9,$
`!`	
`! Remapping:`	*Distributed and Replicated dimensions:*
`!      from Ps(:,block,:)`	$\psi_D = \{\psi_2\},\ \psi_R = \{\psi_1, \psi_3\}$
`!        to Pt(cyclic(2),:)`	$\psi'_D = \{\psi'_1\},\ \psi'_R = \{\psi'_2\}$
`!`	
`!hpf$ processors Pt(5,2)`	$1 \leq \psi'_1 \leq 5,\ 1 \leq \psi'_2 \leq 2,$
`!hpf$ redistribute onto Pt :: &`	$0 \leq \delta'_1 \leq 1,$
`!hpf$     T(*,cyclic(2),block)`	$\theta_2 = 10\gamma'_1 + 2\psi'_1 + \delta'_1 - 1$

Figure 5.1: Remapping example and its linear formalization

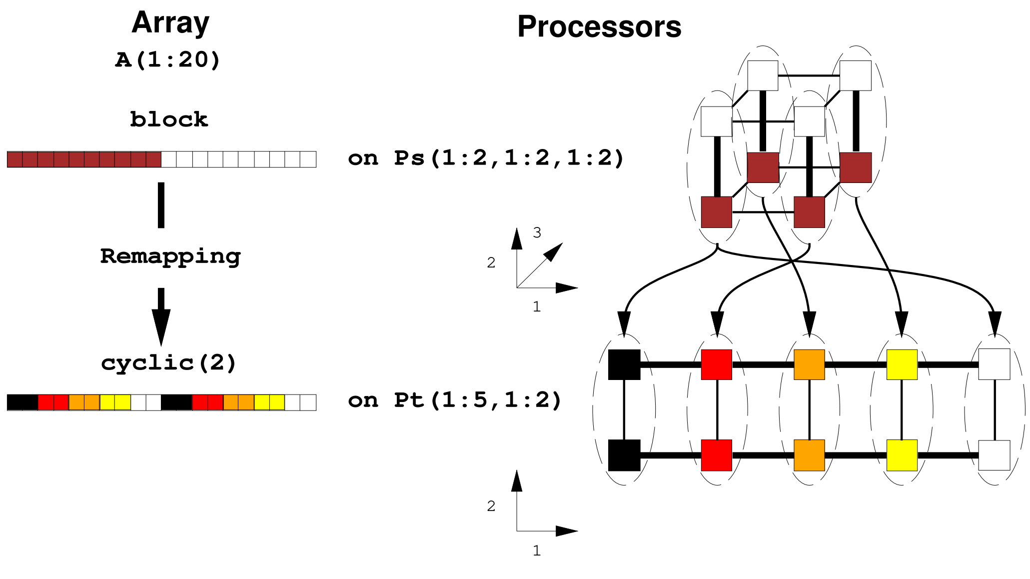


Figure 5.2: Array A remapping

through template T redistribution. Both source and target mappings involve partial replication. The corresponding data layouts are depicted in Figure 5.2. Shades denote data to processor mapping. The initial mapping is a `block` distribution of A onto the second dimension of `Ps`. Each column of dark and light processors in `Ps` owns a full copy of A: A is replicated 4 times. The target mapping is a `cyclic(2)` distribution onto `Pt` first dimension. Each line owns a full copy of A: A is replicated twice. In the following, processors that hold the same data within a processor arrangement due to replication are called twins for that processor arrangement: for instance `Pt` columns in our example are composed of twins. The generated SPMD code handles remapping communication between processors as denoted by the arrows and ovals. On the target side, each column of processors waits for *exactly* the same data, hence the opportunity to broadcast the same messages to these pairs. On the source side, each column can provide any needed data, since it owns a full copy of A. The different columns can deal with different target processors, thus balancing the load of generating and sending the messages. For this example, 5 different target processor sets are waiting for data that can be sent by 4 source processor groups. The source to target processor assignment statically balances targets among possible senders in a cyclic fashion.

5.2.1 Formalization of HPF directives

Declarations and HPF directives are embedded into linear constraints as suggested in [5]. The constraints derived for the example are shown in Figure 5.1 (right), each in front of its corresponding source code line. From declarations, lower and upper bounds are defined for each dimension of array (α), template (θ) and processor (ψ) grids. Simple variables designate the source mapping and primed ones the target. Numerical subscripts show dimension numbers. From HPF directives other constraints are added: our alignment is very simple, but affine expressions are needed for general alignments (for instance `align A(*,i) with T(3*i-2,*)` would lead to $\theta_1 = 3\alpha_2 - 2$); template distributions require additional variables, for modeling blocks and cycles; δ is the offset within a block; γ is the cycle number, *i.e.* the number of wraps around the processors for cyclic distributions; general cyclic distributions need both variables. Distributions on replicated dimensions are not included in the formalization. Finally the processor Distributed and Replicated dimensions for array A are described through $\psi_D[']$ and $\psi_R[']$ sets.

This formalization defines a polyhedron on integers in the array, template, processors etc. index space. This polyhedron can be enhanced by adding array local addresses (β) [61]: A can be declared as `As(1:10)` with local address $\beta_1 = \delta_2 + 1$ on source processors, and `At(1:4)` with $\beta'_1 = 2\gamma'_1 + \delta'_1 + 1$ on target processors. Two dimensions and two equations are simply added to the polyhedron for local addresses. This gives a linear description of the data distribution and of the communication problem, *i.e.* the enumeration of the elements to be sent and received. Indeed, integer solutions to these equations exactly describe the array elements to be communicated, with their associated source and target processors: for instance solution $\alpha = (5), \psi = (1,1,2), \psi' = (3,2), \ldots$ expresses that `A(5)` can be transfered from `Ps(1,1,2)` to `Pt(3,2)`.

5.2.2 Broadcast and load balancing

However, before using this description for generating a SPMD code, it must be noted that the system is not constrained enough for generating a minimal and unique code. For the example above and due to data replication, other solutions on the processor arrangement exist for `A(5)`: $\psi = (1..2, 1, 1..2), \psi' = (3, 1..2)$. There are several candidate senders and receivers for each data. Replication on the target side is an opportunity for broadcasting the very same messages to several processors that need the same data, and will be managed as such at the code generation phase by separating constraints on ψ'_R.

Replication on the source side provides an opportunity for balancing the load of sending data as suggested in Figure 5.2. The depicted source to target assignment can be achieved with one additional equation, linking the source replicated dimensions (ψ_R) to the target distributed dimensions (ψ'_D). For our example, such an equation is $\psi'_1 = 4\lambda + 2\psi_3 + \psi_1 - 2$. It introduces a λ variable for cycling over the available source processors, because there are more targets than sources. Solutions to this equation under the processor declaration and replication constraints ($1 \leq \psi'_1 \leq 5 \ldots$) achieve the desired assignment: for instance $\psi'_1 = 3 \Rightarrow \lambda = 0, \psi_1 = 1, \psi_3 = 2$, thus the third column of `Pt` receives data from the back left column of `Ps`, as depicted.

The general formula for the message load balancing equation is:

$$\mathsf{linearize}(\psi'_D) = \mathsf{extent}(\psi_R) \times \lambda + \mathsf{linearize}(\psi_R)$$

where linearize is a dense linearization of the variables and extent is the number of distinct elements. The rationale for the linearization is to get rid of the dimension structure of the processors, for finding the best load balance by distributing cyclically all distributed target onto all replicated source processors. No assumption is made about the HPF to *real* processor mapping in the following. Different HPF processor arrangements may have common *real* processors.

Let $\mathcal{E}(p, \lambda, e)$ with $p = \psi \cup \psi'$ and e the other variables be the set of all constraints on these variables introduced so far for a remapping. Variables related to the template index space (θ) or to the initial array (α) are of no interest for the actual code generation and can be exactly removed. The $\mathcal{E}$ system defines a polyhedron linking the array elements to their source and target processors and to their local addresses. It allows to build a SPMD code generating communication as shown in the next section.

5.3 SPMD code generation

If all processors must enumerate all integer solutions to polyhedron $\mathcal{E}$, this is equivalent to the runtime resolution technique and is very inefficient. Moreover, it is interesting to pack at once the data to be transfered between two processors, in order to use only one buffer for message aggregation. Therefore some manipulations are needed to generate efficient code. It mainly consists of projecting some variables (*i.e.* eliminating them from the system: $\mathcal{X}_{|y}$ projects y in system $\mathcal{X}$). Such an elimination is not necessarily exact [4, 210], hence the computed *shadow* may be larger than the real one.

Replicated target processor dimensions (ψ'_R) can be extracted from $\mathcal{E}$. This information is only required for broadcasting the messages and has no impact on its contents. $\mathcal{E}_{|\psi'_R}$ exactly stores the remaining information. In order to first enumerate the couples of processors that must communicate, and then the associated message, a superset $\mathcal{P}(\psi, \psi'_D, \lambda) = \mathcal{E}_{|\psi'_R, e}$ of these communicating processors is derived. For our example, the shadow of communicating processors is:

$$\mathcal{P}(\psi, \psi'_1, \lambda) = \{1 \leq \psi_{1,2,3} \leq 2, 1 \leq \psi'_1 \leq 5, \psi'_1 = 4\lambda + 2\psi_3 + \psi_1 - 2\}$$

$\mathcal{P}$ represents processor couples that *may* have to communicate, because the projection performed to build this set is not necessarily exact in the general case. The runtime insures that only needed communication is performed.

5.3.1 Generated code

Figure 5.3 shows the generated SPMD code based on $\mathcal{E}$ and $\mathcal{P}$. Enumeration codes are denoted **for** $v \in \mathcal{X}[z]$: integer solutions to $\mathcal{X}$ are scanned considering variables z as external parameters, that must be instantiated before-hand. Such scanning codes can be generated by several techniques, based on FOURIER elimination [136, 7] or a parametric simplex [86]. $\mathcal{D}(v)$ designate all the possible values of v from the declaration and replication constraints. Functions srcaddr() and trgaddr() insures local address computations for the source and target local arrays. For our running example the local addresses are directly enumerated in e since the local addressing scheme was integrated in the formalization.

Let us focus on the code: each *real* processor plays a part or none in processor grids `Ps` and `Pt`. The send part is guarded for `Ps` and the receive part for `Pt`. A local copy part directly performs a copy when data is available locally. $\mathcal{P}$ is dually used in the send and receive parts: the possible senders select themselves ($\mathcal{P}_{|\psi',\lambda}$) and enumerate their target (ψ') processors, while the receivers select themselves ($\mathcal{P}_{|\psi,\lambda}$) and enumerate their source (ψ) processors. Thus $\mathcal{P}$ allows both guards and enumerations of processors on the source and target sides.

In the send part, once communicating HPF processor couples are selected, the runtime checks whether they refer to processors holding distinct data, so as to avoid generating a message from a processor to itself or to a twin of itself. Then associated array elements (e) are enumerated, packed and broadcast to all processors that may need and that are not twins of the current(*i.e.* do not hold the same data because of replication). The array element enumeration is parameterized by the processor identities (ψ, ψ') which are instantiated in the outer loops and through guards. The receive part is the complement of the send. Possible sources are first enumerated. If they do not hold the same data, a message is received and array elements are unpacked and stored properly. If the source is a twin, then data is available locally and is copied directly. The rationale for copying in the receive part is not to delay messages.

5.3.2 Programming tricks

Some simple programming tricks are used in the generated code. Firstly, empty messages are not sent thanks to the `empty` guard. However these not sent messages must not be received. In order to so, messages are actually received only when

```
– remapping of array A from processors Ps to processors Pt
– local declarations: As on source and At on target
– code for a real processor playing an HPF processor

if (I am in Ps) then – send part
  ψ = my id in Ps
  if (ψ ∈ P_{|ψ',λ}(ψ)) then – something may have to be sent
    for (λ, ψ'_D) ∈ P_{|ψ'_R}[ψ] – enumerate target processors
      if (I and ψ'_D × D(ψ'_R) not all Ps twins) then
        – some distinct distributed targets: pack and send
        empty = true
        for e ∈ E_{|ψ'_R}[ψ, ψ'_D, λ] – enumerate elements to send
          pack As(srcaddr(e)) in buffer
          empty = false – now the buffer is not empty
        endfor
        if (not empty) then
          broadcast buffer to ψ'_D × D(ψ'_R) except my Ps twins
        endif
      endif
    endfor
  endif
endif
if (I am in Pt) then – receive or copy part
  Allocate At
  ψ' = my id in Pt
  if (ψ'_D ∈ P_{|ψ'_R,ψ,λ}(ψ'_D)) then – something may have to be received
    for (ψ, λ) ∈ P_{|ψ'_R}[ψ'_D] – enumerate source processors
      if (ψ and ψ' not Ps twins) then
        – non local, lazy reception and unpacking
        first = true
        for e ∈ E_{|ψ'_R}[ψ, ψ'_D, λ] – enumerate elements to receive
          if (first) then receive buffer from ψ, first = false
          unpack At(trgaddr(e)) from buffer
        endfor
      else – copy local data
        for e ∈ E_{|ψ'_R}[ψ, ψ'_D, λ]
          At(trgaddr(e)) = As(srcaddr(e))
        endfor
      endif
    endfor
  endif
endif
if (I am in Ps) then Free As
```

Figure 5.3: SPMD remapping code

```
! A is declared As(1:10) on Ps and At(1:4) on Pt

IF (I AM IN(Ps)) THEN
  ψ = (ψ1,ψ2,ψ3) = MY ID IN(Ps)
  DO λ = 0, (7-ψ1-2*ψ3)/4
    ψ'1 = -2+ψ1+2*ψ3+4*λ
    index = 0
    DO β1 = 2*ψ'1-1, 2*ψ'1
      BUFFER(index++) = As(β1)
    ENDDO
    CALL BROADCAST(BUFFER,Pt(ψ'1,:)) ! send
  ENDDO
ENDIF

IF (I AM IN(Pt)) THEN
  ψ' = (ψ'1,ψ'2) = MY ID IN(Pt)
  DO ψ1 = 1, 2
    DO ψ3 = 1, 2
      DO λ = (5-ψ1-2*ψ3+ψ'1)/4, (2-ψ1-2*ψ3+ψ'1)/4
        DO ψ2 = 1, 2
          IF (Ps(ψ)<>Pt(ψ')) THEN ! receive and unpack
            CALL RECEIVE(BUFFER,Ps(ψ))
            index = 0
            DO β'1 = 2*ψ2-1, 2*ψ2
              At(β'1) = BUFFER(index++)
            ENDDO
          ELSE ! local copy
            DO β1 = 2*ψ'1-1, 2*ψ'1
              β'1 = β1+2*ψ2-2*ψ'1
              At(β'1) = As(β1)
            ENDDO
          ENDIF
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDIF
```

Figure 5.4: HPFC generated SPMD code for the example

some data *must* be unpacked, through the `first` guard within the unpack loop. Thus no empty message is transfered: if the message is empty the body of the unpack loop is never executed and the receive never performed.

Secondly, messages are only communicated between non twin processors. Such a filtering is performed at runtime through appropriate guards, because no assumption is made at compile time about the HPF to *real* processor mapping. If the selected sender is a twin, data is simply copied locally since it is available.

Finally, node contention issues [213, 214, 215] could be addressed by the generated code. Instead of all the senders to enumerate the receivers in the same order, the generated loop nest can be used to precompute the list of targets. This list can be scanned in a different order on each source, reducing potential contention. From the receive side, arrival-driven reception and unpacking could also be implemented with some support from the communication library. The generated code would precompute the sources with non empty messages through a shortened loop, performing only the local copies and delaying the unpacking of messages on arrival. These two techniques are not yet implemented.

Figure 5.4 shows a prettyprint of a simplified version of the generated code for our example. Buffer overflow, lazy message and twin-related issues have been removed, and some simple cleaning was performed to enhance readability. The integer division requires positive remainders.

5.4 Conclusion

A general compilation technique was presented to handle HPF remappings efficiently. This section first presents optimality results that are proved in Section 5.5.4. The technique is implemented in HPFC, our prototype HPF compiler. Portable PVM-based [103] code is generated. The generated code has been experimented on a Alpha farm and the results are shown. Finally some comments and future work are outlined.

5.4.1 Optimality and correctness

Thanks to the mathematical formalization, HPF properties and simple programming tricks, we can prove that for any remapping, a minimal number of messages, containing only required data, is sent over the network. Thus communication is optimal. The essence of the proof is the following: firstly, replication in HPF is performed on independent dimensions of the processors what allows to manipulate these dimensions independently of other variables; also the distributions partition data, hence there is no doubt about possible senders apart from the orthogonal replication; moreover, the linear algebra based formalization of the problem is *exact*, thus only valid data is enumerated for communication between two processors, and runtime guards against twins insure that no message is sent to a processor that already holds these data; lazy communication prevents empty messages.

The proof is still valid for skewed alignments [2, 246]. It does not include the load balancing aspect of our technique: indeed, this load balancing is statically decided among all processors, without any information about the actual messages needed and the amount of data to be transfered. Runtime guards insure that only really needed messages are built, thus not only the performed communication is optimal, but also the packing. Some of the papers cited in Section 5.1.1 are communication optimal [212, 144]. Empty messages are generated by [246] and more communication than necessary by [234, 235]. However it must be noted that *none* addresses full HPF mappings: alignments, distributions or replication are restricted.

Part of the correctness of the SPMD code can also be shown from Figure 5.3 by proving it on the structure of the code: all variables are properly instantiated, all messages are balanced (as many are sent as received), etc. Also, the coordination of message packing and unpacking is insured because the very same loop nest is used in the send and receive parts for both operations. Finally, an upper bound of the required memory for storing intermediate and local data can be derived: $2 \times (\mathsf{memory}(\mathtt{As}) + \mathsf{memory}(\mathtt{At}))$.

5.4.2 Experimental results

The remapping code generation technique is fully implemented in HPFC, our prototype HPF compiler. HPFC primarily aims at demonstrating feasibility and being portable, rather than achieving very high efficiency on a peculiar architecture. PVM raw-encoded direct connections are used for communication. These new features were tested on a DEC FDDI networked Alpha farm at LIFL (Université de Lilles, France). The experimental results and derived data are presented in this section. They show improvements over the DEC HPF compiler, despite our high level PVM-based implementation. Not surprisingly, simpler and more efficient codes are generated for basic remappings, and more complicated ones for general cyclic distributions.

A square matrix transposition was chosen as a simple common testbed requiring advanced communication to emphasize the data amount and distribution complexity influence. The best wallclock execution times for transposing arrays of various sizes and distributions were measured. The reason for not presenting average times is that the machine was not dedicated to our experiments, thus it was not easy to obtain sound measurements. Experimental conditions (hardware, network, compilers, libraries) are detailed in [68]. For comparison purposes, we introduce the *transposition speed per processor*, expressed in MB/s/pe (MB stands for Mega Bytes, $1\,\mathrm{M} = 2^{20} = 1,048,576$). The rationale for this unit is that it is independent of the matrix size n, the type length l ($l = 8$ bytes in our experiments based on `real*8` arrays) and the number of processors p involved. Thus it allows to compare the performances obtained independently of these parameters, focusing on the achievements of our technique. The figures must be compared to the 12.5 MB/s peak communication bandwidth between two nodes. If t is the measured time, then speed s is defined as:

$$s = \frac{l.n^2}{2^{20}.p.t}$$

Figure 5.5 displays `(block,block)` transposition performances for different processor arrangements and matrix sizes. HPFC generally outperforms the DEC compiler by 20 to 30%, up to the PVM 1 MB message buffer size limit: when a message steps over 1 MB, it is split and a significant degradation occurs; for instance at $n = 1152$ for `(block,block)` distribution on `P(2,3)`. Figure 5.6 shows performances for various distributions on fixed processor arrangements. Small general cyclic distributions perform worst. Some artifacts (we suspect cache effects) are also noticeable for $n = 896$. On `P(2,3)`, `(cyclic,cyclic)` transpositions perform badly: code transformations such as invariant code motion are needed to build a better enumeration code; they were performed by hand on the `cyclic` distributed `P(2,2)` case. However, good performances for complex remappings compared to the simple and straightforward block distribution case were obtained.

5.4.3 Discussion

Our technique puts the remapping code generation problem in a single linear framework which deals with all HPF issues such as alignments or general cyclic distributions. Optimality results were presented. Namely, a minimal number of messages, containing only the required data, is sent over the network. Thus the technique minimizes both latency and bandwidth-related costs of the network, through message aggregation and exact enumeration of elements to be sent. Moreover load balancing issues are discussed and broadcasts are used.

The FOURIER elimination algorithm involved in the code generation phase of our technique has a theoretical exponential worst case behavior. However the practical execution time of our straightforward implementation for one remapping remains in the $0.1 - 1$s range, thanks to the independence of dimensions implied by HPF and to the small number of variables involved. The practical complexity of the code generation for multiple dimensions roughly is the *number of dimensions* times the complexity of the code generation for one dimension.

Extensions to the usual FOURIER elimination algorithm [2] are needed to handle a parametric number of processors. This also raises the issue of the static knowledge available to the compiler for generating faster codes, and the impact it may or should have on the HPF language definition on the one hand, and on the applications on the other. If not enough information is provided to the compiler, the generated code should rely on non necessarily efficient runtime libraries to perform the required communication, compromising the benefit users expect from their parallel code.

Future work includes: a new buffer management for using the PVM in place option; reducing the transfered data to what is actually needed through advanced program analyses [76, 75]; optimizing the generated code further through code transformations; compiling for other communication models such as one-sided communication.

Acknowledgments

We are thankful to (in alphabetical order) François BODIN for informal discussions, Béatrice CREUSILLET for pointers, Jean-Luc DEKEYSER for access of the Alpha farm, François IRIGOIN for the improvements he suggested, Pierre JOUVELOT for corrections, Ronan KERYELL for giving his opinion, Philippe MARQUET for technical support on the farm, William PUGH for suggestions and Xavier REDON for comments.

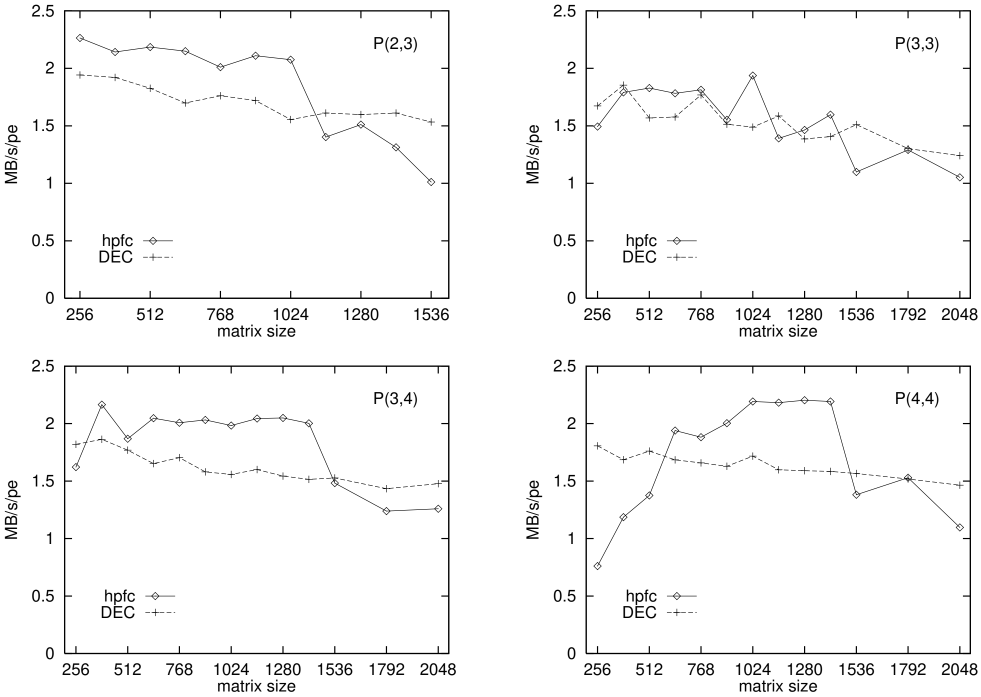


Figure 5.5: Transposition speed per processor for `(block,block)` distributions

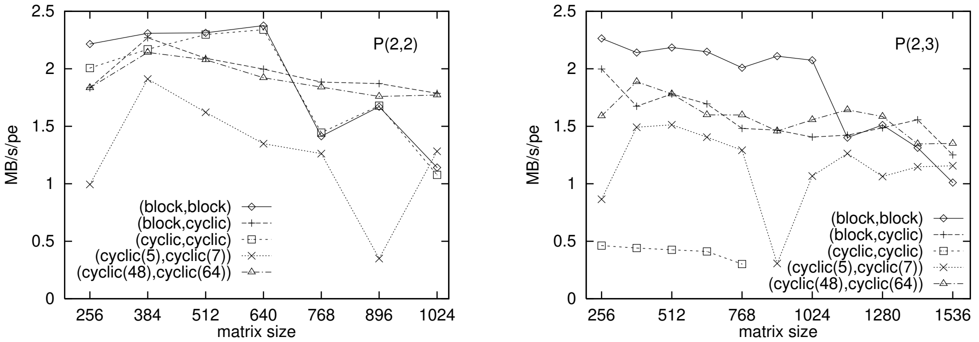


Figure 5.6: Transposition speed per processor on `P(2,2)` and `P(2,3)`

5.5 Appendix

This section presents details that could not be included in the core of the paper because of space limitation.

5.5.1 Notations

Linear algebra provides a powerful framework to characterize array element sets, represent HPF directives, generate efficient code, define and introduce optimizations and make possible correctness proof of the compilation scheme. Our compilation scheme uses polyhedra to represent the HPF data remapping problem. Notations are shown in Figure 5.7, 5.8 and 5.9. Greek letters denote individual or set of integer variables; calligraphic letters systems of linear equalities and inequalities on a set of variables. Such constraints implicitly define a polyhedron on the variables, *i.e.* the set of integer vectors that are solutions to the system. Different operations can be performed on systems of constraints such as projecting variables, or enumerating solutions for some variables, the others being considered as parameters ...

Variables	Comments
α	array dimensions
β	local array dimensions
θ	template dimensions
ψ	processor dimensions
δ	block offsets (for distributions)
γ	cycle numbers (for distributions)
p	all processor variables ($p = \psi \cup \psi'$)
λ	cycle load-balancing variable
e	other variables
ψ_D	set of distributed dimensions
ψ_R	set of replicated dimensions
x_i	ith dimension of x
x'	corresponding target mapping variables
$x[']$	shorthand for x and x'

Figure 5.7: Variables

5.5.2 Detailed formalization

Declarations, HPF directives and local address translations are embedded into linear constraints, as suggested in [5, 61]. This gives a linear description of the data distribution and of the communication problem, *i.e.* the enumeration of the elements to be sent and received. This section presents the derivation of a system of linear constraints that exactly describes the array elements to be communicated, with their associated source and target processors, allowing code generation.

Figure 5.10 shows the declaration constraints for the objects involved in the source and target mappings of the running example. Lower and upper bounds are

Polyhedrons	Constraints
$\mathcal{D}$	declarations
$\mathcal{H}$	HPF-related
$\mathcal{L}$	local declarations
$\mathcal{B}$	load-balancing
$\mathcal{R}$	remapping
$\mathcal{E}$	elements
$\mathcal{P}$	processors

Figure 5.8: Polyhedrons

Notation	Meaning
$\mathcal{X}(V)$	linear system on variables V
$\mathcal{X}_{\|V}$	system after projection of variables V
$Z \in \mathcal{X}[W]$	$\mathcal{X}$ solution enumeration parametrized by W
$\mathcal{X}_3 = \mathcal{X}_1 \cup \mathcal{X}_2$	union of systems
	(*i.e.* intersection of polyhedrons ...)
$\mathcal{X}_3 = \mathcal{X}_1 \times \mathcal{X}_2$	disjoint union
	(*i.e.* union on disjoint set of variables)

Figure 5.9: Polyhedron operators

defined for each dimension of arrays, templates and processor grids. Figure 5.11 shows the constraints derived from HPF directives. The sets of distributed and replicated dimensions are also shown. This linear modelization is extensivelly described in [5]. Figure 5.12 presents the local declarations and global to local address translations generated by HPFC, expressed through linear constraints. Thus they are directly included in our compilation scheme. However such an integration is not required: providing *global to local* address translation functions would be sufficient, although more expensive at run time.

$$\begin{array}{rl}
\texttt{A}(\alpha_1) & 1 \leq \alpha_1 \leq 20 \\
\text{source template } \texttt{T}(\theta_1,\theta_2,\theta_3) & 1 \leq \theta_1 \leq 2,\ \ 1 \leq \theta_2 \leq 20,\ \ 1 \leq \theta_3 \leq 2 \\
\text{source processors } \texttt{Ps}(\psi_1,\psi_2,\psi_3) & 1 \leq \psi_1 \leq 2,\ \ 1 \leq \psi_2 \leq 2,\ \ 1 \leq \psi_3 \leq 2 \\
\text{target template } \texttt{T}(\theta'_1,\theta'_2,\theta'_3) & 1 \leq \theta'_1 \leq 2,\ \ 1 \leq \theta'_2 \leq 20,\ \ 1 \leq \theta'_3 \leq 2 \\
\text{target processors } \texttt{Pt}(\psi'_1,\psi'_2) & 1 \leq \psi'_1 \leq 5,\ \ 1 \leq \psi'_2 \leq 2
\end{array}$$

Figure 5.10: declaration constraints $\mathcal{D}(\alpha, \theta['], \psi['])$

Let us now gather all these constraints in the remapping system (Definition 1). They define a polyhedron on α, β, ψ, δ ... and corresponding primed variables[1]. Solutions to this polyhedron link the array elements α and their mapping on the source ψ and target ψ' processors. $\mathcal{R}$ satisfies some properties because of the HPF

[1] Some variables, as θ, are of no interest for the code generation and can be exactly eliminated, reducing the size of the system without loss of generality nor precision.

`align A(i) with T(*,i,*)`	$\theta_2 = \alpha_1$
idem for target mapping	$\theta'_2 = \alpha_1$
distribution of `A` *onto* `Ps` *thru* `T`	$\theta_2 = 10\psi_2 + \delta_2 - 9, \quad 0 \leq \delta_2 < 10$
distribution of `A` *onto* `Pt` *thru* `T`	$\theta'_2 = 10\gamma'_1 + 2\psi'_1 + \delta'_1 - 1, \quad 0 \leq \delta'_1 < 2$

$$\psi_D = \{\psi_2\} \quad \psi_R = \{\psi_1, \psi_3\}$$
$$\psi'_D = \{\psi'_1\} \quad \psi'_R = \{\psi'_2\}$$

Figure 5.11: HPF-related constraints $\mathcal{H}(\alpha, \theta['], \psi['], \gamma['], \delta['])$ and dimension sets

local source array `As`(β_1)	$\beta_1 = \delta_2 + 1, \ 1 \leq \beta_1 \leq 10$
local target array `At`(β'_1)	$\beta'_1 = 2\gamma'_1 + \delta'_1 + 1, \ 1 \leq \beta'_1 \leq 4$

Figure 5.12: local declaration constraints $\mathcal{L}(\beta['], \delta['], \gamma['], \alpha)$

mapping semantics.

Definition 1 (remapping system $\mathcal{R}$)

$$\mathcal{R}(p, e) = \mathcal{R}(\psi['], \alpha, \beta['], \ldots) = \mathcal{D}(\ldots) \cup \mathcal{H}(\ldots) \cup \mathcal{L}(\ldots)$$

with $p = \psi \cup \psi'$ the source and target processor variables, $e = \alpha \cup \ldots$ the other variables.

Proposition 1 (replication independence) *Processor variables on replicated dimensions are disjoint from others in $\mathcal{R}$ with $p = p_R \cup p_D$:*

$$\mathcal{R}(p, e) = \mathcal{R}_{|p_R}(p_D, e) \times \mathcal{D}(p_R) = \mathcal{R}_{|p_R}(p_D, e) \times \mathcal{D}(\psi_R) \times \mathcal{D}(\psi'_R)$$

Proof: p_R variables appear neither in $\mathcal{H}$ nor in $\mathcal{L}$, and are disjoint in $\mathcal{D}$. $\mathcal{D}(x)$ is simply the cartesian declaration or replication constraints on x variables. □

Proposition 2 (disjoint distribution) *Array elements appear once in $\mathcal{R}_{|p_R}$:*

$$\forall \alpha \in \mathcal{D}(\alpha), \exists!(e, p_D) \text{ with } e = \alpha \cup \ldots | (e, p_D) \in \mathcal{R}_{|p_R}(p_D, e)$$

i.e. *apart from replicated dimensions, only one processor owns a data on the source and target processor grids, thus constraining the possible communications.*

Proof: HPF mapping semantics. This property is also true for skewed alignments. □

A SPMD code must be generated from such a polyhedron linking the array elements to their corresponding source and target processors. However, in the general case, because of data replication, $\mathcal{R}$ is not constrained enough for attributing *one* source to a target processor for a given needed array element. Indeed, $\mathcal{R}_{|p_R}$ assigns exactly one source to a target as shown in Proposition 2, but p_R variables are still free (Proposition 1). The underconstrained system allows choices to be made in the code generation. On the target side, replication provides an opportunity for broadcasts. On the source side, it allows to balance the load of generating and sending the messages.

Broadcasts

In the target processor grid, different processors on the replicated dimensions own the same data set. Thus they must somehow receive the same data. Let us decide that the very same messages will be broadcasted to replicated target processors from the source processors. From the communication point of view, replicated target processors are seen as *one abstract* processor to be sent a message. From the polyhedron point of view, ψ'_R dimensions are collapsed for message generation. The free choice on ψ'_R variables is removed, since the decision implies that the source processor choice is independent of these variables! For the running example, $\psi'_R = \{\psi'_2\}$ and $\mathcal{D}(\psi'_2) = \{1 \leq \psi'_2 \leq 2\}$, thus messages are broadcasted on `Pt`'s second dimension as shown in Figure 5.2.

Load balancing

Now one sender among the possible ones (ψ_R) must be chosen, as suggested in Figure 5.2. This choice must be independent of the replicated target processors, because of the broadcast decision. Moreover, in order to minimize the number of messages by sending elements in batches, it should not depend on the array element to be communicated. Thus the only possible action is to link the *abstract* target processors ψ'_D to ψ_R. These processors wait for disjoint data sets (Proposition 2) that can be provided by any source replicated processors (Proposition 1).

To assign ψ'_D to ψ_R in a balanced way, the basic idea is to attribute cyclically distributed target to replicated source processor dimensions. This cyclic distribution must involve processors seen as vectors on both side. In order to obtain this view of ψ'_D and ψ_R, a linearization is required to associate a single identifier to a set of indices.

The rationale for the linearization is to get rid of the dimension structuration in order to balance the cyclic distribution from all available source replicated processors onto all target distributed processors. Source processors that own the same elements are attributed a unique identifier through $\mathsf{linearize}(\psi_R)$, as well as target processors requiring different elements through $\mathsf{linearize}(\psi'_D)$.

Definition 2 (extent) *Let V be a set of bounded variables. The* extent *operator is the number of distinct elements, and can be defined recursively as:* $\mathsf{extent}(\emptyset) = 1$; $\mathsf{extent}(v) = (\max(v) - \min(v) + 1)$ *and* $\mathsf{extent}(V) = \prod_{v \in V} \mathsf{extent}(v)$.

Definition 3 (linearization) *Let V be a set of bounded variables. The linearization function* linearize *is defined recursively as:* $\mathsf{linearize}(\emptyset) = 0$ *and* $\mathsf{linearize}(V) = \mathsf{extent}(v) \times \mathsf{linearize}(V - \{v\}) + v - \min(v)$.

The following constraint expresses the cyclic distribution of distributed target to replicated source processor dimensions. It introduces a new cycle number variable λ.

Definition 4 (load balance $\mathcal{B}$)

$$\mathcal{B}(\psi_R, \psi'_D, \lambda) = \{\mathsf{linearize}(\psi'_D) = \mathsf{extent}(\psi_R) \times \lambda + \mathsf{linearize}(\psi_R)\}$$

For the running example, linearization of $\psi_R = \{\psi_1, \psi_3\}$ where $1 \leq \psi_1 \leq 2$, $1 \leq \psi_3 \leq 2$ leads to $\mathsf{linearize}(\psi_R) = 2\psi_3 + \psi_1 - 3$, $\psi'_D = \{\psi'_1\}$ with $1 \leq \psi'_1 \leq 5$ leads to $\mathsf{linearize}(\psi'_D) = \psi'_1 - 1$ thus $\mathcal{B}$ is $\psi'_1 - 1 = 4\lambda + 2\psi_3 + \psi_1 - 3$. The resulting assignment is shown in Figure 5.2 and 5.13. Because there are 5 targets and 4 available sources, the distribution cycles around the sources, and the first source processor set gets 2 targets.

Target		Source			Cycle
ψ'_1	$\mathsf{linearize}(\psi'_D)$	ψ_1	ψ_3	$\mathsf{linearize}(\psi_R)$	λ
1	0	1	1	0	0
2	1	2	1	1	0
3	2	1	2	2	0
4	3	2	2	3	0
5	4	1	1	0	1

Figure 5.13: $\mathcal{B}$ target to source assignment for the running example

Proposition 3 (target assignment) *Target processors are assigned to* one *source processor among the replicated ones through $\mathcal{B}$:*

$$\forall \psi'_D, \exists! \psi_R \wedge \exists! \lambda | (\psi_R, \psi'_D, \lambda) \in \mathcal{B}$$

Proof: The linearization is dense. □

5.5.3 SPMD code generation

Let us now introduce the final polyhedron which integrates these choices and is used for the code generation:

Definition 5 (elements $\mathcal{E}$) *With $p = \psi \cup \psi' = \psi_D \cup \psi_R \cup \psi'_D \cup \psi'_R$:*

$$\mathcal{E}(p, e, \lambda) = \mathcal{R}(p, e) \cup \mathcal{B}(\psi_R, \psi'_D, \lambda)$$

Polyhedron $\mathcal{E}$ is constrained enough so that there is only one possible sender (Proposition 5) for a given piece of data to be sent to all target processors requiring it. Thus a precise communication code can be generated: If (α, ψ, ψ') is a solution to $\mathcal{E}$, then ψ must send α to ψ'. Indeed, this polyhedron has the following properties:

Proposition 4 (orthogonality of ψ'_R in $\mathcal{E}$)

$$\mathcal{E}(p, e, \lambda) = \mathcal{E}_{|\psi'_R}(p_D, \psi_R, e, \lambda) \times \mathcal{D}(\psi'_R)$$

Proof: Definition 5 and Proposition 1. □

Proposition 5 (one sender) *For a required data on a target processor, there is only one sender defined in $\mathcal{E}$, which is independent of the replicated target (ψ'_R):*

$$\forall (\psi'_D, \alpha) \in \mathcal{R}(p, e), \exists! \psi \mid (\psi, \psi'_D, \alpha) \in \mathcal{E}$$

Proof: Propositions 2, 3 and 4. □

Proposition 6 (aggregation) *If a target processor requires two different pieces of data that can be sent by the same processor, then there is only one such processor:*

$$\forall\psi', \forall\alpha_i, \forall\alpha_j \mid (\exists\psi_D, (\psi', \psi_D, \alpha_i) \in \mathcal{E} \wedge (\psi', \psi_D, \alpha_j) \in \mathcal{E}) \implies (\exists!\psi|(\psi', \psi, \alpha_i) \in \mathcal{E} \wedge (\psi', \psi, \alpha_j) \in \mathcal{E})$$

Proof: Propositions 2 and 3: $\psi = \psi_D \cup \psi_R$, and ψ_R choice in $\mathcal{B}$ is independent of α. □

If all processors must enumerate all the integer solutions to polyhedron $\mathcal{E}$, this is equivalent to the runtime resolution technique and is very inefficient. Moreover, it would be interesting to pack at once the data to be sent between two processors, in order to have only one buffer for message aggregation. Therefore some manipulations are needed to generate efficient code.

Firstly, replicated dimensions of target processors (ψ'_R) are extracted from $\mathcal{E}$ as allowed by Proposition 4. This information is only required for broadcasting the message to the target processors. $\mathcal{E}_{|\psi'_R}$ stores the remaining information.

Secondly, in order to first enumerate the couples of processors that must communicate, and then to generate the associated message, a *super*set of these communicating processors is derived:

Definition 6 (processors $\mathcal{P}$)

$$\mathcal{P}(\psi_D, \psi_R, \psi'_D, \lambda) = \mathcal{E}_{|\psi'_R, e}$$

This projection may not be exact[2].

5.5.4 Optimality proof

For a given remapping, a minimal number of messages, containing only the required data, is sent over the network:

Theorem 2 (only required data is sent) *For any HPF remapping, only required data is communicated.*

Proof: If the processors sets are disjoint on the real processors: $\mathcal{E}$ exactly describes the array elements and their mapping (derived from Proposition 2). Polyhedron scanning techniques *exactly* enumerate the elements in $\mathcal{E}$ and these elements *must* be communicated if the processors are disjoint. If they are not, the twin guards prevent any data to be sent to a processor that already holds it, preserving the property. □

Theorem 3 (minimum number of messages is sent) *For any HPF remapping, a minimal number of messages is sent over the network.*

[2] A projection may be exact or approximate [4, 210], that is the integer solution to the projection may always reflects, or not, an integer solution to the original polyhedron.

Proof: Only required data is communicated (Theorem 2), all possible aggregations are performed (Proposition 6) and empty messages are not sent. □

Theorem 4 (memory requirements) *The maximum amount of memory required per* HPF *processor for a remapping is:*

$$2 \times (\mathsf{memory}(\mathtt{As}) + \mathsf{memory}(\mathtt{At}))$$

Proof: Local arrays plus send and receive buffers may be allocated at the same time. The buffer sizes are bounded by the local array sizes because no more than owned is sent (even for broadcasts) and no more than needed is received (Theorem 2). □

5.5.5 Experimental conditions

This section presents the hardware and software environment used for the tests, the measurements and the experiments.

DEC Alpha farm: 16 DEC 3000 model 400 AXP (512KB cache, 133MHz Alpha 21064) 64 MB memory workstations linked with a 100Mb/s/link[3] FDDI crossbar. Non dedicated machines.

Compilers: DEC Fortran OSF/1 `f77` version 3.5 with "`-fast -O3 -u`" options. DEC C OSF/1 `cc` with "`-O4`" option (for parts of the hpfc runtime library). DEC HPF `f90` version FT1.2 with "`-fast -wsf` n" options for comparison with the remapping codes generated by HPFC, our prototype HPF compiler.

Communications: PVM version 3.3.9 standard installation, used with direct route option and raw data encoding. `PVMBUFSIZE` not changed. 1 MB intermediate buffer used to avoid packing each array element through PVM.

Transposition: A square matrix transposition was tested for various matrix sizes[4], processor arrangements and distributions. The time to complete `A=TRANSPOSE(B)` with `A` and `B` initially aligned was measured. It includes packing, sending, receiving and unpacking the data, plus performing the transposition. The code is shown in Figure 5.17. The 2D remapping *compilation* times ranged in 0.1 – 1s on a SUN SS10.

Measures: The figures present the best wall-clock execution time of at least 20 instances, after substraction of a measure overhead under-estimation. The starting time was taken between two global synchronizations. The final time was taken after an additional global synchronization. The reason for not presenting average measures is that the machine was not dedicated to our experiments, hence it was not easy to get sound results.

Raw measures for transpositions are displayed. Figure 5.14 and 5.15 show the transposition times for various matrix sizes and distributions. The column heads describe the distribution of the array dimensions: for instance `c5c7` stands for `(cyclic(5),cyclic(7))`. Figure 5.16 show the `(block,block)` transposition time for various array arrangements, involving up to 16 processors. These raw measures are analyzed in Section 5.4.2.

[3] 12.5 MB/s/link

[4] *i.e.* the number of lines and columns

size	bb	cc[a]	bc	c48c64	c5c7
256	0.0564	0.0623	0.0681	0.0681	0.1257
384	0.1218	0.1296	0.1238	0.1312	0.1471
512	0.2161	0.2175	0.2389	0.2405	0.3082
640	0.3287	0.3336	0.3912	0.4064	0.5792
768	0.7958	0.7783	0.5967	0.6108	0.8914
896	0.9168	0.9110	0.8183	0.8700	4.3824
1024	1.7519	1.8563	1.1189	1.1282	1.5577

Figure 5.14: Transposition time (seconds) on `P(2,2)`

size	bb	cc	bc	c48c64	c5c7
256	0.0368	0.1803	0.0417	0.0524	0.0963
384	0.0875	0.4243	0.1120	0.0993	0.1256
512	0.1525	0.7834	0.1871	0.1871	0.2203
640	0.2423	1.2648	0.3072	0.3257	0.3706
768	0.3731	2.4780	0.5058	0.4692	0.5804
896	0.4838	-	0.6962	0.7001	3.3175
1024	0.6425	-	0.9480	0.8562	1.2486
1152	1.2037	-	1.1852	1.0270	1.3355
1280	1.3784	-	1.4008	1.3139	1.9587
1408	1.9197	-	1.6181	1.8723	2.1964
1536	2.9675	-	2.3970	2.2199	2.5921

Figure 5.15: Transposition time (seconds) on `P(2,3)`

	P(2,3)		P(3,3)		P(3,4)		P(4,4)	
size	HPFC	DEC	HPFC	DEC	HPFC	DEC	HPFC	DEC
256	0.0368	0.0429	0.0372	0.0332	0.0257	0.0229	0.0411	0.0173
384	0.0875	0.0976	0.0698	0.0674	0.0433	0.0503	0.0593	0.0417
512	0.1525	0.1825	0.1215	0.1416	0.0892	0.0942	0.0909	0.0710
640	0.2423	0.3065	0.1947	0.2202	0.1272	0.1576	0.1007	0.1159
768	0.3731	0.4256	0.2757	0.2827	0.1868	0.2201	0.1495	0.1696
896	0.4838	0.5934	0.4383	0.4496	0.2512	0.3231	0.1911	0.2350
1024	0.6425	0.8575	0.4588	0.5970	0.3361	0.4279	0.2281	0.2911
1152	1.2037	1.0468	0.8086	0.7092	0.4128	0.5271	0.2900	0.3960
1280	1.3784	1.3036	0.9488	1.0020	0.5084	0.6749	0.3545	0.4913
1408	1.9197	1.5641	1.0522	1.1953	0.6295	0.8320	0.4312	0.5966
1536	2.9675	1.9561	1.8209	1.3236	1.0120	0.9819	0.8147	0.7187
1792	-	-	2.1088	2.0908	1.6474	1.4240	1.0011	1.0085
2048	-	-	3.3807	2.8654	2.1184	1.8043	1.8235	1.3663

Figure 5.16: Transposition time (seconds) for (block,block) distributions

[a] Invariant code motion and some code transformations where performed by hand for this distribution

```
      real*8 A(n,n), B(n,n)
chpf$ dynamic B
chpf$ template T(n,n)
chpf$ processors P(...)
chpf$ distribute T(...) onto P
chpf$ align with T:: A, B
      ...
c
c A = TRANSPOSE(B)
c
c first align A and B transpose
c
chpf$ realign B(i,j) with T(j,i)
c
c now the assignment, everything is local
c
chpf$ independent(j, i)
      do j=1, n
        do i=1, n
          A(i,j) = B(j,i)
        enddo
      enddo
c
c DONE
c
      ...
```

Figure 5.17: Remapping-based transpose code for HPFC

Quatrième partie

Implantation et expériences

Chapitre 1

Introduction

Résumé

Cette partie présente les réalisations effectuées dans le cadre de cette thèse. Il s'agit principalement de l'intégration au sein du paralléliseur PIPS d'un prototype de compilateur HPF appelé HPFC. Le chapitre 2 présente d'abord de manière générale l'environnement de développement proposé par PIPS et les mécanismes d'intégration de nouvelles phases. Le chapitre 3 décrit ensuite le compilateur HPFC, en insistant sur le langage accepté en entrée d'une part, et sur les optimisations implantées d'autre part. Le chapitre 4 décrit enfin quelques expériences effectuées avec les codes compilés sur réseau de stations de travail.

Abstract

This part presents implementations performed during our PhD, namely the integration within the PIPS parallelizer of HPFC, our prototype HPF compiler. Chapter 2 first presents PIPS as a workbench, and how new phases can be integrated. Chapter 3 then focuses on our compiler, its capabilites and optimizations. Finally Chapter 4 describes experiments performed with our generated codes on a network of workstations.

1.1 L'environnement de développement PIPS

L'environnement de développement proposé dans PIPS est décrit au chapitre 2. Cet environnement est le fruit d'un travail collectif. Il part d'une très bonne conception initiale, due à Rémi TRIOLET, François IRIGOIN et Pierre JOUVELOT, qui a été complétée et étendue au cours du temps par de nombreux intervenants. La rédaction de la présentation générale de PIPS incluse ici est principalement l'œuvre de Ronan KERYELL, avec la participation de Corinne ANCOURT, Béatrice CREUSILLET, François IRIGOIN, Pierre JOUVELOT et moi-même.

PIPS vise à implanter rapidement de nouvelles analyses interprocédurales et transformations de programmes. Les nouvelles implantations profitent de la synergie d'autres analyses déjà disponibles, notamment d'analyses sémantiques qui fournissent une abstraction du comportement des programmes.

1.1.1 Survol de PIPS

La section 2.2 survole la structure de PIPS et présente ses possibilités actuelles. PIPS est un compilateur source à source pour Fortran 77. Il accepte en entrée ce langage, mais aussi des extensions comme les directives HPF. Au sortir d'analyses et de transformations, il produit du Fortran parallèle avec l'ajout de directives (extensions Cray, HPF, etc.), suggère des placements de données (CMF, CRAFT) ou encore produit du Fortran à passage de messages à partir de HPF.

De nombreuses analyses sont implantées et disponibles : les effets abstraient les données référencées par le programme ; les *transformeurs* décrivent l'exécution des *statements* ; leur calcul permet de composer ensuite des préconditions portant sur les variables scalaires d'un programme ; les régions de tableaux donnent une description précise des éléments de tableaux référencés par le programme, complétant les effets ; PIPS calcule le graphe de dépendances ; enfin il dérive la complexité symbolique d'un programme.

Ces analyses sont le support de transformations et de générations de codes. Des transformations standard sont proposées (*loop unrolling, strip-mining, interchange*, évaluation partielle etc.) ; d'autres moins fréquentes telles la privatisation de tableau ou des éliminations de code mort à base de préconditions et de chaînes *use-def*. La parallélisation interprocédurale était l'objectif initial du compilateur PIPS. Elle est basée sur l'algorithme de Kennedy et Allen. Du code visant des machines à mémoire répartie est produit par la méthode polyédrale ; enfin PIPS génère du code à passage de messages pour HPF (HPFC), ou pour un modèle de machine à bancs mémoires « intelligents » (WP65).

1.1.2 Conception générale de PIPS

La section 2.3 décrit la structure générale du compilateur. Un programme est analysé globalement par PIPS à l'intérieur d'un espace de travail *(workspace)* dans lequel sont mis toutes les informations calculées. Une base de donnée stocke les *resources* produites (PipsDBM). Un gestionnaire de dépendances *à la make* (PipsMake) organise la cohérence des différentes *phases* de PIPS et assure le fonctionnement du compilateur. PipsMake gère automatiquement les dépendances interprocédurales ascendantes ou descendantes grâce à une interface déclarative. Enfin le comportement du compilateur sur certains points précis peut être modifié en changeant les *properties* de PIPS.

1.1.3 Environnement de développement

La section 2.4 décrit plus précisément l'environnement de programmation offert pour implanter de nouvelles phases au sein du compilateur. Cet environnement inclut l'outil de génie logiciel NewGen. Il permet une déclaration simple et de haut niveau de structures de données. À partir de ces déclarations il génère des constructeurs, destructeurs et observateurs automatiquement. Il gère également la persistance des données (à savoir la possibilité d'écrire et de relire une structure arbitraire sur disque). Il fournit enfin un itérateur général qui autorise des parcours arbitraires de ces structures, en exécutant des opérations sur les nœuds de certains types. Cet outil est complété par une bibliothèque d'algèbre linéaire qui est la base mathématique des analyses et transformations les plus avancées : cette bibliothèque permet notamment de manipuler des polyèdres. La documentation et le débogage sont également discutés plus avant à cette section.

1.1.4 Interfaces utilisateur

La section 2.5 présente les interfaces utilisateur disponibles : shell (`Pips`), en ligne (`tpips`) et graphiques (`wpips` et `epips`). L'interface `epips` utilise l'éditeur de texte Emacs pour afficher les fichiers, et profite ainsi du prettyprint couleur implanté pour les programmes Fortran. Ces interfaces couvrent divers besoins. L'interface shell est souvent utilisée pour le développement. L'interface en ligne est plutôt utilisée pour la validation et l'interface graphique pour les démonstrations et les essais. Un outil permet de transformer automatiquement le *log* des actions demandées à l'interface graphique en un script équivalent qui peut être exécuté par `tpips` pour la validation ou bien pour reproduire une erreur.

1.1.5 Conclusion

La section 2.6 conclut cette présentation de PIPS. Elle décrit les projets comparables, en insistant sur ce qui fait l'originalité de PIPS : les analyses sémantiques basées sur l'algèbre linéaire, la gestion automatique de l'interprocédural et de la cohérence des analyses, l'utilisation de la persistance des données, la disponibilité de plusieurs interfaces couvrant des besoins différents, et son caractère multicibles (machines à mémoire partagée ou répartie, parallélisation ou analyses de programmes etc.).

1.2 Le prototype de compilateur HPFC

Les travaux théoriques de cette thèse ont donné lieu à des réalisations pratiques. Un prototype de compilateur pour le langage HPF a été développé. Il est présenté au chapitre 3. Le prototype a été implanté au sein de PIPS. Il réutilise certaines des analyses pour améliorer le code généré. Les codes produits ont tourné sur réseau de stations de travail, sur CM5 et sur ferme d'Alphas.

1.2.1 Entrée du compilateur

Nous détaillons le langage d'entrée du compilateur à la section 3.2. Il s'agit des principales extensions HPF ajoutées à Fortran 77. Le compilateur supporte également des extensions propres, préfixées par `fcd` au lieu de `hpf`.

Toutes les directives de placement de HPF, statiques et dynamiques, sont analysées et prises en compte, mis à part quelques détails syntaxiques. Par exemple, il n'y a pas pour l'instant d'alignement basé sur la notation des sections de tableaux. Le compilateur analyse également quelques directives simples avec la syntaxe dite libre de Fortran 90. Les directives de parallélisme `independent`, `new` et `reduction` sont prises en compte, ainsi que quelques formes très simples de `forall`. Les réductions sont reconnues par leur nom avec une syntaxe adaptée a Fortran 77. Enfin une directive `pure` remplace l'attribut Fortran 90 correspondant. Ces directives sont compatibles ou proches de la spécification du langage, tant en ce qui concerne l'esprit que la syntaxe.

Nous avons ajouté de nouvelles directives qui ne trouvent pas (encore) leur contrepartie dans le langage. Elles correspondent à des besoins rencontrés pour tester le compilateur, pour des optimisations particulières ou pour compiler efficacement des programmes réels. Une routine peut être déclarée `io`, exprimant qu'elle exécute une entrée-sortie ; elle est alors exécutée par l'hôte uniquement, après récupération des données nécessaires, et avant une mise à jour des données modifiées. Des sections de codes peuvent être gérées de la même façon entre `host` et `end host`. La notion de *scope* proposée au chapitre II.2 est implantée pour les variables privées avec `local` et `end local`. Des sections de codes éventuellement nichées peuvent être instrumentées automatiquement pour mesurer les temps d'exécution globaux avec `time` et `end time`. La directive `synchro` demande une synchronisation globale et permet une meilleure interactivité (par exemple afficher des messages au fur et à mesure de l'exécution du programme). Les valeurs d'un tableau peuvent être *tuées* avec `kill`, ce qui évite des communications en cas de replacement. Enfin les options internes de compilation de HPFC peuvent être modifiées avec la commande `set`, qui a été ajoutée pour valider ces options.

1.2.2 Sortie du compilateur

La section 3.3 décrit le modèle cible du compilateur et les optimisations implantées pour améliorer ce code. Elle présente l'exécutif utilisé par le code généré ainsi que les outils de conduite du compilateur *(drivers)*.

HPFC produit du code à passage de messages. Un code simple est produit pour l'hôte, qui assure principalement les fonctions d'entrées-sorties. Un code SPMD (un programme paramétré par l'identité des processeurs) est produit pour les nœuds de la machine. La gestion des processus et des communications est basé sur la librairie du domaine public PVM. Le code s'appuie directement sur un exécutif qui assure des fonctions de haut niveau (description des objets HPF, calculs de réductions, décalage de parties de tableau, etc.), mais certaines parties comme la compilation des entrées-sorties et des replacements génèrent directement du code PVM.

Le compilateur implante un certain nombre de techniques de compilation standard mais aussi d'autres plus avancées. La mémoire allouée sur chaque nœud pour stocker un tableau réparti est réduite et utilise un nouvel adressage. Les réductions reconnues font appel à des fonctions spécifiques de l'exécutif. Une analyse des recouvrements est faite pour les boucles indépendantes rectangulaires accédant à des données distribuées par blocs : le code produit vectorise les communications à l'extérieur du nid de boucles et utilise un stockage sur les bords du tableau. Le cas particulier des déplacements transversaux *(shift)* de parties de tableaux distribuées par blocs est géré spécifiquement, ainsi que les simples copies de tableaux parfaitement alignés. Enfin les communications liées aux entrées-sorties et aux replacements font l'objet des optimisations décrites à la partie III.

La compilation est commandée par le shell-script `hpfc`, qui lance directement PIPS, et poursuit la compilation et l'édition des liens des codes produits avec les librairies de l'exécutif, pour produire un code prêt à être exécuté. Le processus de compilation a été implanté de manière à éviter la modification de la phase assez fragile d'analyse syntaxique de PIPS. Les codes HPF sont d'abord filtrés pour transformer les directives en appels de routines Fortran 77 qui sont analysés syntaxiquement. La représentation intermédiaire est ensuite nettoyée de ces appels, tout en mettant à jour les structures de données internes qui décrivent les données et leur placement. Le code est ensuite analysé par PIPS, puis compilé effectivement vers du passage de messages, routine par routine.

Le compilateur génère plusieurs fichiers pour chaque routine : les codes de l'hôte et des nœuds d'une part, mais aussi des routines d'initialisation des structures de données de l'exécutif pour décrire le placement des données. Après les routines du programme, les *commons* Fortran sont compilés. Le caractère interprocédural de la compilation est nécessaire pour mettre à jour les recouvrements nécessaires aux tableaux répartis communs.

La section 3.4 conclut cette présentation en référençant les travaux comparables, *i.e.* les compilateurs générant du code pour machines à mémoire répartie.

1.3 Expériences sur réseau de stations de travail

Le chapitre 4 décrit et analyse quelques expériences faîtes sur réseau de stations de travail avec les codes générés par le compilateur. De simples itérations de relaxation d'une matrice ont été testées sur un réseau local Ethernet avec des stations de travail Sun Sparc 1. Le compilateur est d'abord décrit à la section 4.2. Le code testé est essentiellement sensible à l'analyse des recouvrements, ce qui était l'objectif de ce jeu de test.

Les expériences sont ensuite présentées à la section 4.3. Il s'agit d'une relaxation sur une grille à deux dimensions. Les tableaux sont répartis par blocs sur chacune des dimensions. Le programme a été testé sur 1, 4 et 8 processeurs. Nous nous sommes attachés à comparer précisément l'accélération obtenue en se référant aux performances du code en séquentiel sur un seul processeur. Les résultats sont donnés en pourcentage des performances qu'on pourrait obtenir en extrapolant les résultats sur un processeur au nombre de processeurs utilisés.

La section 4.4 analyse ces résultats en construisant un petit modèle prenant en compte le nombre de processeurs, les vitesses de communication et de calcul et la taille des matrices. L'efficacité pour un nombre de processeur p, une taille de matrice n et un rapport vitesse de calculs sur vitesse de communications μ est ensuite dérivé. L'adéquation du modèle et des résultats expérimentaux est très bonne. Le modèle est ensuite utilisé pour prévoir un nombre de processeurs optimal ainsi que la taille de matrice nécessaire pour atteindre une efficacité donnée. Les résultats montrent que pour des machines un peu rapides comme de RS6k (IBM) sur Ethernet, on arrive très vite à saturer le réseau : un réseau local de stations de travail n'est pas un supercalculateur qui s'ignore.

Chapitre 2

PIPS: a Workbench for Interprocedural Compilers

Ronan Keryell, Corinne Ancourt, Fabien Coelho,
Béatrice Creusillet, François Irigoin et Pierre Jouvelot

Cette présentation de PIPS constitue le rapport interne EMP CRI A-289 [156].

Résumé

PIPS est un outil expérimental pour implanter et évaluer des techniques interprocédurales de compilation, de parallélisation, d'analyses et d'optimisation. Cet article se concentre sur l'environnement de développement utilisé pour développer ces compilateurs. Sont inclues la gestion de structures de données, des dépendances entre phases d'analyses et de transformations, et les librairies mathématiques utilisées pour implanter les analyses les plus avancées.

Abstract

PIPS is an experimental tool to implement and evaluate various interprocedural compilation, parallelization, analysis and optimization techniques. This paper focuses on the workbench used to build these compilers. It includes the management of data structures, of dependences between the various analysis and transformation phases, and the mathematical libraries used to implement some of the most advanced analyses.

2.1 Introduction

Detecting the maximum level of parallelism in sequential programs has been a major source of interest during the last decade. This process is of utmost importance when trying to cope with the increasing number of vector and parallel supercomputers and, perhaps unfortunately, the huge number of already existing "dusty deck" sequential programs. Nowadays, since the development of high-end parallel computers has not fulfilled the wildest hopes of entrepreneurial companies in the parallel supercomputing business, we can expect more down-to-earth developments of work-group parallel servers and workstations with few but powerful processors.

Together with this architectural evolution, three main directions in compiler developments have been pursued: compilers for sequential machines (with superscalar processors), parallelizers of sequential programs and compilers for explicit parallel programs. All these approaches benefit from deep global program analyses, such as interprocedural and semantical analyses, to perform optimizations, vectorization, parallelization, transformations, restructuring and reverse-engineering of programs. Since these approaches often require the same or share a common ground of analyses, it is interesting to factorize them out in a common development tool to get the benefit of code re-use and modular programming.

PIPS is such a highly modular workbench for implementing and assessing various interprocedural compilers, optimizers, parallelizers, vectorizers, restructurers, etc. without having to build a new compiler from scratch. It has been used in various compilation areas since its inception in 1988 as the PIPS project (Interprocedural Parallelization of Scientific Programs). PIPS has been developed through several research projects funded by DRET (French ARPA), CNRS (French NSF) and the European Union (ESPRIT programs). The initial design was made by Rémi TRIOLET, François IRIGOIN and Pierre JOUVELOT. It has proved good enough since then not to require any major change.

Our project aims at combining advanced interprocedural and semantical analyses [240] with a requirement for compilation speed. The mathematical foundations of the semantical analysis implemented in PIPS are linear programming and polyhedra theory. Despite such advanced techniques, PIPS is able to deal with real-life programs such as benchmarks provided by ONERA (French research institute in aeronautics), or the Perfect Club in quite reasonable time. An excerpt of a fluid dynamics code for a wing analysis (Figure 2.2) will be used as a running example in this paper.

PIPS is a multi-target parallelizer. It incorporates many code transformations and options to tune its features and phases. PIPS offers interesting solutions to solve programming design problems such as generating high-level data structures from specifications, co-generating their documentation, defining user interface description and configuration files, dealing with object persistence and resource dependences, being able to write parts of PIPS in various languages, etc.

This paper details how PIPS achieves these goals. We first present a general overview of PIPS in the next section. In Section 2.3 we describe the mechanisms that deal with PIPS data structures, their dependences and their interprocedural relationships. The general design is discussed in Section 2.3. The programming environment, including the data structures generator and the linear library, and the documentation process is presented in Section 2.4. At last, the different user interfaces in Section 2.5 are presented before the related work in Section 2.6.1.

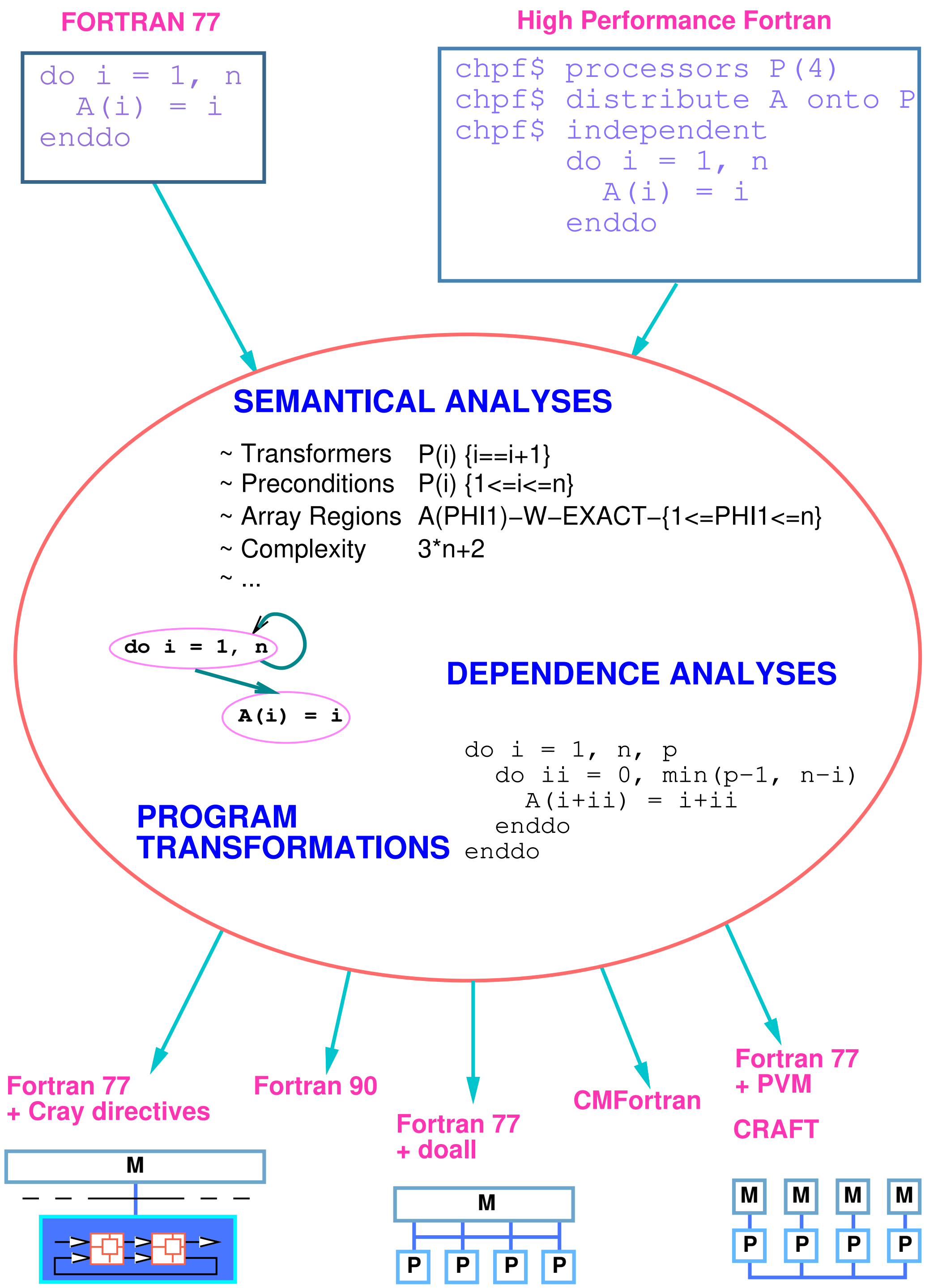


Figure 2.1: User view of the PIPS system.

2.2 Overview

PIPS is a source-to-source Fortran translator (Figure 2.1), Fortran still being the language of choice for most scientific applications. Beside standard Fortran 77, various dialects can be used as input (HPF) or output (CMF, Fortran 77 with Cray directives, Fortran 77 with HPF parallelism directives, etc.) to express parallelism or code/data distributions. Fortran can be seen here as a kind of portable language to run on various computers without having to deal with an assembly code back-end, although the PIPS infrastructure could be used as a basis to generate optimized assembly code. Other parsers or prettyprinters could be added and the internal representation could be extended if need be to cope with other imperative languages such as C.

Following the field research history, PIPS has first been used to improve the vectorization of Fortran code for parallel vector computers with shared memory (Fortran 77 with `DOALL`, Fortran 77 with Cray micro-tasking directives, Fortran 90). It is now mainly targeted at generating code for distributed memory machines, using different methods (processor and memory bank code for control distribution [8], CMF, CRAFT polyhedral method [207] or message-passing code from HPF [59]).

As we see in Section 2.3, the translation process is broken into smaller modular operations called phases in PIPS. For the user, these phases can be classified into various categories according to their usage. Since PIPS is an interprocedural environment, procedures and functions are very important in PIPS and are referred to as *module* in the sequel.

2.2.1 Analyses

An analysis phase computes some internal information that can be later used to parallelize (or generate) some code, some user information (such as a program complexity measure) to be later displayed by a prettyprinter, etc. The most interesting analyses available in PIPS are the semantical analyses described below.

Effects

Since an imperative language such as Fortran deals with memory locations, the basic semantical analyses in PIPS deal with the effects of instructions on memory. First, the *proper* effects are computed for each sub-statement of a module: a read or write operation on an array element, the update of an index variable in a `DO` loop, etc. These effects can be displayed to the user *via* a prettyprinter like:

```
C     <must be read   >: J NC S1 S2 S3
C     <must be written>: T(J,1,NC+5)
         T(J,1,NC+5) = S1*S2/((S3-S1)*(S3-S2))
```

Then, since the abstract syntax tree is recursively defined in PIPS, the *cumulated* effects are recursively aggregated from all the proper effects of the sub-statements in a statement. For the loop `300` on Figure 2.2 this leads to:

```
C     <may be read    >: J J2 JH K L NC S1 S2 S3 T(*,*,*)
C     <may be written >: JH S1 S2 S3 T(*,*,*)
```

```
c
      PROGRAM EXTRMAIN
c
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,...
      COMMON/CJ/J1,J2,JA,...
      COMMON/CK/K1,K2,...
      COMMON/CNI/L
      DATA N1,N4,N7,N17 /1,4,7,17/

      READ(NXYZ) I1,I2,J1,JA,K1,K2
      REWIND NXYZ
c
      IF(J1.GE.1.AND.K1.GE.1) THEN
         N4=4
         J1=J1+1
         J2=2*JA+1
         JA=JA+1
         K1=K1+1
         CALL EXTR(N7,N17)
         CALL EXTR(N4,N17)
         CALL EXTR(N1,N17)
      ENDIF
      END
c
c
c Compute the extrapolation coefficients
c for all the wing area (K=K1)
c
      SUBROUTINE EXTR(NI,NC)
c
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,...
      COMMON/CJ/J1,J2,JA,...
      COMMON/CK/K1,K2,...
      COMMON/CNI/L
      L=NI
      K=K1
      DO 300 J=J1,JA
         S1=D(J,K  ,J,K+1)
         S2=D(J,K+1,J,K+2)+S1
         S3=D(J,K+2,J,K+3)+S2
         T(J,1,NC+3)=S2*S3/((S1-S2)*(S1-S3))
         T(J,1,NC+4)=S3*S1/((S2-S3)*(S2-S1))
         T(J,1,NC+5)=S1*S2/((S3-S1)*(S3-S2))
         JH=J1+J2-J
         T(JH,1,NC+3)=T(J,1,NC+3)
         T(JH,1,NC+4)=T(J,1,NC+4)
         T(JH,1,NC+5)=T(J,1,NC+5)
300   CONTINUE
      END
c
c
c Compute D=DISTANCE
c
      REAL FUNCTION D(J,K,JP,KP)
c
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CNI/L
c
      D=SQRT((T(J,K,L  )-T(JP,KP,L  ))**2
     1      +(T(J,K,L+1)-T(JP,KP,L+1))**2
     2      +(T(J,K,L+2)-T(JP,KP,L+2))**2)
      END
```

Figure 2.2: Code excerpt from an ONERA benchmark.

```
C      <must be read   >: J1 JA
C      <must be written>: J
       DO 300 J = J1, JA
          ...
```

Since having information about effects on internal variables in a function is irrelevant to calling procedures, the *summary* effects of a procedure are constructed by removing these local effects from the cumulated effects of the procedure. Interprocedural propagation will be commented in Section 2.3.4.

Transformers

To improve the accuracy of the effect information, to provide more information for dependence testing and to select better program transformations, statements are labeled with preconditions that express constraints on integer scalar variables. Since such variables are often used in array references and loop bounds, preconditions can often precisely define which subpart of an array is referenced by a memory reference expression, thus refining cumulated effects into Regions (Section 2.2.1).

Mathematically speaking, PIPS preconditions are computed via transformers that are abstract commands mapping a store to a set of stores [138, 137]. They are relations between the values of integer variables in an initial store and those in a final store. In PIPS, the considered relations are polyhedra, represented by systems of linear equalities and inequalities, because they provide a sound and general mathematical framework for reasoning about array accesses which are mostly linear in practice. Below, two statement transformers are given by the prettyprinter.

```
C  T(J1) {J1==J1#init+1}
         J1 = J1+1
C  T(J2) {J2==2JA+1}
         J2 = 2*JA+1
```

Constant propagation, inductive variable detection, linear equality detection or general linear constraints computation can be performed in this workbench using different operators to deal with PIPS basic control structures. To deal with unstructured control graphs, some fancier operators requiring a fix point computation are needed, although they may slow down the analysis. To provide a better trade-off between time and precision to the user, options are available to tune the precondition construction algorithms.

Unlike most analysis in the field, we are dealing with abstract commands instead of abstract stores for two reasons. First, variables appearing in references cannot be aggregated to build cumulated and summary effects unless they denote the same values, that is they refer to the same store. The module's initial store is a natural candidate to be the unique reference store and, as a result, a relationship between this store and any statement's initial store is needed. This relationship is called a precondition [137] in PIPS and is constructed for a statement by aggregating all the transformers of the statements on the paths from the beginning of the module up to that statement. On the same example than the transformers we have:

```
C  P(J1,N) {N==1, 1<=J1, ...}
         J1 = J1+1
C  P(J1,N) {N==1, 2<=J1, ...}
         J2 = 2*JA+1
C  P(J1,J2,JA,N) {J2==2JA+1, N==1, 2<=J1, ...}
```

Second, dependence tests are performed between two statements, Although a relationship between each statement store would be more useful than two stores, each relative to one statement, this would not be realistic because too many predicates would have to be computed. Thus, a common reference to the same initial store seems to be a good trade-off between accuracy and complexity.

Dependence test

Since the dependence graph [255] is used within many parts of PIPS, such as the parallelizers, the validity check for some transformations, the use-def elimination phase, etc., a multi-precision test has been implemented to provide various degrees of precision and speed. First, a crude by fast algorithm is used and if no positive or negative conclusion is found, a more precise but more compute-intensive test is used, and so on [254].

PIPS dependence tests compute both dependence levels and dependence cones. Dependence cones can be retrieved to implement advanced tiling and scheduling methods.

Regions

Array Regions [76, 77] are used in PIPS to summarize accesses to array elements. READ and WRITE Regions represent the effects of statements and procedures on sets of array elements. They are used by the dependence test to disprove interprocedural dependences. For instance, in our example, the READ regions for the sole statement of function D are:

```
C       <T(φ1 , φ2 , φ3 )-R-EXACT-{ φ1 ==J,K<= φ2 <=K+1,L<= φ3 <=L+2}>

            D=SQRT((T(J,K,L  )-T(JP,KP,L  ))**2
     1      +       (T(J,K,L+1)-T(JP,KP,L+1))**2
     2      +       (T(J,K,L+2)-T(JP,KP,L+2))**2)
```

where the ϕ variables represent the three dimensions of A. To compute this Region, interprocedural preconditions must be used to discover the relations between the values of J and JP, and K and KP.

However, since READ and WRITE Regions do not represent array data flow, they are insufficient for advanced optimizations such as array privatization. IN and OUT regions have thus been introduced for that purpose: for any statement or procedure, IN regions contain its imported array elements, and OUT regions contain its exported array elements. The possible applications are numerous. Among others, IN and OUT regions are already used in PIPS to privatize array sections [16, 74], and we intend to use them for memory allocation when compiling signal processing specifications based on dynamic single assignments.

Another unique feature of PIPS array Regions lies in the fact that, although they are over-approximations of the element sets actually referenced, they are flagged as exact whenever possible. Beside a theoretical interest, there are specific applications such as the compilation of HPF in PIPS.

Array Data Flow Graph

This sophisticated data flow graph extends the basic DFG with some kind of array Regions information with linear predicates. Thus, it is able to finely track array elements accesses according to different parameters and loop indices in programs. However, this is only valid when the whole control can be represented in a linear framework [208, 207].

Complexities

When optimizing some code, or assessing whether to parallelize some program fragment or apply a transformation, it is quite useful to have some estimation of the program time complexity. A static complexity analysis phase has been added to PIPS to give polynomial complexity information about statements and modules [257] from preconditions and other semantical information.

2.2.2 Code generation phases

The result of program analyses and transformations is a new code. These phases generate (1) parallel code from data such as sequential code, semantical informations and the dependence graph, or (2) a message passing code for distributed memory architectures.

HPF Compiler

HPFC is a prototype HPF compiler implemented as a set of PIPS phases. From Fortran 77, HPF static (`align distribute processors template`), dynamic (`realign redistribute dynamic`) and parallelism (`independent new`) directives, it generates portable PVM-based SPMD codes which have run on various parallel architectures (SUN NOW [59], DEC alpha farm, TMC CM5, IBM SP2).

Implementing such a prototype within the PIPS framework has proven to be a fine advantage: First, the integration of the various phases (Figure 2.3) within the PipsMake system allows to reuse without new developments all PIPS analyses, and to benefit from the results of these analyses for better code generation and optimizations; Second, the Linear C^3 library has provided the mathematical framework [5, 6] and its implementation to develop new optimizations techniques targeted at generating efficient communication codes. It has been applied to I/O communications [61] and general remappings [68].

Phase	Function
`hpfc_filter`	preprocessing of the file
`hpfc_parser`	Fortran and HPF directive parser
`hpfc_init`	initialization of the compiler status
`hpfc_directives`	directive analysis
`hpfc_compile`	actual compilation of a module
`hpfc_close`	generation of global runtime parameters
`hpfc_install`	installation of the generated codes
`hpfc_make`	generation of executables
`hpfc_run`	execution of the program under PVM

Figure 2.3: HPFC phases in PIPS

Parallelization

The primary parallelization method used for shared-memory machines is based on Allen&Kennedy algorithm and on dependence levels. This algorithm was slightly modified to avoid loop distribution between statements using the same private variable(s). The resulting parallel code can be displayed in two different formats based on Fortran 77 and Fortran 90. Parallel loops are distinguished by the `DOALL` keyword.

Another parallelization algorithm was derived from Allen&Kennedy's to take into account the Cray specificities, the vector units and the micro-tasking library. Two levels of parallelism are selected at most. The inner parallelism is used to generate vector instructions and the outer one is used for task parallelism.

Code distribution

This parallelization method emulates a shared memory machine on a distributed architecture with programmable memory banks. It splits the program in two parts: a computational part and a memory part. More precisely, two programs are generated by using a method based on linear algebra: one SPMD program that does the computations in a block distributed control way and another SPMD program that manages the memory accesses.

The development run-time is implemented over PVM but the target is Inmos T9000 with C104 transputer-powered parallel machines with an hight speed interconnection network. This prototype was built as part of the European Puma project [8].

Parallelization with a polyhedral method

By using the Array Data Flow Graph, it is possible to track the movement of each value in a program where control flow can be statically represented in a linear way. Using this information, a schedule is generated to exploit the maximum parallelism and a placement is selected to reduce the number of communications according to their nature without discarding all the parallelism [208, 207].

2.2.3 Transformations

A transformation in PIPS takes an object, such as the internal representation of the code, and generates a new version of this object. Transformations currently implemented in PIPS are:

- loop distribution;
- scalar and array privatization based on Regions;
- atomizer to split statements in simpler ones such as `A=B op C`;
- loop unrolling;
- loop strip-mining;
- loop interchange using an unimodular transformation;
- loop normalization;
- partial evaluation;
- dead-code elimination based on preconditions;
- use-def elimination;
- control restructurer;
- reduction detection.

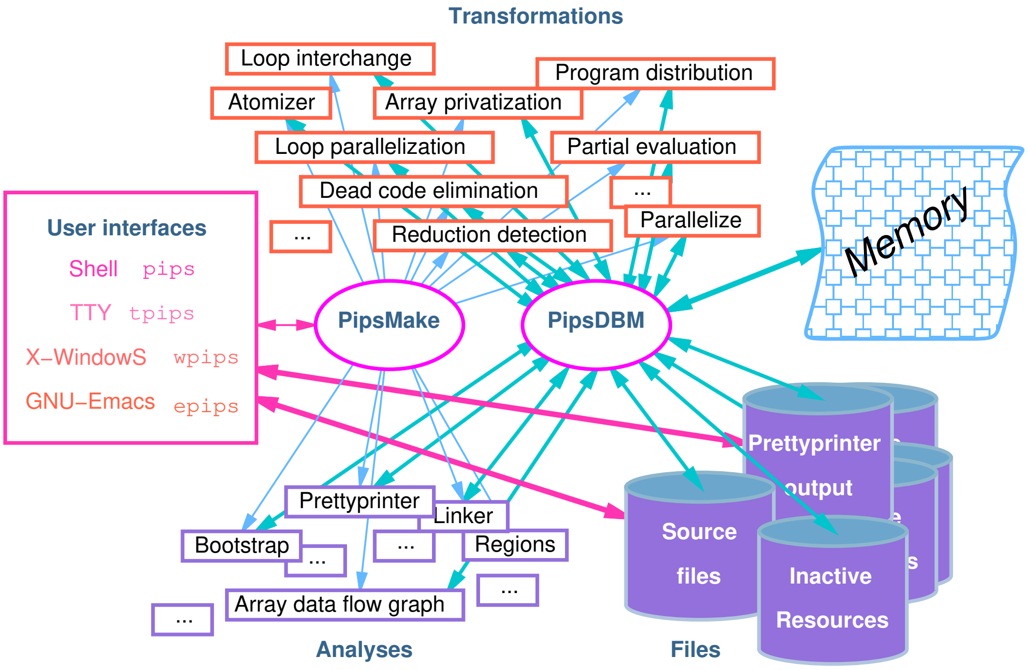


Figure 2.4: Overview of the PIPS architecture.

2.2.4 Pretty-printers

Pretty-printers are used to transform some PIPS internal representation into a human readable file that can be displayed by the user interface. There are pretty-printers to display the original parsed code, the sequential code, the parallel code in some parallel dialect, the dependence graph, the call graph, etc. Other specialized pretty-printers exist to add some informative decoration to the previous ones, such as displaying the code with regions, preconditions, complexities, etc. There are implemented by using hooks in the more classical pretty-printers.

2.3 PIPS general design

For a PIPS developer, PIPS can be viewed as in Figure 2.4. Of course the user interfaces control PIPS behavior but, internally, all the compiling, analyzing and transformation operations are split into basic independent *phases*. They are scheduled by PipsMake, an *à la* `make` interprocedural mechanism that deals with *resources* through the PipsDBM data base manager. An input program is first split into modules that correspond to its subroutines and functions. PipsMake can then apply the transformation phases on these modules according to some predefined rules. The result is a streamlined programming interface with as few side effects as possible between phases. This allows a highly modular programming style with several developers at the same time. To add a phase to PIPS, only a C function (which may be a wrapper if the execution part of the phase is written in another language) is added (which may of course call other functions) that is called by PipsMake with a module name as argument.

2.3.1 Resources

All the data structures that are used, produced or transformed by a phase in PIPS are called resources. A phase can request some data structures in PipsDBM, use or modify them, create some other resources, and then store with PipsDBM the modified or created resources to be used later. Examples of resources are the input code, abstract syntax tree, control flow graph, use-def chains, output code, semantic information such as *effects*, *preconditions*, *regions*, etc. These resources are described with the NewGen description language (Section 2.4.1) that allows transparent access to persistent data in PipsDBM from different languages.

2.3.2 Workspace

Since all the resources about a user code may be stored on disk (Section 2.3.5), resources are located in a workspace directory. This workspace is created from the source file given by the user and also contains a log file and a PipsMake data structure (Section 2.3.4 that encodes a coherent description of its content).

2.3.3 Phases

Each phase in PIPS use the same interface and simply is a C function taking a module name as argument and returning a boolean describing its completion status (success or failure). Figure 2.5 shows the example of the dead code elimination phase.

Since PIPS core is written in C, a wrapper function is needed to program PIPS in other programming languages. Currently, the only phase written in Common-Lisp is the reduction detection phase: it calls a Common-Lisp process to perform the job. Communications are handle through PipsDBM files.

2.3.4 PipsMake consistency manager

From a theoretical point of view, the object types and functions available in PIPS define an heterogeneous algebra with constructors (*e.g.* parsers), extractors (*e.g.* prettyprinters) and operators (*e.g.* loop unrolling). Very few combinations of functions make sense, but many functions and object types are available. This abundance is confusing for casual and even experienced users and it was deemed necessary to assist them by providing default computation rules and automatic consistency management.

The PipsMake library – not so far from the Unix make utility – is intended for interprocedural use. The objects it manages are resources stored in memory or/and on disk. The phases are described in a `pipsmake-rc` file by generic rules that use and produce resources. Some examples are shown on Figure 2.6. The ordering of the computation is demand-driven, dynamically and automatically deduced from the `pipsmake-rc` specification file. Since it is a quite original part of PIPS, it is interesting to describe a little more the features of PipsMake.

Rule format

For each rule, that is in fact the textual name of a phase C function, it is possible to add a list of constraints composed with a qualifier, an owner name and a resource

```
bool suppress_dead_code(string module_name)
{
  /* get the resources */
  statement module_stmt = (statement)
      db_get_memory_resource(DBR_CODE, module_name, TRUE);
  set_proper_effects_map((statement_mapping)
      db_get_memory_resource(DBR_PROPER_EFFECTS, module_name, TRUE));
  set_precondition_map((statement_mapping)
      db_get_memory_resource(DBR_PRECONDITIONS, module_name, TRUE));

  set_current_module_statement(module_stmt);
  set_current_module_entity(local_name_to_top_level_entity(module_name));

  debug_on("DEAD_CODE_DEBUG_LEVEL");
  /* really do the job here: */
  suppress_dead_code_statement(module_stmt);
  debug_off();

  /* returns the updated code to Pips DBM */
  DB_PUT_MEMORY_RESOURCE(DBR_CODE, module_name, module_stmt);
  reset_current_module_statement();
  reset_current_module_entity();
  reset_proper_effects_map();
  reset_precondition_map();

  return TRUE;
}
```

Figure 2.5: Excerpt of the function of the dead code elimination phase.

```
proper_effects    > MODULE.proper_effects
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects

cumulated_effects > MODULE.cumulated_effects
        < PROGRAM.entities
        < MODULE.code MODULE.proper_effects

summary_effects   > MODULE.summary_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects

hpfc_close          > PROGRAM.hpfc_commons
    ! SELECT.hpfc_parser
    ! SELECT.must_regions
    ! ALL.hpfc_static_directives
    ! ALL.hpfc_dynamic_directives
    < PROGRAM.entities
    < PROGRAM.hpfc_status
    < MAIN.hpfc_host

use_def_elimination > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.chains
```

Figure 2.6: Example of `pipsmake-rc` rules.

name. Each constraint has the following syntax:

$$< \mathit{qualifyer} > < \mathit{owner\text{-}name} > . < \mathit{resource\text{-}name} >$$

The qualifier describing the resource can be for each resource:

`<:` needed;

`>:` generated;

`#:` destroyed by the rule;

`=:` asserts that the resource remains valid even if other dependence rules would suggest to invalidate the resource;

`!:` asks for applying an other rule before executing the rule. Thus, this constraint has a special syntax since it composed with a rule instead of a resource name. In Figure 2.6 for instance `hpfc_close` needs an alternate parser for HPF, the must Regions and the directives of all the modules of the program.

This qualifier is thus more complex than an Unix `make` but allows finer tuning to avoid useless computation (for example with the =) and be able to expand the rule dependence behavior with !.

An owner name specifies the owner of the needed resource. For example for the code resource, only the code of this module, or the code of all the modules, etc:

MODULE: the current module itself;

MAIN: the main module if any (that is the Fortran `PROGRAM`);

ALL: all the modules

CALLEES: the set of the called modules by the current one; it allows to describe bottom-up analyses on the callgraph.

CALLERS: the set of the calling modules of the current one; it expresses top-down analyses.

PROGRAM: a resource that is attached to the whole program, such as the entities of a program. This concept is orthogonal to the module concept;

SELECT: select a rule by default.

A rule (that is a phase for a PIPS user) is conventionally called a transformation when a resource name appears in the rule with both $<$ and $>$ qualifiers. For example dead code elimination which transforms the code both needs the code and writes it.

It is possible to chose between several rules to make a resource. By default the first one found in `pipsmake-rc` is selected but the programmer can choose to activate an other default rule through the API (see Section 2.3.4 or with the **SELECT** owner). This feature is used for example in the user interfaces to select the kind of decorations in the prettyprinters. For instance the sequential code can be built by several phases, one for the plain code, another one for the plain code with Regions, etc.

Up to date resources & API

PipsMake is responsible for dynamically activating rules to build up to date resources and iterates through the transitive closure of the resource dependences (something Unix's `make` is unfortunately unable to do). A small programming interface is available to drive PipsMake from user interfaces (Section 2.5) or other phases:

- `make(string resource_name, string owner_name)` builds the resources `resource_name` for the set of modules or the program described by `owner_name`;
- `apply(string phase_name, string owner_name)` applies the phase `phase_name` to the set of modules or the program described by `owner_name`;
- `activate(string phase_name)` activates this rule as the new default one when more than one rule could be used to build some resources. Since with this new default rule these generated resources are no longer up to date, the resources that can generated by the new default rule and that existed before are recursively deleted.

Interprocedurality

Interprocedurality is dealt with by the PipsMake **CALLEES** and **CALLERS** owner features: A bottom-up algorithm on the interprocedural call graph such as effects or interprocedural transformers computation will have some rules using **CALLEES**, while a top-down algorithm such as interprocedural preconditions will have some rules using **CALLERS** so as to properly order the phase execution and propagate resources interprocedurally. The correct ordering of the computation is deduced from the `pipsmake-rc` file.

For instance, effects computation relies on the three dependence rules shown on Figure 2.6: First the proper effects are computed from the code and the summary effects of the **CALLEES** to get the summary effects of the modules called in the current module. Second these proper effects, that only relate to each simple statement, are transformed into the summary effects that take into account the hierarchical structure of the abstract syntax tree. Third, the summary effects are computed to describe the effects that are only non local to the current module and that may later be used to compute the effects of a caller.

Persistence and interruption

Since PipsMake itself is implemented with NewGen, the PipsMake data structure describing the state of PIPS is saved on disk when PIPS exits. Thus, a PIPS operation can continue from a previous saved state.

Furthermore, at a phase boundary, all PIPS data structures are managed with PipsDBM. Thus the PIPS process can be stopped cleanly at a phase boundary when the control is under PipsMake. Of course the `make()` operation asked to PipsMake fails, but next time the user asks for a `make()`, PipsMake will only launch the phases required to build the lacking resources. This mechanism is heavily used in the user interfaces to insure interactivity.

2.3.5 PipsDBM resource manager

All the data structures used by the phases to seamlessly communicate with each other are under the responsibility of the PipsDBM resource manager.

PipsDBM can decide to store a resource in memory or on disk. Typically, resources that have been modified are written on disk when PIPS exits but the programmer can ask to immediately create a file resource to be viewed by a user interface. The programmer can also declare some resources as unloadable or as memory-only.

PipsDBM also manages a logical date for each resource for the dependence mechanism of PipsMake. Logical dates have been introduced to overcome the lack of Unix time stamp precision (1 second under SunOS4) and the difficulty to have very well synchronized client-server operations in an NFS environment. Of course, the Unix date is also used in order to detect resource files modified by the user (typically an edit through a user interface) where the 1 second precision is good enough.

The programming interface is reduced to

```
string db_get_resource(string ressource_name, string module_name, bool pure)
void db_put_resource(string ressource_name, string module_name, void* value)
```

`pure` is used to specify if the programmer wants the genuine resource or only a copy (for example to modify it and give back another resource).

2.3.6 PIPS properties

Global variables to modify or finely tune PIPS behavior are quite useful but unfortunately are often dangerous. Thus these variables are wrapped in PIPS as properties. A property can be a boolean, an integer or a string. Using the fact that properties are centralized, they are initialized from a default property file and possibly from a local `property.rc`. Since the behavior is modified by the properties, they are stored in the workspace in order to restart a PIPS session later in the same way even if the global property file has been modified.

2.4 Programming environment

The PIPS workbench is implemented to form a programming environment. Typically, each phase has its own directory and its own library so that it is easy to add a new phase or modify a phase without too much trouble-shooting. Furthermore, almost all the directory structure is duplicated to have both a production and a development version at the same time.

2.4.1 NewGen: data structure and method generation

In oder to ease data structure portability between different languages and also to allow data persistence, a tool and a language to declare data structures has been developed: NewGen [143]. Once a data description of the *domains* (that are the data structures that can be defined in NewGen) is written, NewGen constructs several methods to create initialized or not data values, access or modify certain parts of constructed values, write or read data from files, or recursively deallocate data values.

A domain can be a sum (`+` like a C-union) or a product (`x` like a C-struct) of other domains, list (`*`), set (`{}`), array (`[]`) or a map (`->`) of domains. Some domains can be declared as "persistent" to avoid being deleted and the "tabulated" attribute associates a unique identifier to each domain element, allowing unique naming through files and different program runs. The mapping of domains is used to attach semantic information to objects: `alignmap = entity->align`.

Of course, some external types can be imported (such as for using the linear C^3 library in PIPS). The user must provide a set of functions for these domains to write, read, free and copy methods. The NewGen environment includes also various low-level classes such as lists, hash-tables, sets, stacks etc. with their various methods (iterators, etc.) that ease the writing of new phases.

```
expression = reference + range + call ;
reference = variable:entity x indices:expression* ;
range = lower:expression x upper:expression x increment:expression ;
call = function:entity x arguments:expression* ;
statement = label:entity x number:int x ordering:int x
            comments:string x instruction ;
instruction = block:statement* + test + loop + goto:statement +
              call + unstructured ;
test = condition:expression x true:statement x false:statement ;
```

Figure 2.7: Excerpt of the PIPS abstract syntax tree.

An excerpt of the internal representation for Fortran in PIPS is shown on Figure 2.7. For example, a `call` is a `function entity` with a list of `arguments expressions`. An `expression` can be a `reference`, a `range` or a `call`, and so on. In fact an assignment is represented itself by a call to a pseudo-intrinsic function "`=`" with 2 arguments, the left hand side and the right hand side argument. And so on for all the intrinsics of the Fortran language. All the syntactic sugar of Fortran has been removed from the internal representation and in this way it is expected not to stick on Fortran idiosyncrasies. The representation is quite simple: there are only 5 cases of statements and 3 cases of expressions. This simplicity also benefits all the algorithms present in PIPS since there are very few different cases to test.

An important feature of NewGen is the availability of a general multi domain iterator function (`gen_multi_recurse`). From any NewGen domain instance (e.g., `statement`, `instruction`, `expression`) one can start an iterator by providing the list of domains to be visited, and for each domain the function to be applied top-down, which returns a boolean telling whether to go on with the recursion or not, and a function to be applied bottom-up. The iterator is optimized so as to visit only the needed nodes, and not to follow paths on which no nodes of the required types might be encountered.

In the Figure 2.8 example, some entities are replaced with others. Entities may appear as variable references, as loop indices, as called functions and in program codes. The recursion starts from `statement stat`: on each `loop`, `reference`, `call` or `code` encountered top-down, the function `gen_true` is applied. This function does nothing but returning true, hence the recursion won't be stopped before having processed all the required domains that can be reached from `stat`. While going back bottom-up, the `replace` functions are applied and perform the required substitution. From the user point of view, only replace functions are to be written.

```
static void replace(entity * e)
{ /* replace entity (*e) if required... */}

static void replace_call(call c)
{ replace(&call_function(c));}

static void replace_loop(loop l)
{ replace(&loop_index(l));}

static void replace_reference(reference r)
{ replace(&reference_variable(r)); }

static replace_code(code c) // code_declarations() is a list of entity
{ MAPL(ce, replace(&ENTITY(CAR(ce))), code_declarations(c));}

// args: (obj, [domain, filter, rewrite,]* NULL);
   gen_multi_recurse(stat,
        reference_domain, gen_true, replace_reference,
        loop_domain,      gen_true, replace_loop,
        call_domain,      gen_true, replace_call,
        code_domain,      gen_true, replace_code,
                      NULL);
```

Figure 2.8: `gen_multi_recurse` example

An example of how NEWGEN defined types are used is shown on Figure 2.9. It is a follow-up of Figure 2.5. `suppress_dead_code_statement()` begins with a NEWGEN generic iterator to walk down the module statements applying `dead_statement_filter` on them and walk up applying `dead_statement_rewrite` on all the `statement` domains encountered. In `dead_statement_filter()` one can see that `statement_instruction()` is the accessor to the `instruction` of a `statement` generated by NEWGEN from the internal representation description. `instruction_tag()` describes the content of an `instruction` (which is a "union"). For loops the result is `is_instruction_loop`. NEWGEN also generates predicates such as `instruction_loop_p()` to test directly the content of an `instruction`.

NEWGEN can output data structure declarations and methods for C and Common-Lisp languages since they are the languages used to build PIPS. The translation environment is described in Section 2.4.4. Note that at the beginning of the PIPS project, C++ was not considered stable enough so that all the object

```
static bool dead_statement_filter(statement s)
{
   instruction i = statement_instruction(s);
   bool ret = TRUE;

   if (!statement_weakly_feasible_p(s))       // empty precondition
      ret = remove_dead_statement(s, i);
   else
   {
      switch (instruction_tag(i)) {
      case is_instruction_loop:
      {
         loop l = instruction_loop(i);
         if (dead_loop_p(l))                   // DO I=1,0
            ret = remove_dead_loop(s, i, l);
         else if (loop_executed_once_p(s, l)) { // DO I=N,N
            remove_loop_statement(s, i, l);
            suppress_dead_code_statement(body);
            ret = FALSE;
         }
         break;
      }
      case is_instruction_test:
         ret = dead_deal_with_test(s, instruction_test(i));
         break;
      case is_instruction_unstructured:
         dead_recurse_unstructured(instruction_unstructured(i));
         ret = FALSE;
         break;
      }
   }

   if (!ret) // Try to rewrite the code underneath
      dead_statement_rewrite(s);

   return ret;
}

void suppress_dead_code_statement(statement module_stmt)
{
    gen_recurse(module_stmt,            // recursion from...
                statement_domain,       // on statements
                dead_statement_filter,  // entry function (top-down)
                dead_statement_rewrite);// exit function (bottom-up)
}
```

Figure 2.9: Excerpt of the function of the dead code elimination phase.

methods have been written in C, wrapped and hidden in macros and functions.

2.4.2 Linear C^3 library

An important tool in PIPS is the Linear C^3 library that handles vectors, matrices, linear constraints and other structures based on these such as polyhedrons. The algorithms used are designed for integer and/or rational coefficients. This library is extensively used for analyses such as the dependence test, precondition and region computation, and for transformations, such as tiling, and code generation, such as send and receive code in HPF compilation or code control-distribution in WP65. The linear C^3 library is a joint project with IRISA and PRISM laboratories, partially funded by CNRS. IRISA contributed an implementation of CHERNIKOVA algorithm and PRISM a C implementation of PIP (Parametric Integer Programming).

2.4.3 Debugging support

Developing and maintaining such a project requires a lot of debugging support from the programming environment at many levels. At the higher level, it is nice to be able to replay a PIPS session that failed since a bug can appear after quite a long time. For this purpose there is a tool that transforms a WPIPS log file into a TPIPS command file to be re-executed later. Every night, a validation suite is run on hundreds of test cases to do some basic non-regression testing. It is not enough to do intensive debugging but is generally sufficient to ensure that a modification is acceptable.

At the programmer level there are some macros to enable or disable the debug mode in some parts of PIPS according to some environment variables that indicate the debug level. Often, there is such a variable per phase at least and the debug modes can be nested to avoid debugging all the functions used in a phase to debug only this phase for example. The debug levels can also be modified according to the property of the same names.

At the NEWGEN level, the recursive type coherence of a data structure can be verified with `gen_consistent_p()` or even at each assignment or at creation time by setting the variable `gen_debug` to an adequate value.

At last, some Emacs macros and key bindings have been defined for the `gdb` Emacs-mode to display the type of a NEWGEN object, the type of a PIPS entity, an expression of the PIPS abstract syntax tree, a PIPS statement, etc. to alleviate the development effort.

2.4.4 Documentation and configuration files

Such a big project (over 200,000 lines of code for the compilers, mathematical libraries and tools) with developers on different sites) needs to manage a strong documentation support of course, but also as much automatically generated and non-redundant as possible to keep the incoherence level low.

The main idea is to write all the configuration files as technical reports and automatically extract the configuration files from them. Figure 2.10 is a simplified synopsis of the documentation flow. C headers are generated from `.newgen` files that come from some document describing the usage of the data structure, such as `ri.tex` for the internal representation . The menu files for the user interfaces are

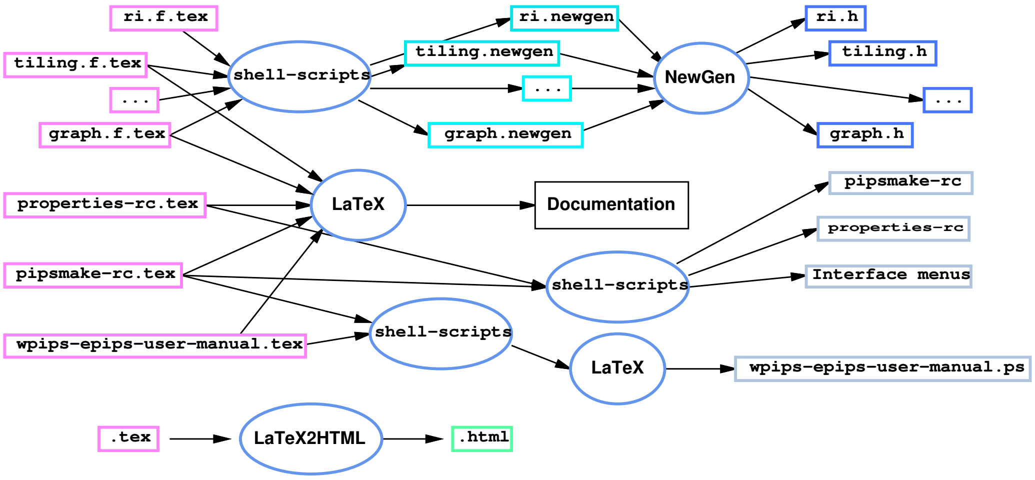


Figure 2.10: Documentation and configuration system.

generated from `pipsmake-rc` and the property file. All the information needed to feed PipsMake and PipsDBM comes from the `pipsmake-rc` file. At last, LaTeX2HTML is used to generate most of `http://www.cri.ensmp.fr/pips` from all the LaTeX files.

2.5 User interfaces

Typically, a user interface will use the `make()` command to build a resource to display, use `apply()` to apply a transformation on the code and select default rules, that are options, with `activate()` (Section 2.3.4). There are as many as four interfaces to PIPS:

- `pips` is a shell interface, can do only one operation on a workspace and relies a lot on PipsDBM persistence since the `pips` process exits after each operation;
- `tpips` is a line interface with automatic completion. It's much faster to use for experimentation and debugging than a full graphical interface;
- `wpips` is an XView based easy interface interface;
- `epips` is an extended version of the previous one for GNU-Emacs users. The interface inherits the Fortran mode and color highlight features. Further more graphs are displayed with `xtree` and `daVinci`.

All these interfaces are configured according to the documentation 2.4.4: when a programmer adds a transformation, it automatically appears in all the interfaces after reconfiguration of the `pipsmake-rc` file.

When dealing with big programs, it may be necessary to interrupt PIPS if a user chooses too costly an option or applies an heavy analysis by mistake. To solve this issue, the user interfaces can interrupt PipsMake at the phase interface level, without compromising the coherence through PipsMake and PipsDBM (Section 2.3.4).

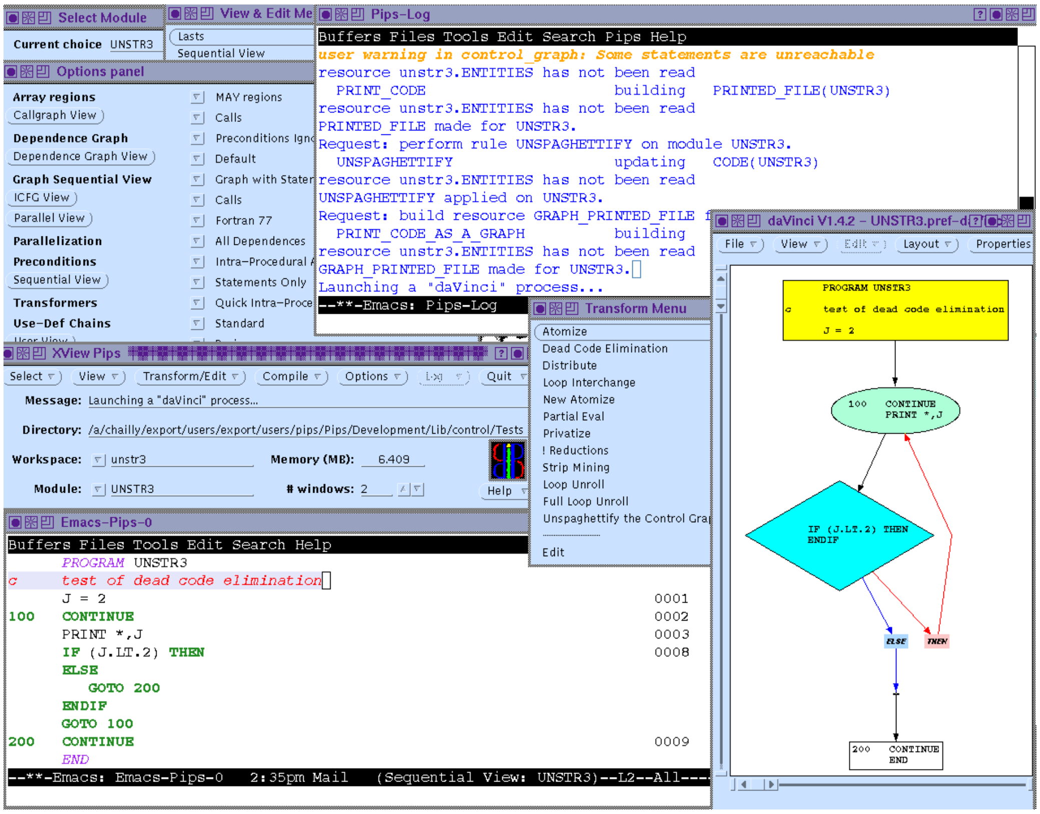


Figure 2.11: Emacs-PIPS interface snapshot.

2.6 Conclusion

2.6.1 Related Work

Many compiler prototypes have been implemented to test analysis, transformation or code generation techniques.

Parafrase 2 (UIUC): This compiler is a source to source code restructurer for Fortran and C performing analysis, transformation and code generation. It allows automatic vectorization and parallelization.

Polaris (UIUC): This compiler is also a source to source restructurer developed for Fortran 77. It performs interprocedural analysis for automatic parallelization of programs, targeting shared-memory architectures.

SUIF (Stanford): The *Stanford University Intermediate Form* is a low level representation plus some high level additions that keep track of the program structure such as loops, conditions and array references. The input language is C and Fortran through the `f2c` translator. The compiler outputs machine code for different architectures. The compiler initial design and the current distribution of SUIF does not incorporate interprocedural analysis. Such an addition is being worked on [122, 120]. Another problem is that the consistency of analyses is not automatically inforced (for instance invalidation after code transformations).

The originality of PIPS with respect to these other prototypes is the conjunction of:

- Interprocedurality and coherency automatic management in a demand driven system combining database (PipsDBM) and dependence (PipsMake) functions.
- Advanced interprocedural semantic and data-flow analyses already implemented, and their mathematical foundation upon linear algebra (Linear C^3 library).
- Software foundation upon an high level description system (NewGen) which automatically generates functions (access, constructor, destructor) and can iterate on arbitrary structures. An additional by-product is that the code look and feel is homogeneous over the 150, 000 lines of the compiler.
- Multi-target:
  - Automatic parallelization for shared-memory
  - Compilation for distributed-memory models (Hpfc, wp65)
  - Automatic data mapping
- The availability of different interfaces (shell, terminal Tpips, graphical Wpips).

Compiler development is a hard task. Here are examples of many research prototypes. They would benefit a lot from already available infrastructures, and of the synergy of other analyses and transformations, especially interprocedural ones.

- (small) program analyzers and parallelizers: Bouclette (LIP, ENS-Lyon), LooPo (U. Passau), Nascent (OGI), PAF (PRISM, UVSQ), Petit (U. of Maryland), Tiny.
- Parallel Fortran (HPF spirit) compilers: Adaptor (GMD), Annai (CSCS-NEC), D system (Rice U.), Fx (Carnegie Melon U.), Paradigme (UIUC, based on Parafrase 2), SHPF (Southhampton), VFCS (Vienna)
- Parallel C compilers: Pandore (Irisa), EPPP (CRI Montréal)

2.6.2 Remarks

Eight years after its inception, PIPS as a workbench is still alive and well. PIPS provides a robust infrastructure for new experiments in compilation, program analysis, optimization, transformation and parallelization. The PIPS developer environment is described in [139], but it also is possible to develop new phases on top of but *outside* of PIPS since all (in fact, most...) PIPS data structures can be reloaded using NEWGEN primitives.

PIPS can also be used as a reverse engineering tool. Region analyses provide useful summaries of procedure effects, while precondition-based partial evaluation and dead code elimination reduce code size.

PIPS may be less robust than other publicly available source-to-source compilers but the stress is put on the ability to quickly add new phases such as program transformations, semantic analyses, parsers, parallelizers, etc. or new user interfaces without spending time developing a cumbersome infrastructure.

2.6.3 Acknowledgment

We would like to thank many people that have contributed to the PIPS project: Bruno BARON, Denis BARTHOU, Pierre BERTHOMIER, Arnauld LESERVOT, Guillaume OGET, Alexis PLATONOFF, Rémi TRIOLET, Yi-Qing YANG, Lei ZHOU.

Chapitre 3

HPFC: a Prototype HPF Compiler

Cette présentation du prototype de compilateur HPFC *est celle proposée sur le serveur* Web *du centre (`http://www.cri.ensmp.fr/pips/hpfc.html`). Cette page est référencée par la page du Forum HPF.*

Résumé

HPFC est un prototype de compilateur HPF qui a été développé au cours de cette thèse. Implanter ou essayer d'implanter réellement des techniques de compilation donne la vraie mesure des problèmes que doivent gérer les développeurs, problèmes qui sont facilement minimisés ou oubliés lors d'un travail plus théorique de recherche. Ce chapitre expose les caractéristiques du compilateur : son langage d'entrée, son modèle cible, les optimisations implantées, les fonctions de support pour l'exécution et enfin l'interface utilisateur.

Abstract

HPFC is a prototype HPF compiler which has been developed as part of this thesis. Implementing or attempting to actually implement compilation techniques gives the real measure of the problems to be faced by implementors, that are easily overlooked with only theoretical paperwork. This chapter presents HPFC main features: The input language, the target programming model, the optimizations performed, the runtime support functions and the user interface.

3.1 Introduction

HPFC is an *High Performance Fortran* Compiler which is being developed at CRI. This project aims at building a prototype compiler to be able to test new optimization techniques for compiling HPF. It has been supported by PARADIGME [33]. The compiler is implemented on top of PIPS [138, 156, 155] *(Scientific Programs Interprocedural Parallelizer)* which provides many program analyses such as effects, regions, etc. Realistic codes (hundreds of lines, heavy I/Os, not fully optimizable) are compiled by the prototype and have run on a network of workstations and on a CM5. Some experiments were also performed on an alpha farm.

3.2 Compiler input language

The compiler does not take as input full HPF, but includes both simplification and extensions (FC directives).

3.2.1 Base language

The base language is mainly Fortran 77, plus some extensions toward HPF.

- The input language syntax is restricted to Fortran 77, and more precisely to its PIPS Fortran subset. Few features (computed `GOTO` and substrings) are missing, but it matches the standard closely.
- HPF directives start on column 1 with `[Cc!\*][Hh][Pp][Ff]$`.
- FC (Fabien Coelho) directives with `[Cc!\*][Ff][Cc][Dd]$`.
- Very simple `forall` instructions without continuation are handled. They must be independent.

```
!hpf$ independent
      forall(i=1:n) a(i)=a(i)+1
```

- `!` style Fortran 90 comments are managed in directives.

```
!hpf$ dynamic A     ! here is a comment
!hpf$ template T(n) ! and another...
```

- 6th column and `&` style continuations are handled in directives.

```
!hpf$ processors
!hpf$x   P(hpf_number_of_processors())
!hpf$ distribute T(block) &
!hpf$    onto P
```

- The directive analysis does handle Fortran lexical issues in directives. However, I do not recommand to use this feature which is a bad Fortran custom.

3.2.2 Static mapping directives

The HPF mapping directives are both simplified and extended on a syntactic point of view:

- Arrays may be used as templates.

```
      real A(n,n), B(n,n), C(n)
!hpf$ align B(i,j) with A(i,j)
*hpf$ align C(i) with A(*,i)
```

- No alignment trees (A is aligned with B which is aligned with C).
- Dummy variable-based or implicit syntax for alignments (the implicit syntax is not strictly conformant but definitely convenient)

```
!hpf$ align A(i,j) with T(j,i)
!hpf$ align D with T2
```

- list of distributee and alignee are allowed:

```
!hpf$ align A(i,j), B(j,k) with T(i,k)
!hpf$ distribute T(block,*), D(*,cyclic) onto P
```

- The default distribution is `block` on each dimension of the processor arrangement.

```
!hpf$ template T(n,n,n)
!hpf$ processors P4(2,2)
!hpf$ distribute T onto P4 ! == T(block,block,*)
```

- The default processors arrangement is a vector of 8 processors named `hpfcX`.

```
!hpf$ distribute T(cyclic)
!
! same as
!
!hpf$ processors hpfcX(8)
!hpf$ distribute T(cyclic) onto hpfcX
```

- Static HPF mapping directives are allowed for variables which are local or in commons. This is not strictly speaking HPF conformant. They cannot be remapped. Commons with distributed arrays should be declared *exactly* the same in the different subroutines and functions. Overlaps are managed thanks to the interprocedural compilation.

```
      integer n
      parameter(n=1000)
      common /shared/ V(n,n), U(n,n), vmax
      real V, U, vmax
!hpf$ align with U :: V
!hpf$ distribute U(block,block) onto P
```

- All sizes of HPF objects (distributed arrays, templates, processors) must be statically known. `hpf_number_of_processors()` is given a default value.

```
      integer n,m
      parameter (n=10,m=10000)
      real A(m,n)
!hpf$ template T(m,n), T2(100,100,100)
!hpf$ template D(100,m)
!hpf$ processors P(hpf_number_of_processors())
!hpf$ processors p2(4,4), proc(n)
```

- Some simple so called free style is handled:

```
!hpf$ align with T :: A, B
!hpf$ distribute onto P :: T(block)
!
!hpf$ re align with T(i) :: C(i)
!hpf$ redistribute onto P :: T(cyclic)
!
!hpf$ distribute (block) onto P :: X, Y
!hpf$ realign (i,j) with D(j,i) :: V, W
```

3.2.3 Dynamic mapping directives

The mapping of an array may be modified dynamically within the program or at subroutine boundaries. Dynamic directives are as constrained as static directives.

- Dynamic HPF mapping directives (`realign`, `redistribute`) are only allowed on local arrays, and so that there is only one reaching distribution for each array reference in an instruction.

```
!hpf$ distribute A(block,block)
      if (a(1,1).lt.1.0) then
!hpf$   redistribute A(block,cyclic)
!hpf$   independent(i,j)
        A = A ...
      endif
!hpf$ redistribute A(block,block) ! back...
      print *, A
```

- *Prescriptive* and *descriptive* mappings at subroutine interface are allowed. Thus full arrays can be passed as subroutine arguments.

```
      subroutine predes(X,Y)
      real X(100), Y(100)
!hpf$ distribute X * (block) ! descriptive
!hpf$ distribute Y(block)    ! prescriptive
```

- Local sections of distributed arrays may be passed to a pure function within a parallel loop, if properly aligned...

```
      subroutine fft1(signal,spectrum)
!fcd$ pure ! equivalent to the f90 attribute...
      complex signal(255), spectrum(255)
      ...
      end

      subroutine process
      complex signal(255,255), spectrum(255,255)
!hpf$ align with signal:: spectrum
!hpf$ distribute signal(*,block)
!hpf$ independent
      do i=1, 255
        call fft1(signal(1,i),spectrum(1,i)) ! signal(:,i)
      enddo
```

- Directive `inherit` is handled as a default prescriptive block distribution, thus both following directives are strictly equivalent for HPFC:

```
!hpf$ inherit X
!hpf$ distribute X
```

3.2.4 HPF Parallelism directives

Parallelism can be expressed:

- Directives `independent` and `new` are taken into account. They are managed orthogonaly and homogeneously as suggested in Chapter II.2. Moreover `independent` can be specified the parallel loop indexes. Thus one can write

```
!hpf$ new(x),
```

```
!hpf$.   independent(j,i)
      do j=1, n
        do i=1, n
          x = A(i,j)*A(i,j)+B(i,j)
          C(i,j) = x*x+x
        enddo
      enddo
```

- Reduction functions (sum, prod, min, max) are recognized by name and with a special FORTRAN 77 syntax to please the parser.

```
      real A(n)
      x = redmin1(A, 5, n-3) ! x = MIN(A(5:n-3))
```

- The `reduction` directive is also implemented for +, *, min and max:

```
      s = 2.1
!hpf$ independent,
!hpf$x    reduction(s)
      do i=2, n-1
        s = s + a(i)
      enddo
      print *, 'sum is ', s
```

- There is a `pure` directive (instead of an attribute to a function or subroutine, because the compiler deal with FORTRAN 77 only). They should only deal with scalar data. Actually it declares elemental functions.

```
      program foo
      external myop
!fcd$ pure myop ! the external real myop function is pure
      real A(100)
!hpf$ distribute A
!hpf$ independent
      do i=1, n
         A(i) = myop(A(i))
      enddo
      end
      real function myop(x)
!fcd$ pure ! the current function is pure
      myop = x*x+sqrt(sqrt(x))
      end
```

3.2.5 Extensions to HPF

HPFC incorporates many extensions of HPF, the need of which was encountered in real applications.

- There is an `io` directive (functions declared io are just called by the host after a gather of the needed data).

```
      program blah
!fcd$ io output ! output subroutine is io...
      ...
      call output(A)
```

```
      end
      subroutine output(x)
!fcd$ io ! current function is io
      print *, 'here is the output', x
      end
```

- Code sections can be managed by the host only, without fine grain update of the values on the nodes, with `host` and `end host`. All user's functions called within such section should be either `pure` or `io`. Example:

```
c m will be updated on the node only once
c   a correct value is obtained from the user...
cfcd$ host
 10   print *, 'm = ?'
      read *, m
      if (m.lt.0.or.m.gt.100) then
         print *, 'invalid value, retry'
         goto 10
      endif
cfcd$ end host
```

- Simple sections of codes can be localized with `local` and `end local`. The point is to tell the compiler about a variables private to a section of code. For instance:

```
cfcd$ local, new(c1,w)
      c1 = 4/((1/v(np-1,np))+(1/v(np-1,np-1)) +
     .  (1/v(np-2,np))+(1/v(np-2,np-1)) )
      w = c1*deltat
      be(np-1,1) = 2*w/(4*w+3*h)
      be(np-1,2) = 3*h/(4*w+3*h)
      v(np-1,np)  = c1
cfcd$ end local
```

- Synchronization (`synchro`) and timing (`time/end time`) functions can be requested for performance measurements. Timing functions can be nested (up to ten). They must apply on a section of code. The message attached to the end is displayed together with the time. These directives cannot appear within a parallel loop. Such an approach with special directives should have been chosen by vendors instead of propriatary calls (for instance the DEC compiler handles `call hpf_synchro()` to synchronize HPF programs on user request, what cannot be considered as portable).

```
!fcd$ synchro
      print *, 'let us go'
!fcd$ time
      do i=1, 10
!fcd$   time
        call foo
!fcd$   end time('foo call')
      enddo
!fcd$ end time('full loop')
!fcd$ synchro
      print *, 'that is all'
```

`time on` and `time off` are obsolescent directives for `time` and `end time`.

- Arrays the values of which are dead at a point can be marked as such with the `kill` directive. For remappings on dead arrays, no communication will occur at runtime.

```
!hpf$ template T(n)
!hpf$ align with T :: A
!hpf$ processors P(4)
!hpf$ distribute T(block) onto P
!
! A is used
!
!fcd$ kill A ! I know the value will not be reused...
!hpf$ redistribute T(cyclic) onto P
!
! A is used again, but defined before being referenced.
!
```

- The `set` directive (with `int` and `bool` variants) allows to switch compiler options from the source code. They are declaration directives. The options are those reported in the properties of PIPS. To be manipulated with great care. Introduced for validation purposes.

```
!fcd$ set bool('HPFC_USE_BUFFERS',1)   ! set to TRUE
!fcd$ set int ('HPFC_BUFFER_SIZE',500000)
!fcd$ set bool('HPFC_LAZY_MESSAGES',0) ! set to FALSE
!fcd$ setbool ('HPFC_IGNORE_FCD_SYNCHRO',1) ! no SYNCHRO
```

- The `fake` directive tells the compiler that the source code provided for a given function is a fake one which mimics its effects. Thus the compiler will not attempt to generate any code for this function. It is simply understood as "do not compile me". This feature is used for interfacing Hpfc with **XPOMP**. A fake function must be either `pure` or `io`.

```
      subroutine xpomp_get_depth(d)
      integer d
!fcd$ io
!fcd$ fake
      read *, d
      end
```

- Fake `io` and `pure` function will not be compiled by the Hpfc compiler. However, these function will have to be provided for linking the final executables. The specials `ld io` and `ld pure` directives allow this information to be provided as additional shell-interpreted linker options:

```
! Tells where to find xpomp functions:
!fcd$ ld io -L$XPOMP_RUNTIME -lxpomp
```

The `io` version is linked to the *host* and initial (sequential) executables. The `pure` applies to all executables.

3.3 Compiler output

The compiler generates code for the SPMD message passing programming model. It includes many optimizations. The generated code is based on a runtime support library. The communications are handled by PVM. The whole compilation process is driven and automated by shell scripts.

3.3.1 Target programming model

Hpfc's target programming model is a host-node distributed memory message passing model. The host is in charge of I/Os and of the process management. The nodes perform the computations. Two programs are generated by the compiler. One for the host, and an SPMD code for the nodes, which is parametrized by the node id. The generated codes rely on a runtime library which provides many services. Lower level issues in the process management (spawning of tasks, etc.) and basic communications are handled by PVM.

3.3.2 Compiler optimizations

The basic compilation technique is known as the *runtime resolution of accesses to distributed arrays*. Each processor executes the whole control flow of the original program, and each reference to distributed data is resolved dynamically, and elementary messages are exchanged, following the *Owner Computes Rule*. This is of course very inefficient, and optimizations are needed if the program is expected to run quicker on a parallel machine.

The following optimizations are implemented within Hpfc:

- Distributed array declarations are reduced on the nodes.
- Reductions are compiled thru calls to dedicated runtime functions.
- Overlap analysis.

  Conditions:

  - Independent loop nest, maybe non perfectly nested.
  - Rectangular iteration space, maybe not statically known.
  - Multidimensional block distributions.
  - Arrays should be *nearly* aligned one with the other.
  - No alignement stride, no replication of written arrays.
  - flip-flops are allowed.

  Output:

  - New loop nest on local data for the nodes.
  - Communications are vectorized outside of the loop nest.
  - Array declarations are extended to store data from neighbour processors on overlap areas.
- I/O communications (collect to/update from host) are specially managed with a polyhedron-based technique (abstract [61]), see Chapter III.3.
- 1D shift on rectangular areas of block-distributed stride-1 aligned arrays are recognized and compiled thru calls to appropriate runtime support functions. It is not far from a hack, but it allows to handle wraparound computations. Example:

  ```
  !hpf$ distribute V(block,block) onto P
  ```

```
!hpf$ independent
      do i=2, n-1
         V(1,i) = V(n,i)
      enddo
```

- loop nests that fully copy an array into another, both being aligned, are especially handled. One more hack.
- Remappings (`realign` and `redistribute`) are taken into account (abstract [68]), see Chapter III.5.
- And more linear algebra techniques to come (abstract [56])

3.3.3 Runtime library

The HPFC generated code relies on a runtime library and on runtime datastrutures. These functions are written in FORTRAN 77 on top of PVM3. Many functions are generated automatically from m4 templates, depending on the manipulated data types and arities. Parts of the library are ported to use the CM5 communication library instead of PVM3. The runtime library was successfully compiled on the following PVM architectures: `SUN4 SUNMP RS6K ALPHA CM5`.

Runtime data structures:

- HPF objects description
  - Arity and object bounds
  - Alignements and Distributions
  - Distributed arrays new declarations
- PVM processes management and communication synchronization
  - PVM process identifiers for the host and the nodes
  - PVM identity of the process
  - Number of messages exchanged with each other processes
  - Number of multicasts
  - HPF processors/PVM processes mapping
- Current status (library internal data)

Runtime functions:

- Runtime data structures initialization
- Processes initializations
- Ownership computations (for the runtime resolution)
- Some coherency checkings
- High level packing functions
  - The data must be block-distributed and stride-1 aligned

  - A local rectangular area is packed
  - Any type and arity

- Reduction functions for distributed data
  - The data must be block-distributed and stride-1 aligned
  - The accessed area must be rectangular
  - Implemented: MIN, MAX, SUM, PROD
  - Any type and arity (when it makes sense)

- Subarray shift functions
  - The data must be block-distributed and stride-1 aligned
  - The shifted area must be rectangular
  - The shift must be on one dimension

3.3.4 Driver and utilities

HPFC is integrated in PIPS, a prototype parallelizer. There are many utilities used by HPFC or that can be used to manipulate HPFC.

- `hpfc`, the compiler driver: Allows to call the HPFC many scripts.
  - interactive sessions (thanks to GNU readline library)
  - compile (HPF to F77 with pips)
  - make (F77 to PVM executables)
  - run (the original sequential or new parallel codes)
  - delete (hpfc directories)
  - clean (pips directories)
  - pvm (start, halt and so on)
  - ...

- shells used by the compiler:
  - `hpfc_directives`: manage HPF directives.
  - `hpfc_add_includes`: adds file inclusions.
  - `hpfc_add_warning`: adds a warning to automatically generated files.
  - `hpfc_generate_init`: generates an init file
  - `hpfc_generate_h`: generates a header file from a FORTRAN file.
  - `hpfc_install`: installation of the needed and generated files in a directory.

- Options:
  - for inlining or not *global to local* runtime resolution computations.
  - for managing especially buffers over PVM.

  - for ignoring or not timing and synchronization directives.
  - for applying the dynamic liveness analysis of array copies.

- Implementation overview
  - `hpfc` is the compiler driver shell.
  - HPF directives are translated into fake Fortran calls to reuse the PIPS parser using a shell and sed script.
  - These calls are filtered out of the parsed code and analyzed to set internal data structures which describe array mappings and so.
  - New declarations are computed for the distributed arrays.
  - The input code is analyzed by PIPS (array regions, preconditions).
  - Then 2 codes are generated for each subroutine. One for the host and an SPMD code for the nodes. Implemented as a double rewriting of the AST.
  - Initialization codes for the runtime data structures are also generated.
  - Finally common declarations are dealt with, plus global initializations.
  - The generated codes are compiled and linked to PVM and the Hpfc Runtime library. Two executables are built (host and nodes). The host spawns the nodes.

Figure 3.1 shows an overview of Hpfc within PIPS, decomposed into several interacting phases within the compiler.

3.4 Conclusion

Hpfc is a prototype HPF compiler in which some of our techniques have been implemented. It serves as a realistic testbed for testing new optimization techniques. The compiler includes 18,000 lines of ANSI C code as a PIPS module, plus 1,300 of shell and other sed scripts for the drivers, and finally 3,500 lines of M4/Fortran 77 for the runtime support library (expanded to +10,000).

3.4.1 Related work

Prototype compilers that generate code from Fortran (more or less HPF) or C for distributed-memory machines: See the excellent survey maintained by Jean-Louis Pazat at IRISA.

- Adaptor (GMD):

  `http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html`

- Annai (CSCS-NEC):

  `http://www.cscs.ch/Official/Project_CSCS-NEC.html`

- DEC f90 (Digital):

  `http://www.digital.com/info/hpc/fortran`

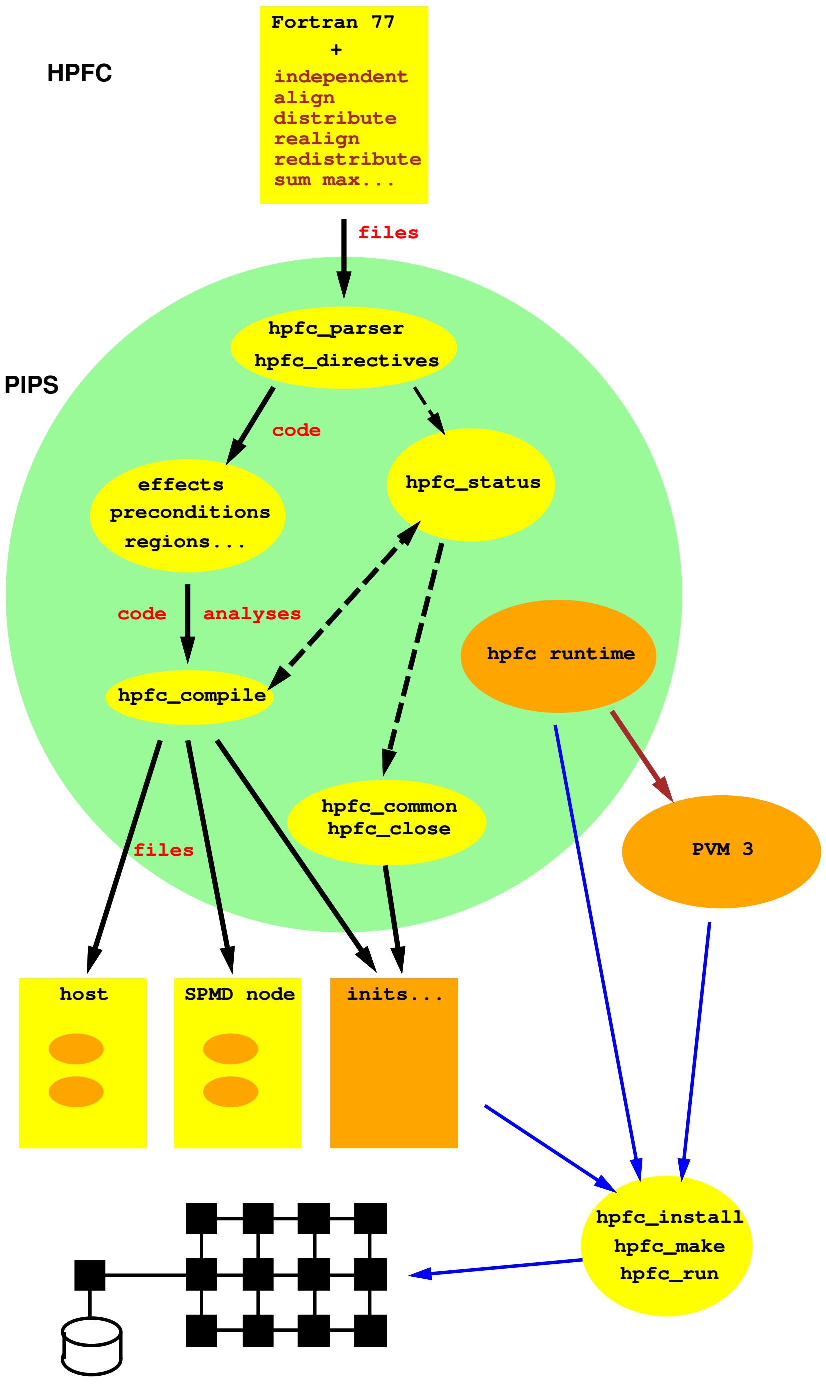


Figure 3.1: HPFC overview

- D System (Rice, Syracuse):
  `http://www.cs.rice.edu/fortran-tools/DSystem/DSystem.html`
- EPPP (CRI Montréal):
  `http://www.crim.ca/apar/Group/English/Projects/EPPP.html`
- Fx (Carnegie Mellon University):
  `http://www.cs.cmu.edu/~fx`
- HPFC (École des mines de Paris):
  `http://www.cri.ensmp.fr/pips/hpfc.html`
- HPF Mapper (NA Software):
  `http://www.connect.org.uk/merseymall/NASoftware/mapper.html`
- Pandore (IRISA):
  `http://www.irisa.fr/pampa/PANDORE/pandore.html`
- Paradigm (CRHPC, University of Illinois at Urbana-Champaign):
  `http://www.crhc.uiuc.edu/Paradigm`
- pghpf (Portland Group Inc.):
  `http://www.pgroup.com/HPF.src.html`
- PPPE (Esprit Project):
  `???`
- Prepare (Esprit Project):
  `http://www.irisa.fr/pampa/PREPARE/prepare.html`
- sHPF Translator (University of Southampton):
  `http://www.hpcc.ecs.soton.ac.uk/shpf/shpf.html`
- vastHPF (Pacific-Sierra Research):
  `http://www.psrv.com/vast/vasthpf.html`
- VFCS (University of Vienna):
  `http://www.par.univie.ac.at/project/vfcs.html`
- xHPF (Applied Parallel Research):
  `http://www.infomall.org/apri/prodinfo.html`
- xlhpf (IBM):
  `http://www.software.ibm.com/ap/fortran/xlhpf/index.html`

Chapitre 4

Experiments with HPF Compilation for a NOW

Ce chapitre a fait l'objet d'une communication à HPCN'94 [59].

Résumé

Le *High Performance Fortran* (HPF) est un Fortran à parallélisme de données pour les multiprocesseurs à mémoire répartie. HPF propose un modèle de programmation interessant, mais les compilateurs sont encore à venir. L'implémentation d'un prototype de compilateur HPF optimisant est décrite. Des expériences avec le code généré basé sur PVM 3 sur un réseau de stations de travail sont analysées et discutées. On montre que si de tels systèmes permettent d'obtenir de bonnes accélérations à moindre coût, il ne peuvent donner des performances extensibles sans un réseau de communication spécifique. D'autres implémentations de compilateurs HPF, par des groupes universitaires et commerciaux sont présentés et comparés à nos travaux.

Abstract

High Performance Fortran (hpf) is a data-parallel Fortran for Distributed Memory Multiprocessors. Hpf provides an interesting programming model but compilers are yet to come. An early implementation of a prototype hpf optimizing compiler is described. Experiments of PVM 3-based generated code on a network of workstations are analyzed and discussed. It is shown that if such systems can provide very good speedups at low cost, they cannot allow scalable performance without specific communication hardware. Other early implementations of hpf compilers from academic and commercial groups are presented and compared to our work.

4.1 Introduction

The most promising parallel machines seem to be the Distributed-Memory Multiprocessors (DMM), such as the Intel's Paragon or the Thinking Machine's CM5. They are scalable, flexible, and offer good price/performance ratio. However their efficient programming using the message-passing paradigm is a complex and error-prone task, which makes coding hard and expensive. To ease the programmer's burden, new languages have been designed to provide a uniform name space to the user. The compiler must handle the communications, and the machine can be used as a Shared-Memory Multicomputer.

High Performance Fortran (HPF) is a new standard to support data-parallel programming for DMMs. The HPF forum, composed of vendors, academics and users, specified HPF [95] in 1992. It is intended as a *de facto* standard, supported by many companies. The language is based on Fortran 90 with additions to specify the mapping of data onto the processors of a DMM (namely directives `align` and `distribute`), parallel computations with independent and data-parallel loops (`independent` directive and `forall` instruction), reductions, plus some other extensions like new intrinsics to query about the system at run-time. The specification effort has restarted in January 1994 to address problems left aside in 1992 such as parallel I/O and irregular computations.

Section 4.2 gives a brief overview of the implementation and the optimizations performed by our compiler. Section 4.3 describes results of runs of Jacobi iterations on a network of workstations, and Section 4.4 analyzes what can be expected from this kind of applications on such systems. Section 4.5 describes the current status of other HPF compilers and compares the shortcomings of the different implementations with respect to our experiments.

4.2 Compiler implementation

A prototype HPF compiler [56] has been developed within PIPS [138] (Scientific Programs Interprocedural Parallelizer) at CRI. It is a 15,000 lines project that generates SPMD (Single Program Multiple Data) distributed code. Fortran 77 and various HPF constructs such as independent loops, static mapping directives and reductions are taken as input. The full semantics of the HPF mapping is supported, e.g. replicated dimensions and arbitrary cyclic distributions are handled. From this input, a 2PMD (2 Programs, Multiple Data, see Figure 4.3) Fortran 77 message passing code is generated. The first program mainly deals with I/O to be performed by the host processor, and the other is an SPMD code for the nodes. The code uses a library of PVM 3-based [103] run-time support functions. Thus the compiler output is as portable as PVM. Two files are also generated to initialize run-time data structures describing the distributed arrays.

The compilation is divided into 3 phases:

First, the input code is parsed and normalized (e.g. temporaries are inserted to avoid indirections), the directives are analyzed and new declarations are computed for the distributed arrays. These new declarations reduce when possible the amount of allocated memory on each node.

The second phase generates the run-time resolution code [45] for the host and nodes as a double rewriting scheme on the abstract syntax tree of the program. The *owner computes rule* is used to define the processor that performs the computations. Accesses to remote data are guarded by tests and communications if necessary. This rewriting scheme has been formalized, which enabled us to prove parts of the correctness of the compilation process. For instance, the balance of communications can easily be proved by checking that each rewriting rule generates as many sends as receives with complementary guards and destination.

The third phase performs optimizations when necessary conditions are met. An overlap analysis for multi-block-distributed arrays [105] is implemented. Alignment shifts at the template level are used to evaluate the overlap width in each directions. Guards are generated for non contributing nodes. Messages are vectorized by the process, and moved outside of the loop body. Reductions are compiled through calls to dedicated run-time functions.

4.3 Experiments

Experiments have been performed on an unloaded Ethernet network, with up to 8 Sparc Stations 1. The application (see Figure 4.1 with HPF directives) computes 100 Jacobi iterations on a 2D plate. The computation models heat transfers in an homogeneous square plate when the temperature is controlled on the edges. This application is fully parallel and few communications are needed. The kernel comes from [241], with a few I/O and initializations added. The communication pattern induced by the computation kernel, as well as the different initializations of some areas within the plate, are representative of stencil computations such as wave propagation in geophysics. The arrays are aligned and block-block distributed to minimize the amount of data exchange. Figure 4.2 shows extracts from the generated code for the machine nodes: It is an SPMD code parametrized by the processor's identity. Declarations are reduced to what is necessary on one processor, and extended to store remote data in overlap areas. Run-time support functions are prefixed by `hpfc`. The `north` vector initialization introduces a guard so that only the appropriate processors execute the loop. The local loop bounds are computed by each processor before the loop. The last part of the code shows the first send within the kernel of the computation: Each selected processor first computes to which neighbor it has to send the data, then packs and sends them. The last line is the comment that introduces the corresponding receive. All codes were compiled with all compilers' optimizations set on.

The measures are given for 1, 4 and 8 processors, and for plate width of 20 to 1,000 points (up to 2,000 points for 8 processors). They are based on the best of at least 5 runs. Figure 4.4 shows the Mflop/s achieved by the sequential runs when the plate width is scaled up. From these measures, 1.45 Mflop/s is taken as the Sparc 1 performance on this application. It matches most experimental points, and cache effects explain the other results. This reference is scaled to the number of processors to provide a realistic peak performance for the network of workstations used. Other measures are displayed as the percentage of this peak performance reached by the best run. This metric (e) is called *relative realistic efficiency*. Real speedups were rejected because of the artefacts linked to cache effects: A good speedup may only mean a bad sequential performance for a given plate width. Moreover sequential experiments were stopped because of memory limitations on the tested configuration. An absolute measure (e.g. Mflop/s) would not have clearly shown the relative efficiency achieved. Note that $p.e$ (p processors, efficiency e) is a speedup, but not a measured one.

Figures 4.5 and 4.6 show the experimental data using this metric and theoretical curves derived from realistic assumptions in the next section. Large computations lead to up to 80–90% efficiency. The complementary part evaluates the loss

```
      program jacobi
      parameter (n=500)
      real tc(n,n), ts(n,n), north(n)
chpf$ template t(n,n)
chpf$ align tc(i,j), ts(i,j) with t(i,j)
chpf$ align north(i) with t(1,i)
chpf$ processors p(2,4)
chpf$ distribute t(block,block) onto p
chpf$ independent(i)
      do i=1,n
         north(i) = 100.0
      enddo
      ...
      do k=1,time
c     kernel of the computation
chpf$ independent(j,i)
      do j=2,n-1
        do i=2,n-1
          ts(i,j) = 0.25 *
     $      (tc(i-1,j) + tc(i+1,j)
     $     + tc(i,j-1) + tc(i,j+1))
         enddo
      enddo
      ...
```

Figure 4.1: Jacobi iterations kernel

```
      program node
      include 'fpvm3.h'
      include 'parameters.h'
      ...
      real*4 north(1:125),
     $    ts(1:250,1:125), tc(0:251,0:126)
      call hpfc_init_node
      ...
      if (mypos(1,1).eq.1) then
        call hpfc_loop_bounds(i_5,i_6,1,500..
        do i_4 = i_5, i_6
          north(i_4) = 100.0
        enddo
      endif
      ...
      do k = 1, time
c p(1:2,2:4) send tc(1:250,1) to (-1)
      if (mypos(2,1).ge.2) then
        call hpfc_cmpneighbour(-1)
        call hpfc_pack_real4_2(tc, ...)
        call hpfc_sndto_n
      endif
      ...
c p(1:2,1:3) receive tc(1:250,126) from (+1)
```

Figure 4.2: Node code extract

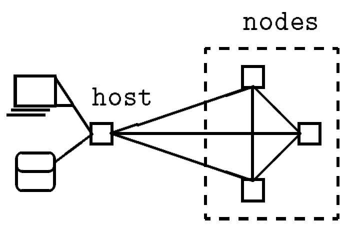


Figure 4.3: Machine model

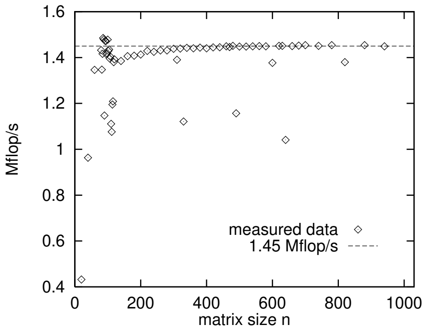


Figure 4.4: Sequential Mflop/s

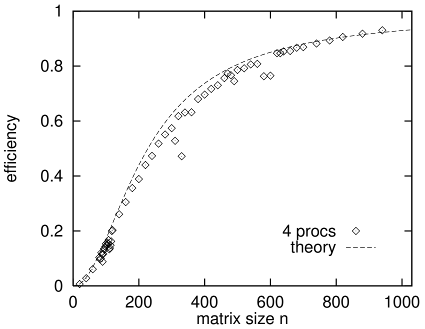


Figure 4.5: Efficiency with 4 procs

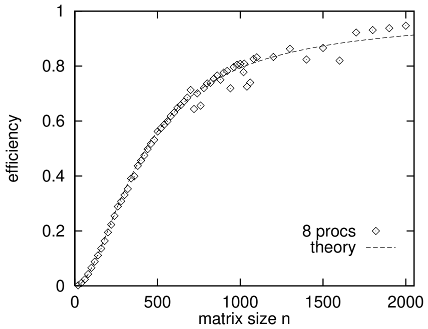


Figure 4.6: Efficiency with 8 procs

due to latencies and communications. These results are used to test the formulas derived in the next section. These formulas are then used to precisely analyze the performance which can be expected from such applications run on networks of workstations when the parameters change.

4.4 Analyses

The theoretical curves (doted lines on Figures 4.5 and 4.6) are based on assumptions about the network bandwidth (γ = 700 KB/s through PVM), the realistic peak performance of one Sparc 1 (π = 1.45 Mflop/s) and an offset ($\alpha p + \delta$) to model the initialization times. It is assumed (a) that floats are 4 bytes long, (b) that the application is block-block distributed, (c) that communications and computations do not overlap, and (d) that the communications are sequentialized on the medium. If p is the number of processors, n the plate width and t the number of iterations, then the total time complexity $\Theta_t(n, p)$ to run the program is:

$$\Theta_t(n,p) \simeq \frac{4n^2t}{\pi p} + \frac{16n(\sqrt{p}-1)t}{\gamma} + \alpha p + \delta \tag{4.1}$$

The first term is the computation time and the second the communication time. If $\mu = \frac{\pi}{\gamma}$, the continuous regime relative realistic efficiency is:

$$e_\infty(n,p) = \lim_{t\to\infty} \left(\frac{4n^2t}{\pi p}\right) \Theta_t(n,p)^{-1} = \left(1 + \mu\frac{4p(\sqrt{p}-1)}{n}\right)^{-1} \tag{4.2}$$

These functions match very well the experimental points as can be seen on the Figures. They can be used to see what can be expected from such applications on networks of workstations. The interesting point is not to scale up the problem size for better efficiency, but to analyze what happens when the number of processors is scaled with a constant problem size: The efficiency quickly decreases. The minimal size to get efficiency e with p processors can be computed (4.3). An optimal number of processors is also easily derived (4.4), for which 2/3 of the network bandwidth is used.

$$n \geq \left(\frac{4e}{1-e}\right) \mu p(\sqrt{p}-1) \tag{4.3}$$

$$p \simeq \left(\frac{n}{2\mu}\right)^{\frac{2}{3}} \tag{4.4}$$

The μ parameter is expected to belong to a large range of values as processor and network performances change. For a network of RISC 6000 on Ethernet, $\mu \simeq 100$[1], while for SS1 $\mu \simeq 2$. For a realistic application with $n = 1000$, $\mu \simeq 100$

[1]This figure is based on the peak performance of a RS 6000 running at 62.5 MHz

gives the optimal number of processors $p = 3$, a 1.5 speedup and an overloaded network! This clearly shows that a set of fast workstations linked by Ethernet is definitely not an underused massively parallel machine, even for an embarrassingly parallel program like Jacobi iterations. Scaling the problem size to show better results is a fake, because the convergence speed and other requirements of the algorithms [201] would dramatically slow down the resolution.

The first problem is the low bandwidth of the network, especially when compared to processor speed. The local networks should be upgraded as much as the workstations to get good performance. The second problem is the complexity in p of the functions since the communications are sequentialized on the medium. The network is the bottleneck of the system.

4.5 Related work

A lot of work about compilation and prototype compilers for DMMs was published during the last few years. The Vienna Fortran Compiling System [260] and Fortran D [241] are early developments of data-parallel languages on which HPF is based. The Fortran D prototype allows only one dimension to be distributed, and no I/O. Adaptor [39] is a pragmatic and efficient public domain software which implements some features of HPF and which was initially based on CM-Fortran. Many vendors have announced commercial compilers: Applied Parallel Research, Digital Equipment, Pacific-Sierra Research, the Portland group, etc. These products are not yet available, or do not yet implement all HPF features. Moreover the performances of these systems will depend on the implemented optimizations, and many points are still an open research area. Performance figures for some of these prototypes are available [132, 40], but comparisons are difficult because the applications and the machines are different.

4.6 Conclusion

Our optimizing compiler supports the full HPF mapping semantics and produces portable code. Experiments with Jacobi iterations on a network of workstations were performed with the compiler output. Very good efficiencies were measured for large problem sizes. These results have been analyzed and discussed, and it has been shown that networks of fast workstations cannot provide good performance without specific communication hardware.

Some HPF features, such as procedure interfaces and the `forall` instruction, are missing. Moreover I/O are not yet efficiently compiled. Future work will include a better handling of I/O, experiments using more realistic programs and real parallel machines (Paragon, CM5, Alpha farm), as well as implementation of new optimization techniques [5].

Conclusion

Abstract

This study is coming to an end. It has focussed on designing, implementing and testing compilation techniques for *High Performance Fortran*. By doing so, we have touched the core of the language and discussed the impact of choices in the initial design of the language on possible implementation and optimization techniques. This discussion weights our actual realization in a working prototype compiler. This conclusion outlines our contributions and suggests possible future work.

Résumé

Cette étude, qui arrive maintenant à son terme, s'est attachée à concevoir, développer et tester des techniques de compilation pour le *High Performance Fortran*. Ce faisant, nous sommes remontés vers les sources du langage pour en discuter la pertinence au regard des techniques d'optimisation que la définition du langage rend applicable ou au contraire interdit. Ce retour sur les spécifications a le poids des réalisations effectuées. Cette conclusion reprend les différentes contributions de nos travaux. Elle met en perspective leurs différentes articulations et suggère des voies futures de développement.

Techniques de compilation

La programmation des machines parallèles à mémoire répartie est complexe parce que le programmeur n'a pas de vue globale et unifiée des données. Il manipule au contraire sur chaque processeur un espace de noms différent et doit gérer de manière explicite les communications. Pour simplifier la programmation de ces machines, des langages de programmation classiques ont été repris pour donner à l'utilisateur une vue globale des données et lui cacher les communications. Le compilateur a alors la charge de traduire le programme sous la forme précédente pour exploiter le parallélisme. HPF, basé sur Fortran 90, permet de suggérer au compilateur le placement des données et les zones de calculs parallèles. Allégé de la partie analyse du parallélisme et choix du placement, dont l'automatisation est l'objet de nombreuses recherches, il n'y a plus qu'à opérer la traduction du programme. Ce n'est pas forcément aisé si le code produit doit être efficace.

Nos travaux dans le domaine de la compilation se sont attachés à développer et implanter des techniques pour faire cette traduction du langage HPF à parallélisme et placement explicites, vue globale des données et communications implicites vers un modèle où les données réparties sont vues localement et les communications explicitées entre processus s'exécutant en parallèle. Ces travaux se sont appuyés sur des analyses et optimisations classiques de flot de données d'une part, et sur une modélisation basée sur l'algèbre linéaire, à savoir la manipulation de polyèdres paramétriques en nombres entiers, d'autre part.

Des analyses et optimisations de flot de données ont été proposées dans le cadre de la compilation des replacements, *i.e.* de la modification dynamique du placement des données. Ces mouvements de données requis par l'utilisateur sont à la fois utiles et coûteux. Nous avons donc travaillé à en réduire le nombre en notant que certains sont évitables, soit parce que les données replacées ne sont pas référencées sous cette nouvelle forme, soit parce qu'une copie encore à jour des données sous la forme requise a été conservée. Découvrir ces opportunités d'optimisation se formule naturellement dans le cadre des analyses de flot de données sur le graphe de contrôle du programme. Les exploiter requiert l'assistance de l'exécutif pour garder trace de l'état courant des données et des copies vivantes éventuellement disponibles.

La compilation du langage HPF, particulièrement la génération des codes locaux de communications et de calculs, s'est appuyée sur une modélisation basée sur l'algèbre linéaire des ensembles de données concernés, à savoir des polyèdres paramétriques en nombres entiers. Les directives de placement de HPF, les accès à des tableaux et les espaces d'itérations affines se prêtent particulièrement à cette formulation. Il s'agit ensuite de générer du code pour énumérer les ensembles en question et effectuer les communications et calculs nécessaires. Nous avons ensuite appliqué cette approche aux communications induites par les entrées-sorties, en nous appuyant sur la technique de l'analyse des régions de tableaux et en profitant de l'information d'exactitude fournie. Enfin les replacements ont fait l'objet d'optimisations propres pour bénéficier de la duplication éventuelle des données sous forme d'opportunités pour diffuser un même message à plusieurs processeurs, ainsi que pour partager la charge.

Ce type d'approche convient bien aux programmes simples et réguliers, typiquement de l'algèbre linéaire ou des différences finies, pour lesquels les espaces d'itérations et accès aux tableaux respectent naturellement les restrictions souhaitées. Il convient également à d'autres types de programmes, par exemple de calculs creux, sous réserve que les dimensions creuses des tableaux, qui sont référencées indirectement, ne soient pas distribuées. Il est aussi important de noter que les ensembles décrits peuvent dépendre de valeurs paramétriques inconnues à la compilation. Cette approche décide et gère au niveau de la compilation les calculs et communications, et nécessite donc naturellement une connaissance précise de ce qui est requis. Le problème technique de la compilation de HPF est formulé mathématiquement et est résolu grâce à des outils et algorithmes mathématiques standard.

Modifications et extensions du langage

Le développement de techniques de compilation d'un nouveau langage de programmation conduit naturellement à s'interroger sur la pertinence des décisions prises au moment de la conception du langage. Certaines contraintes, ou leur absence, imposent au compilateur une technique particulière de compilation, pas nécessairement la plus efficace, et interdisent ou réduisent les possibilités d'optimisation du code généré. Un exemple typique est la possibilité d'avoir des références dont le placement dépend dynamiquement du flot de contrôle du programme : le compilateur doit offrir une technique qui gère de tels cas improbables et inutiles avant les cas courants et utiles. Au lieu de dépenser le temps des développeurs pour améliorer les performances des codes produits, celui-ci est consommé par la correction du compilateur vis à vis de la spécification du langage.

Par ailleurs, les exemples d'applications auxquels on confronte un prototype de compilateur montrent les faiblesses et les manques éventuels du langage qui ne permettent pas d'exprimer des propriétés utiles à une bonne compilation et à une exécution efficace. D'autres besoins apparaissent également lors des phases de tests où les codes doivent être instrumentés et les options de compilation testées : il est plus commode de laisser le compilateur se charger automatiquement de ces activités. Enfin, lorsqu'on présente et enseigne un langage, on est amené à remarquer les imperfections certes mineures mais qui nuisent à sa compréhension et à son apprentissage, que ce soit sur les aspects syntaxiques ou sémantiques. Les applications et les techniques de compilation influencent la définition du langage.

Nos développements nous ont donc naturellement conduits à proposer et implanter des extensions et corrections au langage HPF. Ces propositions visent à simplifier le langage d'un point de vue syntaxique, en homogénéisant par exemple la syntaxe des directives. Elles portent aussi sur la sémantique de manière à permettre certaines techniques de compilation à la fois directes et faciles à optimiser. Enfin, quelques propositions sont des extensions qui répondent à des besoins particuliers, mais toujours accompagnées de techniques précises d'implantation efficace. Un certain nombre de suggestions ont donc été présentées, améliorant la syntaxe des directives pour la rendre plus simple et homogène, la sémantique des directives pour les rendre plus faciles à implanter et optimiser, et enfin proposant des extensions pour permettre une plus grande expressivité et de meilleurs conseils.

Dans le domaine de la syntaxe, nous avons trouvé un manque d'homogénéité et d'orthogonalité entre les directives : par exemple, la directive `independent` qui désigne une boucle parallèle ne prend pas d'argument, tandis que la directive `new` qui précise les variables privées aux itérations en prend ; certaines directives comme `new` ne s'appliquent qu'à des boucles parallèles, alors qu'il est facile de trouver des exemples où une telle directive serait souhaitable pour une section de code. Pour répondre à ces problèmes, nous avons donc suggéré d'homogénéiser la syntaxe et de fournir un marqueur de porté *(scope)* pour les directives.

Pour ce qui est de la sémantique, nous avons suggéré des restrictions au langage, de manière à simplifier l'implantation. L'exemple le plus clair est la demande d'une contrainte interdisant les références avec un placement ambigu, afin de permettre la compilation des replacements au moyen de simples copies entre tableaux différemment placés. On passe ainsi d'un programme aux placements dynamiques à une forme statique, dont le compilateur sait tirer bénéfice. Une autre requête est le retrait des placements d'arguments *transcriptifs* (directive `inherit`) qui ne donnent aucune information au compilateur. Cependant, des extensions sont nécessaires pour proposer une autre solution plus commode pour la compilation qui satisfasse les besoins réels actuellement couverts par ce type de placement.

La principale extension de langage que nous avons présentée est la directive `assume`. Elle permet de déclarer plusieurs placements possibles aux arguments d'une fonction, et évite ainsi la pratique de clonage manuel nuisible à la qualité des codes sources, ou bien l'utilisation de placements transcriptifs, nuisible elle à la qualité des codes objets. Par ailleurs notre prototype de compilateur inclut des directives propres, dont la nécessité ou l'utilité ont été mises en évidence lors de la compilation de programmes réels : par exemple, `io` permet de déclarer qu'une routine est utilisée pour effectuer des entrées-sorties, `host` identifie une section de code pour laquelle on souhaite une exécution séquentielle sur le processeur hôte, `kill` précise que les valeurs des éléments d'un tableau ne seront plus utilisées, ou `time` demande l'instrumentation d'une section de code pour en mesurer les temps d'exécution.

La philosophie générale de ces suggestions est de simplifier le langage d'une part, mais aussi de fournir plus d'informations au compilateur, informations qui nous ont paru utiles pour les applications réelles et pour certaines techniques de compilation. Nous visons donc à ramener au niveau du compilateur autant de décisions que possible, dans le but de générer un code plus efficace en tirant parti statiquement des informations disponibles.

Réalisations effectives

Proposer, décrire, présenter des techniques de compilation ou des modifications et extensions d'un langage de programmation est une chose. Cependant le discours a plus de poids si ces techniques ont été effectivement éprouvées et testées au sein d'un prototype de compilateur. Nous avons donc implanté certaines des techniques et extensions suggérées dans `hpfc`, un prototype de compilateur HPF développé au sein du paralléliseur automatique PIPS. Ce prototype représente au total 25 000 lignes de code.

L'implantation de nos techniques s'est appuyée sur l'environnement de programmation proposé par PIPS, qui englobe l'outil de génie logiciel NEWGEN et la bibliothèque de manipulation de polyèdres en nombres entiers LINEAR C^3. Nous avons été amenés au cours de cette implantation à compléter ces deux librairies, en ajoutant par exemple un itérateur général à NEWGEN pour se déplacer rapidement et simplement dans des structures complexes, ou en optimisant et améliorant la fonction de génération de code de parcours de polyèdres de la bibliothèque linéaire.

Notre implantation incorpore des techniques usuelles de compilation pour machines parallèles à mémoire répartie comme la résolution dynamique des accès aux variables distribuées, ainsi que l'analyse des recouvrements pour les tableaux distribués par blocs. Elle ajoute à cette base nos techniques pour la compilation des entrées-sorties et pour les replacements. Elle inclut enfin les petites extensions utiles à certains programmes mais aussi à l'instrumentation des codes pour les tests. Les nombreuses fonctions de support nécessaires à l'exécution du programme sont implantées au sein d'une librairie. Cette librairie assure des communications de haut niveau comme les mises à jour de recouvrements ou les calculs de réductions. Les communications de bas niveau sont assurées par la bibliothèque domaine public PVM.

Une caractéristique importante de cette implantation est qu'elle réutilise les analyses de programme disponibles dans PIPS, au bénéfice de la qualité du code généré. L'analyse syntaxique de Fortran est évidemment reprise, à travers un filtre qui met sous forme analysable les directives. Les préconditions sont utilisées lors de la génération des codes pour profiter des informations disponibles sur les liens et bornes des différentes variables scalaires. Enfin l'analyse des régions de tableaux permet de réduire les communications liées aux entrées-sorties aux éléments effectivement référencés.

Cette implantation a permis de tester l'efficacité de nos techniques sur des programmes réels et des machines parallèles. Notamment, l'analyse des recouvrements pour une relaxation jacobienne a été éprouvée sur réseau de stations de travail, et des transpositions de matrices basées sur des replacements ont permis de mesurer leurs performances.

Portée des travaux

Ces travaux améliorent l'état de l'art de la compilation en amenant plus de décisions et choix au niveau du compilateur, qui tire ainsi parti au maximum des informations statiques disponibles. Nous avons donc remonté la frontière entre compilation et exécutif, avec l'idée de générer des codes plus performants car plus dédiés au problème. Par exemple, des décisions de partage de charge sont prises statiquement dans le cadre des replacements. Pour les entrées-sorties, des communications sont évitées quand les analyses annoncent une description exacte des éléments référencés.

Un second aspect non moins important est le caractère scientifique de l'approche adoptée, qui consiste à modeliser les problèmes techniques de compilation et d'optimisation dans des cadres mathématiques éprouvés, comme les manipulations de polyèdres paramétriques en nombres entiers, ou les formulations standard de calculs de flot de données. De nombreux travaux suivent le même type d'approche et de philosophie de par le monde, dans le domaine de la parallélisation automatique, des transformations de programmes, du placement automatique des données, ou de la génération de code assembleur. Ce faisant, notre implantation au sein d'un prototype de belle facture a montré leur réalisme et leur efficacité, au-delà du simple attrait intellectuel. Elle a également montré les perfectionnements nécessaires des outils génériques sur lesquels nous nous sommes appuyés, tant pour améliorer leur efficacité intrinsèque en précision et rapidité, que pour améliorer les codes générés.

Nos travaux sur la compilation font apparaître trois articulations qu'il nous parait intéressant de signaler comme telles. Elles concernent le langage compilé, les analyses de programmes et les outils conceptuels utilisés. En développant des techniques de compilation, nous avons remis en cause naturellement certains choix faits lors de la conception du langage, qui limitent les optimisations et les techniques à mettre en œuvre pour traduire un programme. Nous avons pu mesurer l'intérêt pour nos activités de certaines analyses de programme et leur impact sur les performances. En s'appuyant sur des outils génériques, nous avons pris conscience à la fois de leurs forces et de leurs faiblesses, et cherché à les améliorer.

Travaux futurs

Nos futurs travaux sont naturellement dans la continuation des travaux présentés dans cette thèse. D'abord, ils concernent la compilation : intégration de nos techniques au sein de notre prototype, extension de leurs champs d'application et optimisation des codes générés. L'impact de telles techniques de compilation sur les analyses de programme, éventuellement interprocédurales, est aussi à explorer. Nous cherchons à améliorer et à étendre le langage HPF, et plus généralement à discuter la bonne manière de programmer une machine parallèle efficacement en facilitant à la fois le travail de l'utilisateur et du compilateur.

Il est utile de poursuivre le développement de notre prototype pour inclure plus de nos techniques et montrer leur intérêt. Il manque principalement l'implantation de la compilation des nids de boucles parallèles à base de polyèdres. Une fois implantée, cette technique devra faire l'objet d'expérimentations et tests pour valider plus avant nos travaux, et discuter les performances obtenues. Il est vraisemblable que cette implantation fera apparaître de nouveaux problèmes qui appelleront de nouvelles discussions.

Il serait intéressant d'étendre le champs d'application de ces techniques. Par exemple on peut étudier un modèle de communications de type *get-put* comme celui disponible sur le T3D. Les problèmes sont alors un peu différents, il faut connaître l'adresse de la donnée dans la mémoire distante pour assurer la communication. D'autres discussions et extensions sont nécessaires aussi pour prendre en compte les nouvelles extensions de HPF, comme le placement sur des sections de processeurs. L'interaction de nos techniques avec différents modèles de parallélisme de tâches, et l'apport qu'elles peuvent représenter est aussi une voie à étudier.

En aval de ce type de génération de code, il nous paraît important de noter que les codes d'énumération des solutions d'un système qui sont produits ont une structure très particulière. Ils forment un ou plusieurs nids de boucles dont les bornes ne sont pas simples. Ces structures tireraient particulièrement bénéfice d'optimisations standard de type déplacement d'invariants, détection de sous-expressions communes et extractions de variables inductives, spécialement en prenant en compte la commutativité et de l'associativité des opérateurs impliqués. De plus, ces bornes contiennent souvent des divisions entières à reste positif dont certaines propriétés peuvent être utilisées pour simplifier et optimiser le code.

Nos techniques utilisent certaines analyses de programme comme l'analyse des régions. Ces techniques peuvent être étendues et complétées. Il nous paraît par exemple intéressant d'étendre les régions aux Z-polyèdres pour pouvoir décrire des ensembles comportant des points régulièrement espacés. De plus, le compilateur a parfois besoin d'une information supplémentaire pour exécuter une optimisation, ce qui peut se traduire soit par une extension de langage soit par une analyse de programme. Par exemple, pour les accès irréguliers et la méthode de l'inspecteur-exécuteur, il est très important de pouvoir réutiliser au maximum les motifs de communications calculés. Pour cela, le compilateur et l'exécutif ont besoin de mesurer de manière interprocédurale l'invariance éventuelle des données dont dépend ce motif.

Un point fondamental et pratique qui mérite selon nous une grande attention est l'utilisation d'entrées-sorties parallèles, et leur impact potentiel sur le langage. Ce point a déjà été légèrement abordé au cours de nos recherches, mais les problèmes ne sont pas encore résolus. Il nous semble qu'un mécanisme puissant de compilation des replacements peut servir de base à une implantation efficace et pratique de tels stockages. L'approche habituelle pour traiter les entrées-sorties s'appuie sur l'utilisation de librairies qui doivent assurer les mouvements de données a priori arbitraires, ce qui pose éventuellement problème lorsque le placement des données n'est pas très simple.

D'un point de vue plus philosophique, nous nous interrogeons sur les limites floues et subtiles qui doivent guider les décisions de conception d'un langage. Un même besoin peut s'appuyer pour sa résolution sur le langage (et donc l'utilisateur), sur des analyses de programme requises ou utiles, ou encore sur les librairies et l'exécutif. Ces choix conditionnent la souplesse et la légèreté du langage ou de ses extensions. Il faut éviter la rigidité et la lourdeur, mais pas au prix des performances.

Épilogue

Ce travail de thèse m'aura apporté beaucoup. Une telle expérience, un travail précis, approfondi et complet poursuivi au long de trois années, est enrichissant sur de nombreux aspects. On apprend l'humanité, l'imagination, la rigueur et le pragmatisme.

L'humanité d'abord, c'est la joie de travailler avec des gens qui partagent leurs connaissances, leurs expériences et leur temps sans compter. En un mot, une partie de leur vie dont on tire profit pour faire avancer l'état de l'art de la science ou de la technologie. J'espère que ces rencontres auront été, sont et seront toujours aussi enrichissantes pour ces personnes que pour moi.

L'imagination ensuite, c'est l'effort qui mêle connaissances et arts, de chercher des solutions nouvelles à des problèmes que l'on formule à peine, pour créer petit à petit des réponses qui paraîtront évidentes à tous si elles sont les bonnes. Le but n'est pas tant de trouver une réponse satisfaisante que *la* réponse qui satisfera. J'espère que certains points des travaux décrits dans cette thèse auront apporté *la* bonne réponse à de vrais problèmes.

La rigueur après, c'est le travail intellectuel, systématique et complet, qui vise à décrire, justifier, étendre ses intuitions et ses idées pour les transformer en un discours clair et juste. Clair parce que intégrées dans une formulation classique, compréhensible et qui tire profit du travail de générations de chercheurs. Juste parce que démontrées et générales. J'espère que les résultats présentés l'ont été avec une rigueur suffisante.

Le pragmatisme enfin, c'est la satisfaction des réalisations effectuées, imparfaites mais réelles, résultats de longues heures de développement, de tests, d'essais infructueux, qui font le lien entre les solutions qu'on a imaginées, les théories qu'on a pu concevoir, et la pratique. J'espère avoir montré le réalisme des solutions envisagées à travers le prototype développé au cours de ces travaux.

Bibliographie

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] Saman P. Amarasinghe and Monica S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 126–138, June 1993.

[3] G. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conf.*, pages 483–485, 1967.

[4] Corinne Ancourt. *Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales*. PhD thesis, Université Paris VI, March 1991.

[5] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static HPF Code Distribution. In *Workshop on Compilers for Parallel Computers, Delft*, December 1993. To appear in Scientific Programming. Available at `http://www.cri.ensmp.fr`.

[6] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static HPF Code Distribution. *Scientific Programming*, ??(??):??–??, ?? 1996. To appear.

[7] Corinne Ancourt and François Irigoin. Scanning Polyhedra with DO Loops. In *Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[8] Corinne Ancourt and François Irigoin. Automatic code distribution. In *Third Workshop on Compilers for Parallel Computers*, July 1992.

[9] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1993.

[10] Françoise André, Olivier Chéron, and Jean-Louis Pazat. Compiling sequential programs for distributed memory parallel computers with Pandore II. Technical report, IRISA, May 1992.

[11] Françoise André, Marc Le Fur, Yves Mahéo, and Jean-Louis Pazat. Parallelization of a Wave Propagation Application using a Data Parallel Compiler. Publication interne 868, IRISA, October 1994.

[12] Françoise André, Jean-Louis Pazat, and Henry Thomas. Pandore: A system to manage data distribution. Publication Interne 519, IRISA, February 1990.

[13] Françoise André and Thierry Priol. Programming distributed memory parallel computers whitout explicit message passing. In *Scalable High Performance Computing Conference*, pages 90–97, 1992.

[14] ANSI. *American National Standard Programming Language Fortran, ANSI x3.9-1978, ISO 1539-1980.* New-York, 1983.

[15] Béatrice Apvrille. Calcul de régions de tableaux exactes. In *Rencontres Francophones du Parallélisme*, pages 65–68, June 1994.

[16] Béatrice Apvrille-Creusillet. Régions exactes et privatisation de tableaux. Master's thesis, Université Paris VI, September 1994. .

[17] Béatrice Apvrille-Creusillet. Régions exactes et privatisation de tableaux (exact array region analysis and array privatization). Master's thesis, Université Paris VI, France, September 1994. Available via `http://www.cri.ensmp.fr/~creusil`.

[18] Béatrice Apvrille-Creusillet. Calcul de régions de tableaux exactes. *TSI, Numéro spécial RenPar'6*, mai 1995.

[19] Florin Balasa, Frank H. M. Fransen, Francky V. M. Catthoor, and Hugo J. De Man. Transformation on nested loops with modulo indexing to affine recurrences. *Parallel Processing Letters*, 4(3):271–280, March 1994.

[20] V. Balasundaram and Ken Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1989.

[21] Prithviraj Banerjee, John A. Chandy, Manish Gupta, Eugene W. Hodges IV, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, and Ernesto Su. An Overview of the PARADIGME Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10), October 1995.

[22] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. Recent Enhancements to PVM. *Int. J. of Supercomputer Applications and High Performance Computing*, 9(2):108–127, summer 1995.

[23] Andy D. Ben-Dyke. Architectural taxonomy - a brief review. News, `comp.parallel`, June 1993.

[24] Siegfried Benkner. Handling Block-Cyclic Distributed Arrays in Vienna Fortran. Technical Report 9, Institute for Software Technology and Parallel Systems, University of Vienna, November 1994.

[25] Siegfried Benkner, Peter Brezany, and Hans Zima. Processing Array Statements and Procedure Interfaces in the Prepare High Performance Fortran Compiler. In *5th International Conference on Compiler Construction*, April 1994. Springer-Verlag LNCS vol. 786, pages 324–338.

[26] Siegfried Benkner, Barbara M. Chapman, and Hans P. Zima. Vienna Fortran 90. In *Scalable High Performance Computing Conference*, pages 51–59, 1992.

[27] Robert Bernecky. Fortran 90 Arrays. *ACM SIGPLAN Notices*, 26(2):83–97, February 1991.

[28] John A. Bircsak, M. Regina Bolduc, Jill Ann Diewald, Israel Gale, Jonathan Harris, Neil W. Johnson, Shin Lee, C. Alexander Nelson, and Carl D. Offner. Compiling High Performance Fortran for Distributed-Memory Systems. *Digital Technical Journal*, 7(3), 1995.

[29] François Bodin, Lionel Kervella, and Thierry Priol. Fortran-S: A Fortran Interface for Shared Virtual Memory Architectures. In *Supercomputing*, November 1993.

[30] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and Evaluation of Primitives for Parallel I/O. In *Supercomputing*, pages 452–461, November 1993.

[31] Rajesh R. Bordawekar, Alok N. Choudhary, and Juan Miguel del Rosario. An Experimental Performance Evaluation of Touchstone Delta Concurrent File System. In *ACM International Conference on Supercomputing*, pages 367–376, July 1993.

[32] Vincent Bouchitté, Pierre Boulet, Alain Darte, and Yves Robert. Evaluating array expressions on massively parallel machines with communication/computation overlap. *Int. J. of Supercomputer Applications and High Performance Computing*, 9(3):205–219, Fall 1995.

[33] Luc Bougé. *ParaDigme* – Le parallélisme de données comme modèle fondamental pour le calcul massivement parallèle. Rapport final, ENS-Lyon, December 1994. Thème 10 : Langages, modèles d'exécution et architectres pour le parallélisme massif, appel d'offres du Ministère de la Recherche et de l'Espace.

[34] Luc Bougé. The Data-Parallel Programming Model: a Semantic Perspective. In *PRS Spring School, Les Menuires*, March 1996. To be published in the LNCS series.

[35] Thomas Brandes. Adaptor language reference manual. Internal Report Adaptor 3, German National Research Institute for Computer Science, October 1992. ftp `ftp.gmd.de`, `gmd/adaptor`.

[36] Thomas Brandes. Adaptor the distributed array library. Internal Report Adaptor 4, German National Research Institute for Computer Science, October 1992. ftp `ftp.gmd.de`, `gmd/adaptor`.

[37] Thomas Brandes. Adaptor users guide. Internal Report Adaptor 2, German National Research Institute for Computer Science, October 1992. ftp `ftp.gmd.de`, `gmd/adaptor`.

[38] Thomas Brandes. Efficient Data-Parallel Programming without Explicit Message Passing for Distributed Memory Multiprocessors. Internal Report AHR-92 4, German National Research Institute for Computer Science, September 1992.

[39] Thomas Brandes. Adaptor: A compilation system for data parallel fortran programs. Technical report, German National Research Institute for Computer Science, August 1993.

[40] Thomas Brandes. Results of Adaptor with the Purdue Set. Internal Report AHR-93 3, German National Research Institute for Computer Science, August 1993.

[41] Thomas Brandes. Evaluation of High Performance Fortran on some Real Applications. In *High-Performance Computing and Networking, Springer-Verlag LNCS 797*, pages 417–422, April 1994.

[42] Thomas Brandes. ADAPTOR Programmer's Guide (Version 4.0). Technical documentation, German National Research Institute for Computer Science, April 1996. Available via anonymous ftp from `ftp.gmd.de` as `gmd/adaptor/docs/pguide.ps`.

[43] P. Brezany, O. Chéron, K. Sanjari, and E. van Kronijnenburg. Processing irregular codes containing arrays with multidimensional distributions by the PREPARE HPF compiler. In *The International Conference and Exhibition on High-Performance Computing and Networking*, pages 526–531, Milan, Italie, May 1995. LNCS 919, Springer Verlag.

[44] D. Callahan and Ken Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Jounal of Parallel and Distributed Computing*, 5:517–550, 1988.

[45] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[46] David Cann. Retire Fortran? a debate relinked. *Communications of the ACM*, 35(8):81–89, August 1992.

[47] CFPJ. *Abrégé du Code typographique à l'usage de la presse.* Les guides. Centre de formation et de perfectionnement des journalistes, 1991.

[48] Zbigniew Chamski. Fast and efficient generation of loop bounds. Research Report 2095, INRIA, October 1993.

[49] Zbigniew Chamski. Nested loop sequences: Towards efficient loop structures in automatic parallelization. Research Report 2094, INRIA, October 1993. In Proceedings of the 27th Annual Hawaii Int. Conf. on System Sciences, 1994, p. 14–22.

[50] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[51] Siddharta Chatterjee, John R. Gilbert, Robert Schreiber, and Thomas J. Sheffler. Modeling data parallel programs with the alignment-distribution graph. *Journal of Programming Languages*, 2(3):227–258, September 1994.

[52] Siddhartha Chatterjee, John R. Gilbert, Fred J. E. Long, Robert Schreiber, and Shang-Hua Teng. Generating local addresses and communication sets for data-parallel programs. *Jounal of Parallel and Distributed Computing*, 26(1):72–84, April 1995.

[53] Siddhartha Chatterjee, John R. Gilbert, Fred J. E. Long, Robert Schreiber, and Shang-Hua Teng. Generating local addresses and communication sets for data-parallel programs. In *Symposium on Principles and Practice of Parallel Programming*, 1993.

[54] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Thomas J. Sheffler. Array distribution in data-parallel programs. In *Language and Compilers for Parallel Computing*, pages 6.1–6.17, August 1994.

[55] Marina Chen and James Cowie. Prototyping Fortran-90 compilers for Massively Parallel Machines. *ACM SIGPLAN Notices*, pages 94–105, June 1992.

[56] Fabien Coelho. Étude de la Compilation du High Performance Fortran. Master's thesis, Université Paris VI, September 1993. Rapport de DEA Systèmes Informatiques. TR EMP E/178/CRI.

[57] Fabien Coelho. Présentation du high performance fortran. In *Journée Paradigme sur HPF*. École des mines de Paris, April 1993.

[58] Fabien Coelho. Compilation du *high performance fortran*. In *Journées du Site Expérimental en Hyperparallélisme*, pages 356–370, January 1994.

[59] Fabien Coelho. Experiments with HPF Compilation for a Network of Workstations. In *High-Performance Computing and Networking, Springer-Verlag LNCS 797*, pages 423–428, April 1994.

[60] Fabien Coelho. HPF et l'état de l'art. TR A 260, CRI, École des mines de Paris, July 1994.

[61] Fabien Coelho. Compilation of I/O Communications for HPF. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 102–109, February 1995.

[62] Fabien Coelho. Discussion about HPF kernel/HPF2 and so. `hpff-distribute` list, December 13, 1995.

[63] Fabien Coelho. Présentation du *high performance fortran*. Tutoriel HPF – Transparents, Renpar'7, Mons, Belgique, May 1995. Aussi journée PARADIGME (Apr 93), séminaire de l'IDRIS (Jun 94), journée du CEA (Feb 96). `http://www.cri.ensmp.fr/~coelho`.

[64] Fabien Coelho. Compiling dynamic mappings with array copies. TR A 292, CRI, École des mines de Paris, May 1996.

[65] Fabien Coelho. Discussing HPF Design Issues. In *Euro-Par'96, Lyon, France*, pages 571–578, August 1996. Also report EMP CRI A-284, Feb. 1996.

[66] Fabien Coelho. HPF reduce directive. `hpff-task` list, January 2, 1996.

[67] Fabien Coelho. Init-time Shadow Width Computation through Compile-time Conventions. TR A 285, CRI, École des mines de Paris, March 1996.

[68] Fabien Coelho and Corinne Ancourt. Optimal Compilation of HPF Remappings. CRI TR A 277, CRI, École des mines de Paris, October 1995. To appear in JPDC, 1996.

[69] Fabien Coelho, Cécile Germain, and Jean-Louis Pazat. State of the Art in Compiling HPF. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, pages 104–133. Springer Verlag, first edition, September 1996. Proceedings of the PRS Spring School, Les Ménuires, March 1996. Also TR EMP CRI A-286.

[70] Fabien Coelho and Henry Zongaro. `ASSUME` directive proposal. TR A 287, CRI, École des mines de Paris, April 1996.

[71] Jean-François Collard, Paul Feautrier, and Tanguy Risset. Construction of DO loops from Systems of Affine Constraints. LIP RR93 15, ENS-Lyon, May 1993.

[72] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra Johnson Baylor. Parallel Access to Files in the Vesta File System. In *Supercomputing*, pages 472–481, November 1993.

[73] Béatrice Creusillet. Analyse de flot de données : Régions de tableaux IN et OUT. In *Rencontres Francophones du Parallélisme*, mai-juin 1995.

[74] Béatrice Creusillet. Array regions for interprocedural parallelization and array privatization. Report CRI A 279, CRI, École des mines de Paris, November 1995. .

[75] Béatrice Creusillet. IN and OUT array region analyses. In *Workshop on Compilers for Parallel Computers*, June 1995.

[76] Béatrice Creusillet and François Irigoin. Interprocedural Array Region Analyses. In *Language and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 46–60. Springer Verlag, August 1995.

[77] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. *Int. J. of Parallel Programming special issue on LCPC*, 24(6):?, 1996. Extended version of [76].

[78] P. Crooks and R. H. Perrott. Language constructs for data partitioning and distribution. Submitted to Scientific Programming, 1993.

[79] Ron Cytron. Doacross: Beyond vectorization for multiprocessors (extended abstract). In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844. IEEE, August 1986.

[80] Leonardo Dagum, Larry Meadows, and Douglas Miles. Data Parallel Direct Simulation Monte Carlo in High Performance Fortran. *Scientific Programming*, xx(xx):xx, June 1995. To appear.

[81] Alain Darte and Yves Robert. Mapping uniform loop nests onto distributed memory architectures. RR 93 3, ENS-Lyon, January 1993.

[82] Jack Dongarra and David Walker. The design of linear libraries for high performance computers. LAPACK Working Note 58, Department of Computer Science, University of Tenneessee, 1993.

[83] Jack J. Dongarra. *MPI: a Message-Passing Interface Standard.* MIT Press, Cambridge, MA, December 1994.

[84] P. Dreyfus. Programming design features of the gamma 60 computer. In *Proceedings of the 1958 Eastern Joint Computer Conference*, pages 174–181, 1958.

[85] Richard E. Ewing, Robert C. Sharpley, Derek Mitchum, Patrick O'Leary, and James S. Sochacki. Distributed computation of wave propagation model using pvm. *IEEE Parallel and Distributed Technology Systems and Applications*, 2(1):26–31, 1994.

[86] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[87] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part i, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[88] Paul Feautrier. Somme efficient solution to the affine scheduling problem, part ii, multidimensional time. *Int. J. of Parallel Programming*, 21(6), December 1992.

[89] Paul Feautrier. Toward automatic partitioning of arrays on distributed memory computers. In *ACM International Conference on Supercomputing*, pages 175–184, July 1993.

[90] Jeanne Ferrante, G. R. Gao, Sam Midkiff, and Edith Schonberg. A critical survey of automatic data partinioning. Research report, IBM, 1992.

[91] Samuel A. Fineberg. Implementing the NHT-1 Application I/O Benchmark. *ACM SIGARCH Computer Architecture Newsletter*, 21(5):23–30, December 1993.

[92] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12), December 1966.

[93] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, c-21(9):948–960, September 1972.

[94] High Performance Fortran Forum. *High Performance Fortran Journal of Development*. Rice University, Houston, Texas, May 1993.

[95] High Performance Fortran Forum. *High Performance Fortran Language Specification*. Rice University, Houston, Texas, May 1993. *Version 1.0*.

[96] High Performance Fortran Forum. *High Performance Fortran Language Specification*. Rice University, Houston, Texas, November 1994. *Version 1.1*.

[97] High Performance Fortran Forum. *HPF-2 Scope of Activities and Motivation Examples*. Rice University, Houston, Texas, November 1994.

[98] Ian Foster. Task Parallelism and High Performance Languages. In *PRS Spring School, Les Menuires*, March 1996. To be published in the LNCS series.

[99] Geoffrey C. Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau-Wen Tseng, and M. Wu. The Fortran D Language Specification. TR 90079, Center for Research on Parallel Computation, Rice University, December 1990.

[100] N. Galbreath, W. Gropp, and D. Levine. Application-Driven Parallel I/O. In *Supercomputing*, pages 462–471, November 1993.

[101] Franco Gasperoni and Uwe Scheiegelshohn. Scheduling loop on parallel processors: A simple algorithm with close to optimum performance. Lecture Note, INRIA, 1992.

[102] Bernard Gaulle. *Notice d'utilisation du style french multilingue*. GUTenberg, v3,35 edition, December 1994.

[103] Al Geist, Adam Beguelin, Jack Dongarra, Jiang Weicheng, Robert Manchek, and Vaidy Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee, May 1993.

[104] Cécile Germain and Franck Delaplace. Automatic Vectorization of Communications for Data-Parallel Programs. In *Euro-Par'95, Stockholm, Sweden*, pages 429–440, August 1995. Springer Verlag, LNCS 966.

[105] Hans Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Vienna, 1989.

[106] Hans Michael Gerndt and Hans Peter Zima. Optimizing Communication in Superb. In *CONPAR90*, pages 300–311, 1990.

[107] Michael Gerndt. Work distribution in parallel programs for distributed memory multiprocessors. In *ACM International Conference on Supercomputing*, pages 96–103, June 1991.

[108] Bernard Goossens and Mohamed Akil. The challenges of the 90'. LITP 93 19, Laboratoire Informatique Théorique et Programmation, Institut Blaise Pascal, May 1993.

[109] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley Publishing Company, 1994.

[110] Philippe Granger. *Analyses Sémantiques de Congruence*. PhD thesis, École Polytechnique, July 1991.

[111] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 61–72, June 1994.

[112] M. Gupta and P. Banerjee. A methodology for high-level synthesis of communication on multicomputers. In *ACM Int. Conf. on Supercomputing*, pages 1–11, 92.

[113] M. Gupta, E. Schonberg, and H. Srinivasan. A unified data-flow framework for optimizing communication, LNCS 892. In *7th Workshop Languages and compilers for parallel computing*, pages 266–282. Springer-Verlag, 94.

[114] Manish Gupta and Prithviraj Banerjee. Paradigme: A compiler for automatic data distribution on multicomputers. In *ACM International Conference on Supercomputing*, pages 87–96, July 1993.

[115] Manish Gupta, Sam Midkiff, Edith Schonberg, Ven Seshadri, David Shields, Ko-Yang Wang, Wai-Mee Ching, and Ton Ngo. An HPF Compiler for the IBM SP2. In *Supercomputing*, December 1995.

[116] Manish Gupta, Sam Midkiff, Edith Schonberg, Ven Seshadri, David Shields, Ko-Yang Wang, Wai-Mee Ching, and Ton Ngo. An HPF Compiler for the IBM SP2. In *Workshop on Compilers for Parallel Computers, Malaga*, pages 22–39, June 1995.

[117] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. In *International Conference on Parallel Processing*, pages II–301–II–305, August 1993.

[118] S.K.S. Gupta, C.-H. Huang, and P. Sadayappan. Implementing Fast Fourier Transforms on Distributed-Memory Multiprocessors using Data Redistributions. *Parallel Processing Letters*, 4(4):477–488, December 1994.

[119] S.K.S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. Technical Report 19, Department of Computer and Information Science, The Ohio State University, 1994.

[120] Mary Hall, Brian Murphy, Saman Amarasinghe, Shih-Wei Liao, and Monica Lam. Interprocedural analysis for parallelization. In *Language and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 61–80. Springer Verlag, August 1995.

[121] Mary W. Hall, Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines. In *Supercomputing*, pages 522–534, 1992.

[122] Mary W. Hall, John M. Mellor-Crummey, Alan Carle, and René G. Rodríguez. FIAT : A framework for interprocedural analysis and transformation. In *Sixth International Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[123] Wolfgang Händer. Innovative computer architecture - how to increase parallelism but not complexity. In David J. Evans, editor, *Parallel Processing Systems - An Advanced Course*, pages 1–42. Cambridge University Press, 1982.

[124] Fawsi Hassaine. *Placement des données et Génération de code Automatique pour les Multiprocesseurs à Mémoire Distribuée.* PhD thesis, Université Paris VI, June 1993.

[125] Philip J. Hatcher, Anthony J. Lapadula, Michael J. Quinn, and Ray J. Anderson. Compiling data-parallel programs for MIMD architectures. In *European Workshops on Parallel Computing*, 1992.

[126] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers.* Scientific and Engineering Computation. MIT Press, Cambridge, MA, 1991.

[127] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[128] Ernst A. Heinz. Modula-3*: An efficienly compilable extension of Modula-3 for problem-oriented explicitly parallel programming. In *Joint Symposium on Parallel Processing*, pages 269–276, May 1993.

[129] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[130] Seema Hiranandani, Ken Kennedy, John Mellor-Crummey, and Ajay Sethi. Compilation techniques for block-cyclic distributions. In *ACM International Conference on Supercomputing*, July 1994.

[131] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed-Memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[132] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluation of compiler optimizations for Fortran D on MIMD Distributed-Memory machines. In *ACM International Conference on Supercomputing*, 1992.

[133] Seema Hirannandani, Ken Kennedy, John Mellor-Crummey, and Ajay Sethi. Advanced Compilation Techniques for Fortran D. CRPC-TR 93338, Center for Research on Parallel Computation, Rice University, October 1993.

[134] Mitsuru Ikei and Michael Wolfe. Automatic array alignment for distributed memory multicomputers. Technical report, Oregon Graduate Institute of Science and Technology, 1992. Référence incomplète.

[135] Imprimerie nationale. *Lexique des règles typographiques en usage à l'Imprimerie nationale*. Imprimerie nationale, third edition, October 1990.

[136] François Irigoin. Code generation for the hyperplane method and for loop interchange. ENSMP-CAI-88 E102/CAI/I, CRI, École des mines de Paris, October 1988.

[137] François Irigoin. Interprocedural analyses for programming environment. In Jack J. Dongara and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 333–350, Saint-Hilaire-du-Touvet, September 1992. North-Holland, Amsterdam, NSF-CNRS.

[138] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *ACM International Conference on Supercomputing*, pages 144–151, June 1991.

[139] François Irigoin, Pierre Jouvelot, Rémi Triolet, Arnauld Leservot, Alexis Platonoff, Ronan Keryell, Corinne Ancourt, Fabien Coelho, and Béatrice Creusillet. *PIPS: Development Environnement.* CRI, École des mines de Paris, 1996.

[140] ISO/IEC. *International Standard ISO/IEC 1539:1991 (E)*, second 1991-07-01 edition, 1991.

[141] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, Inc., 1962.

[142] Eric E. Johnson. Completing an MIMD multiprocessor taxonomy. *ACM SIGARCH Computer Architecture Newsletter*, 16(3):44–47, June 1982.

[143] Pierre Jouvelot and Rémi Triolet. Newgen: A language-independent program generator. EMP-CRI 191, CRI, École des mines de Paris, December 1990.

[144] Edgar T. Kalns and Lionel M. Ni. Processor mapping techniques toward efficient data redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1234–1247, December 1995.

[145] Tsunehiko Kamachi, Kazuhiro Kusano, Kenji Suehiro, Yoshiki Seo, Masanori Tamura, and Shoichi Sakon. Generating Realignment-based Communication for HPF Programs. In *International Parallel Processing Symposium*, pages 364–371, April 1996.

[146] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967.

[147] S. D. Kaushik, C.-H. Huang, and P. Sadayappan. Compiling Array Statements for Efficient Execution on Distributed-Memory Machines: Two-level Mappings. In *Language and Compilers for Parallel Computing*, pages 14.1–14.15, August 1995.

[148] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. UMIACS-TR-93 134, Institute for Advanced Computer Studies, University of Maryland, April 1993.

[149] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, February 1995.

[150] Ken Kennedy. A survey of data flow analysis techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54. Prentice-Hall, Inc., Engelwood Cliffs, 1979.

[151] Ken Kennedy and Ulrich Kremer. Automatic Data Layout for High Performance Fortran. CRPC-TR94 498-S, Center for Research on Parallel Computation, Rice University, December 1994.

[152] Ken Kennedy, Nenad Nedeljković, and Ajay Sethi. A Linear Time Algorithm for Computing the Memory Access Sequence in Data-Parallel Programs. In *Symposium on Principles and Practice of Parallel Programming*, 1995. Sigplan Notices, 30(8):102–111.

[153] Ken Kennedy, Nenad Nedeljković, and Ajay Sethi. Efficient address generation for block-cyclic distributions. In *ACM International Conference on Supercomputing*, pages 180–184, June 1995.

[154] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., 1978.

[155] Ronan Keryell. *WPips and EPips User Manual (Paralléliseur Interprocédural de Programmes Scientifiques) — Linear Algebra based Automatic Parallelizer and Program Transformer*. CRI, École des mines de Paris, April 1996. Also report A-288.

[156] Ronan Keryell, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, and Pierre Jouvelot. PIPS: A Framework for Building Interprocedural Compilers, Parallelizers and Optimizers. Technical Report 289, CRI, École des mines de Paris, April 1996.

[157] Gary A. Kildall. A unified approach to global program optimization. In *Symposium on Principles of Programming Language*, pages 194–206, 1973.

[158] Alexander C. Klaiber and James L. Frankel. Comparing data-parallel and message-passing paradigms. In *International Conference on Parallel Processing*, pages II–11–II–20, August 1993.

[159] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on simd machines. *Jounal of Parallel and Distributed Computing*, 8, 1990.

[160] Kathleen Knobe and Venkataraman Natarajan. Data optimisation: Minimizing residual interprocessor data motion on SIMD machines. In *Third Symposium, on the Frontiers of Massively Parallel Computation*, pages 416–423, October 1990.

[161] Charles Koelbel. Compile-time generation of communication for scientific programs. In *Supercomputing*, pages 101–110, November 1991.

[162] Charles Koelbel, David Loveman, Robert Schreiber, Guy Steele, and Mary Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.

[163] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[164] Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. Supporting shared data structures on distributed memory architectures. Technical Report ASD 915, Purdue University, January 1990.

[165] Charles Koelbel and Robert Schreiber. ON clause proposal. `hpff-task` list, January 8, 1996.

[166] Ulrich Kremer. *Automatic Data Layout for Distributed Memory Machines*. PhD thesis, Rice University, Houston, Texas, October 1995. Available as CRPC-TR95-559-S.

[167] Leslie Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17:83–93, February 1974.

[168] Leslie Lamport. *LaTeX, A Document Preparation System*. Addison-Wesley Publishing Company, 5 edition, 1986.

[169] Marc Le Fur. Scanning Parametrized Polyhedron using Fourier-Motzkin Elimination. Publication interne 858, IRISA, September 1994.

[170] Marc Le Fur. Scanning Parameterized Polyhedron using Fourier-Motzkin Elimination. In *High Performance Computing Symposium*, pages 130–143, July 1995.

[171] Marc Le Fur, Jean-Louis Pazat, and Françoise André. Commutative loop nests distribution. In *Workshop on Compilers for Parallel Computers, Delft*, pages 345–350, December 1993.

[172] Marc Le Fur, Jean-Louis Pazat, and Françoise André. An Array Partitioning Analysis for Parallel Loop Distribution. In *Euro-Par'95, Stockholm, Sweden*, pages 351–364, August 1995. Springer Verlag, LNCS 966.

[173] Hervé Le Verge, Vincent Van Dongen, and Doran K. Wilde. Loop nest synthesis using the polyhedral library. Publication Interne 830, IRISA, May 1994.

[174] Oded Lempel, Shlomit S. Pinter, and Eli Turiel. Parallelizing a C dialect for Distributed Memory MIMD machines. In *Language and Compilers for Parallel Computing*, August 1992.

[175] Hugues Leroy. Les machines parallèles à mémoire distribuée. copies de transparents, journées GUS, June 1993.

[176] Arnauld Leservot. *Analyse interprocédurale du flot des données*. PhD thesis, Université Paris VI, March 1996.

[177] John M. Levesque. *FORGE 90 and High Performance Fortran*. Applied Parallel Research, Inc., 1992. xHPF77 presentation.

[178] John M. Levesque. Applied Parallel Research's xHPF system. *IEEE Parallel and Distributed Technologies*, page 71, Fall 1994.

[179] J. Li and Marina Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[180] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, September 1986.

[181] Fujitsu Limited. Langage processors system on vpp500 system. Copies de transparents, confidentiel, 1993.

[182] Z. Lin and S. Zhou. Parallelizing I/O Intensive Applications for a Workstation Cluster: a Case Study. *ACM SIGARCH Computer Architecture Newsletter*, 21(5):15–22, December 1993.

[183] Tom MacDonald, Doug Pase, and Andy Meltzer. Addressing in Cray Research's MPP fortran. In *Third Workshop on Compilers for Parallel Computers*, pages 161–172, July 1992.

[184] Yves Mahéo and Jean-Louis Pazat. Distributed Array Management for HPF Compilers. Publication Interne 787, IRISA, December 1993.

[185] Yves Mahéo and Jean-Louis Pazat. Distributed Array Management for HPF Compilers. In *High Performance Computing Symposium*, pages 119–129, July 1995.

[186] Philippe Marquet. *Langages explicites à parallélisme de données*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille — Université des Sciences et Technologies de Lille, February 1992.

[187] François Masdupuy. *Array Indices Relational Semantic Analysis using Rational Cosets and Trapezoids*. PhD thesis, École Polytechnique, November 1993. Technical Report EMP-CRI A/247.

[188] Larry F. Meadows, Douglas Miles, Clifford Walinsky, Mark Young, and Roy Touzeau. The Intel Paragon HPF Compiler. Technical report, Portland Group Inc., June 1995.

[189] Peter G. Meisl, Mabo R. Ito, and Ian G. Cumming. Parallel synthetic aperture radar processing on workstation networks. In *International Parallel Processing Symposium*, pages 716–723, April 1996.

[190] John Merlin. Techniques for the automatic parallelisation of 'Distributed Fortran 90'. SNARC 92 02, University of Southampton, 1992.

[191] Ravi Mirchandaney, Joel S. Saltz, Roger M. Smith, David M. Nicol, and Kay Crowley. Principles of Runtime Support for Parallel Processors. In *ACM International Conference on Supercomputing*, pages 140–152, July 1988.

[192] A. Gaber Mohamed, Geoffrey C. Fox, Gregor von Laszewski, Manish Parashar, Thomas Haupt, Kim Mills, Ying-Hua Lu, Neng-Tan Lin, and Nang-Kang Yeh. Application benchmark set for Fortran-D and high performance fortran. Technical Report 327, Syracuse University, Syracuse, New-York, 1992.

[193] Thierry Montaut. *Méthode pour l'élimination du faux-partage et l'optimisation de la localité sur une machine parallèle à mémoire virtuelle partagée.* PhD thesis, Université de Rennes I, May 1996. To be defended.

[194] Steven A. Moyer and V. S. Sunderam. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In *Scalable High Performance Computing Conference*, pages 71–78, 1994.

[195] Carl D. Offner. Digital's HPF Compiler: Meeting the Challenge of Generating Efficient Code on a Workstation Farm. In *NASA Ames Workshop*, 1993.

[196] Carl D. Offner. *Shadow Width Declarations.* Digital Equipment Corp., January 1996. Proposal for HPF.

[197] Joel Oren and Gowri Ramanathan. Survey of commercial parallel machines. News, `comp.parallel`, March 1993.

[198] Maria Ines Ortega. *La mémoire virtuelle partagée répartie au sein du système Chorus.* PhD thesis, Université Paris VII, December 1992.

[199] Edwin Paalvast. *Programming for Parallelism and Compiling for Efficiency.* PhD thesis, Delft University of Technology, June 1992.

[200] Edwin M. Paalvast, Henk J. Sips, and A.J. van Gemund. Automatic parallel program generation and optimization from data decompositions. In *1991 International Conference on Parallel Processing — Volume II : Software*, pages 124–131, June 1991.

[201] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *Computer*, pages 42–50, July 1993.

[202] Nicolas Paris. Definition of POMPC (Version 1.99). Technical Report LIENS-92-5-bis, Laboratoire d'Informatique, École Normale Supérieure, March 1992.

[203] Douglas M. Pase. MPP Fortran Programming Model. Draft version 1.2, January 1992.

[204] Barbara K. Pasquale and George C. Polyzos. A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload. In *Supercomputing*, pages 388–397, November 1993.

[205] Michael Philippsen, Ernst A. Heinz, and Paul Lukowicz. Compiling machine-independent parallel programs. *ACM SIGPLAN Notices*, 28(8):99–108, August 1993.

[206] Alexis Platonoff. Quel fortran pour la programmation massivement parallèle ? Note CEA 2713, Commissariat à l'Énergie Atomique, November 1992.

[207] Alexis Platonoff. Automatic data distribution for massively parallel computers. In *Workshop on Compilers for Parallel Computers, Malaga*, pages 555–570, June 1995.

[208] Alexis Platonoff. *Contribution à la Distribution Automatique de Données pour Machines Massivement Parallèles.* PhD thesis, Université Paris VI, March 1995.

[209] Loic Prylli and Bernard Tourancheau. Efficient block cyclic data redistribution. Research Report 2766, INRIA, January 1996. To appear in Europar'96.

[210] William Pugh. A pratical algorithm for exact array dependence analysis. *CACM*, 35(8):102–114, August 1992.

[211] Frédéric Raimbault. *Étude et Réalisation d'un Environnement de Simulation Parallèle pour les Algorithmes Systoliques.* PhD thesis, Université de Rennes 1 – Institut de Formation Supérieur en Informatique et en Communication, January 1994.

[212] Shankar Ramaswamy and Prithviraj Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 342–349, February 1995.

[213] S. Ranka and J.-C. Wang. Static and runtime scheduling of unstructured communication. *Int. J. of Computing Systems in Enginieering*, 1993.

[214] S. Ranka, J.-C. Wang, and M. Kumar. Personalized communication avoiding node contention on distributed memory systems. In *International Conference on Parallel Processing*, pages I–241–I–243, August 1993.

[215] S. Ranka, J.-C. Wang, and M. Kumar. Irregular personalized communication on distributed memory systems. *Jounal of Parallel and Distributed Computing*, 1995. To appear.

[216] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[217] S. S. Reddi and E. A. Feustel. A conceptual framework for computer architecture. *ACM Computing Surveys*, 8(2), June 1976.

[218] Xavier Redon. *Détection et exploitation des récurrences dans les programmes scientifiques en vue de peur parallélisation.* PhD thesis, Université Paris VI, January 1995.

[219] Xavier Redon and Paul Feautrier. Detection of reductions in sequential programs with loops. In *International Parallel Architectures and Languages Europe*, pages 132–145, June 1993. LNCS 694.

[220] Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1989.

[221] Anne Rogers and Keshav Pingali. Compiling for distributed memory architectures, June 1992. Sent by the first author, when asked for her PhD.

[222] Gary Sabot and Skef Wholey. Cmax: A fortran translator for the connection machine system. In *ACM International Conference on Supercomputing*, pages 147–156, July 1993.

[223] Robert Schreiber, Charles Koelbel, and Joel Saltz. reductions. `hpff-task` list, January 2, 1996.

[224] Robert S. Schreiber. An Introduction to HPF. In *PRS Spring School, Les Menuires*, March 1996. To be published in the LNCS series.

[225] Alexander Schrijver. *Theory of linear and integer programming.* Wiley, New-York, 1986.

[226] Henk J. Sips, Kees van Reeuwijk, and Will Denissen. Analysis of local enumeration and storage schemes in HPF. In *ACM International Conference on Supercomputing*, May 1996.

[227] Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. The solomon computer. In *Proceedings of the Fall 1962 Eastern Joint Computer Conference*, pages 97–107, December 1962.

[228] J. Stichnoth, O'Hallaron D., and T. Gross. Generating communication for array statements: Design, implementation and evaluation. *Jounal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.

[229] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation and evaluation. In *Language and Compilers for Parallel Computing*, August 1993.

[230] James M. Stichnoth. Efficient compilation of array statements for private memory multicomputers. CMU-CS-93 109, School of Computer Science, Carnegie Mellon University, February 1993.

[231] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Publishing Company, 1994.

[232] E. Su, D.J. Palermo, and P. Banerjee. Automatic parallelization of regular computations for distributed memory multicomputers in the paradigm compiler. In *1993 Int. Conf. on parallel processing*, pages II30–II38. CRC Press, 93.

[233] Ernesto Su, Antonio Lain, Shankar Ramaswamy, Daniel J. Palermo, Eugene W. Hodges IV, and Prithviraj Banerjee. Advanced Compilation Techniques in the Paradigme Compiler for Distributed-Memory Multicomputers. In *ACM International Conference on Supercomputing*, pages 424–433, July 95.

[234] Rajeev Thakur, Alok Choudhary, and Geoffrey Fox. Runtime array redistribution in HPF programs. In *Scalable High Performance Computing Conference*, pages 309–316, 1994.

[235] Rajeev Thakur, Alok Choudhary, and J. Ramanujam. Efficient Algorithms for Array Redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 1996. To appear.

[236] Thinking Machine Corporation, Cambridge, Massachussetts. *C* Programming guide*, version 6.0 edition, November 1990.

[237] Thinking Machine Corporation, Cambridge, Massachussetts. *CM Fortran Programming Guide*, January 1991.

[238] Ashwath Thirumalai and J. Ramanujam. Fast Address Sequence Generation for Data-Parallel Programs using Integer Lattices. In *Language and Compilers for Parallel Computing*, pages 13.1–13.19, August 1995.

[239] Rémi Triolet. *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédures.* PhD thesis, Université Paris VI, 1984.

[240] Rémi Triolet, Paul Feautrier, and François Irigoin. Direct parallelization of call statements. In *Proceedings of the ACM Symposium on Compiler Construction*, volume 21(7), pages 176–185, July 1986.

[241] Chau-Wen Tseng. *An Optimising Fortran D Compiler for MIMD Distributed Memory Machines.* PhD thesis, Rice University, Houston, Texas, January 1993.

[242] Russ Tuck. *Porta-SIMD: An Optimaly Portable SIMD Programming Language.* PhD thesis, University of North Carolina at Chapel Hill, May 1990.

[243] Louis H. Turcotte. A survey of software environments for exploiting networked computing resources. Technical report, Engineering Research Center for Computational Field Simulation, Mississippi State, USA, 1993.

[244] S. H. Unger. A computer oriented toward spatial problems. In *Proceedings of the IRE*, pages 1744–1750, October 1958.

[245] Vincent Van Dongen. Compiling distributed loops onto SPMD code. *Parallel Processing Letters*, 4(3):301–312, March 1994.

[246] Vincent Van Dongen. Array redistribution by scanning polyhedra. In *PARCO*, September 1995.

[247] C. van Reeuwijk, H. J. Sips, W. Denissen, and E. M. Paalvast. Implementing HPF distributed arrays on a message-passing parallel computer system. Computational Physics Report Series, CP-95 006, Delft University of Technology, November 1994.

[248] A. Veen and de Lange M. Overview of the PREPARE Project. In *4th Int. Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, 1993.

[249] Jian Wang and Christine Eisenbeis. Decomposed software pipelining: A new approach to exploit instruction level parallelism for loop programs. In *Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, January 1993.

[250] Doran K. Wilde and Sanjay Rajopadhye. Allocating memory arrays for polyhedra. Research Report 2059, INRIA, July 1993.

[251] Gregory Wilson. Timeline. News `comp.parallel`, article 4663, August 1993.

[252] Michael J. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[253] Jingling Xue. Constructing do loops for non-convex iteration spaces in compiling for parallel machines. In *International Parallel Processing Symposium*, April 1995.

[254] Yi-Qing Yang. *Tests des Dependances et Transformations de Programme.* PhD thesis, Université Paris VI, November 1993.

[255] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations. In *Language and Compilers for Parallel Computing*, July 1994. Also available as TR EMP A/266/CRI.

[256] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations (extended version). *Int. J. of Parallel Programming*, 23(4):259–388, August 1995.

[257] Lei Zhou. *Statical and Dynamical Analysis of Program Complexity.* PhD thesis, Université Paris VI, September 1994. .

[258] Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. Vienna Fortran - A Language Specification. ftp `cs.rice.edu`, 1992. Version 1.1.

[259] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

[260] Hans Zima and Barbara Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, February 1993.

[261] Hans Peter Zima, H. J. Bast, and Hans Michael Gerndt. SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, 6:1–18, 1988.

Cet ouvrage a été composé par l'auteur avec LaTeX et BibTeX.
En particulier le style `french` de Bernard Gaulle [102] a été utilisé.
Quelques ouvrages généraux de typographie [135, 47] ont
également contribué à la justesse de cette composition.
Les documentations de Leslie Lamport [168] et de
Michel Goosens, Frank Mittelbach et Alexander Samarin [109]
ont été indispensables.
Enfin je tiens à remercier particulièrement Ronan Keryell pour son aide.