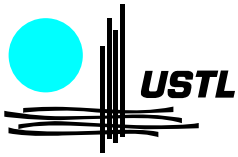
Contributions aux environnements de programmation pour le calcul intensif

Pierre Boulet

N° d'ordre : H355



Mémoire présenté par

Pierre Boulet

pour obtenir

l'habilitation à diriger des recherches
en Sciences Mathématiques (spécialité informatique)

Contributions aux environnements de programmation pour le calcul intensif

---

2 décembre 2002

Composition du jury :

Président :	Jean-Marc Geib
Rapporteurs :	Michel Cosnard
	Paul Feautrier
	Jean Roman
Directeur de recherches :	Jean-Luc Dekeyser
Examinateurs :	Bernard Dion
	Philippe Kajfasz
	Yves Sorel

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
LIFL – UMR 8022 – Bât. M3 – UFR IEEA – 59655 Villeneuve d'Ascq cedex
Tél. : (33) 03 20 43 47 24 – Fax. : (33) 03 20 43 65 56 – Mél : Pierre.Boulet@lifl.fr

Résumé

Mes recherches concernent les outils de développement pour le calcul intensif. Une application sera dite intensive s'il faut fortement l'optimiser pour obtenir la puissance de calcul requise pour répondre aux contraintes, d'une part, de temps d'exécution et, d'autre part, de ressources de la plateforme d'exécution. De telles applications se retrouvent aussi bien dans le domaine du calcul scientifique que dans le domaine du traitement de signal intensif (télécommunications, traitement multimédia). Les difficultés de développement de telles applications sont principalement l'exploitation du parallélisme des architectures d'exécution (des supercalculateurs aux systèmes sur silicium en passant par les grappes de stations de travail), l'hétérogénéité de ces mêmes architectures et le respect des contraintes de temps et de ressources.

Le but de mes recherches est de proposer des outils permettant la programmation efficace des applications de calcul intensif. Ceux-ci peuvent être des compilateurs, des paralléliseurs ou des environnements de spécification. Mes travaux ont commencé par les compilateurs paralléliseurs et s'orientent de plus en plus vers les environnements de spécification. Ces environnements comportent des compilateurs paralléliseurs. Cette évolution consiste donc à remplacer le langage de programmation et la phase d'analyse des programmes par une spécification des algorithmes de plus haut niveau et facilitant la phase d'analyse de dépendances. En effet, le but de cette analyse est de retrouver l'essence de l'algorithme codé par le programme analysé. La spécification de l'algorithme par les seules dépendances de données permet d'éliminer l'analyse de dépendances et d'avoir toute l'information nécessaire pour les optimisations du compilateur paralléliseur.

Quatre principes dirigent mes recherches :

1. Programmer au plus haut niveau possible. Il ne devrait pas être de la responsabilité du programmeur de gérer les détails de l'exécution de son application. Idéalement, il devrait exprimer son algorithme et les compilateurs devraient générer le code le plus efficace possible sur l'architecture visée.
2. Promouvoir le parallélisme de données. Ce paradigme de programmation permet justement une programmation de haut niveau dans bien des cas. Il est bien adapté au calcul intensif où les traitements sont souvent réguliers et la quantité de données manipulées importante.
3. Optimiser au plus tôt dans la chaîne de développement. Je suis convaincu que plus les informations de performances et les optimisations sont faites tôt dans le développement d'une application, plus ce développement sera rapide et l'application efficace. L'environnement de conception doit donc faire apparaître ces informations si elles sont disponibles, y compris avant compilation de l'application.
4. Restreindre le domaine d'application. Il est très difficile d'optimiser tous les programmes en général. Le domaine du calcul intensif lui-même est déjà ambitieux. En se focalisant sur un domaine d'application précis, on peut espérer réduire la variété des applications et ainsi proposer des optimisations adaptées. C'est la voie que j'ai suivie dans mes recherches les plus récentes en restreignant le domaine d'application au traitement de signal intensif.

Table des matières

Annexes

À ma grand-mère pour avoir su transmettre
sa passion de l'enseignement à son fils et à travers lui à son petit-fils.

Remerciements

Je tiens à remercier tout d'abord les différentes personnes qui m'ont accueilli dans leur équipe de recherche depuis la fin de ma thèse et grâce à qui j'ai pu mener à bien ces recherches, par ordre chronologique : Paul Feautrier, Yves Robert et Jean-Luc Dekeyser. J'ai énormément appris sur le métier de chercheur à leur contact et si je me sens aujourd'hui capable d'encadrer des recherches, c'est grâce à leur exemple.

Ensuite, ces travaux sont avant tout un travail d'équipe. Un grand merci donc à tous les gens avec qui j'ai collaboré tout au long de ces six dernières années :

- Denis, Vincent, Michel, Albert, Jean-François et Paul au PR*i*SM ;
- Alain, Frédo, Georges, Pierre-Yves et Yves au LIP ;
- Jack Dongarra à l'université du Tenessee à Knoxville ;
- les permanents de l'équipe WEST du LIFL : les deux Jean-Luc, Philippe et Xavier ;
- et bien sûr tous les autres membres de l'équipe WEST et en particulier les étudiants avec qui j'ai travaillé : Julien, Denis, Thierry, Manu, Florent, Abdelkader, Philippe, Ouahid et Michaël, sans oublier tous les stagiaires de DESS, de maîtrise ou d'IUP GMI sans qui les prototypes logiciels ne seraient pas ce qu'ils sont.

Merci ensuite aux rapporteurs qui m'ont fait l'honneur de donner leur appréciation sur ce manuscrit. J'ai connu Michel Cosnard à l'ENS Lyon dès mon arrivée dans cette école en 1990 où j'ai toujours aimé les anecdotes historiques qui agrémentaient ses cours. Il a ensuite dirigé le LIP où j'ai effectué ma thèse et présidé mon jury de thèse. C'est toujours un plaisir de le rencontrer. Paul Feautrier m'a accueilli à Versailles pour mon service national comme scientifique du contingent. J'y ai passé une excellente année et nous nous croisons depuis régulièrement dans les GDR ou Actions Spécifiques du CNRS auxquelles nous participons. Comme beaucoup, j'écoute toujours attentivement ses interventions pour ses immenses connaissances et expériences. Je connais moins personnellement Jean Roman mais nous sommes amenés à nous revoir dans le cadre de l'INRIA Futurs. Je remercie encore les trois rapporteurs pour le temps qu'ils m'ont accordé.

Merci à Jean-Marc Geib d'avoir présidé mon jury et pour l'énergie qu'il déploie pour le LIFL. Il donne une dynamique très intéressante au laboratoire. Merci à Bernard Dion, Philippe Kajfasz et Yves Sorel d'avoir participé à mon jury et d'avoir apporté chacun un point de vue différent sur mon travail. Je suis sûr que nos collaborations au sein des projets européens sera fructueuse pour tous et que la confrontation de nos expériences diverses nous fera tous avancer.

La bonne ambiance qui règne au LIFL participe des bonnes conditions qui m'ont permis d'effectuer ce travail. Je remercie donc tous mes collègues avec qui il est agréable de travailler aussi bien en recherche qu'en enseignement.

Merci enfin aux gens extérieurs au monde du travail qui font que je me sens bien dans ma vie : ma parents, mes deux frères et ma grand-mère pour tant de choses ; les Gueux de l'Ovale Club de Phalempin pour le plaisir de jouer au rugby et leur expérience de la vie ; et tous mes amis pour leur amitié et les interminables parties de jeux de rôles qui nous font nous évader dans l'imaginaire.

Chapitre 1

Introduction

1.1 Cadre scientifique et économique

1.1.1 Où sont les besoins de calcul intensif ?

La définition du calcul intensif que je vais considérer fait intervenir la *puissance* de calcul nécessaire à l'exécution de l'application. La puissance de calcul est la quantité de calcul (nombre d'opérations élémentaires) utilisée par unité de temps. En effet, l'application peut devoir répondre à des contraintes de temps ou de ressources. Une quantité de calcul relativement modeste devant s'exécuter en un temps réduit peut demander une grande puissance de calcul.

Une application sera considérée intensive si son code doit être fortement optimisé pour utiliser au mieux les ressources de calcul disponibles et respecter les contraintes de temps.

Quand on pense au calcul intensif, on se réfère souvent aux *Grand Challenges*[1]. Un des points communs de ces projets de calcul intensif est qu'ils s'intéressent à la simulation. La plupart du temps, ce sont des projets de calcul scientifique, comme la modélisation du climat, la simulation d'explosions nucléaires, l'aéronautique, etc. Ce type d'application demande en général une énorme quantité de calcul et les contraintes de temps d'exécution *implicites* sont liées à l'échelle de temps des activités humaines : pour caricaturer, il ne sert à rien de prévoir le climat des années à venir en plusieurs siècles.

D'un autre côté, on assiste à une montée en puissance des applications de traitement du signal ou de l'image dans des domaines tels que les télécommunications (radiotéléphonie de 3^{e} génération) ou le multimédia (radio, télévision ou cinéma numériques). Ces applications, mêmes si elles sont relativement moins demandeuses en quantité de calcul doivent souvent répondre à des contraintes de temps qui les rendent gourmandes en *puissance* de calcul. De plus, elles sont souvent embarquées. Des contraintes de taille et de consommation énergétiques se superposent alors aux contraintes de temps pour nécessiter une forte optimisation de leurs codes.

Il existe beaucoup d'autres domaines requérant des ressources importantes et qui ne font pas l'objet de cette thèse. On peut citer en particulier la fouille de données qui est de plus en plus répandue. Les applications de la fouille de données vont des biotechnologies aux applications commerciales. Elles nécessitent surtout des ressources de mémoire gigantesques et non de calcul proprement dit. Elles ne seront donc pas au cœur de mes préoccupations.

[1] Voir par exemple `http://www.nhse.org/grand_challenge.html` pour une liste de tels projets.

1.1.2 Machines capables de répondre à ces besoins

L'architecture des ordinateurs dédiés au calcul intensif tel que défini ci-dessus évolue rapidement. On peut distinguer quatre grandes classes de machines adaptées à différents types d'applications :

- les supercalculateurs ;
- les grappes de stations de travail (ou *commodity clusters*) ;
- les systèmes distribués et grilles de calcul ;
- les systèmes sur silicium (ou *System on Chip*, SoC).

Supercalculateurs

Les ordinateurs les plus puissants [82] depuis plusieurs années sont des machines massivement parallèles (plusieurs milliers de processeurs). Leur organisation est généralement centrée sur un réseau de communication à haut débit (de l'ordre de quelques Go/s à quelques dizaines de Go/s) auquel sont connectés des nœuds de calcul constitués d'ensembles de quelques (de 4 à 16) processeurs, d'architecture RISC ou plus rarement vectorielle, partageant une mémoire commune. Des dispositifs de stockage de masse sont connectés aussi bien aux nœuds de calcul que directement sur le réseau d'interconnexion.

Leur puissance de calcul ne cesse d'augmenter [63] pour dépasser la dizaine de TFlops (milliers de milliards d'opérations en arithmétique à virgule flottante à la seconde) et sont utilisés pour des simulations de plus en plus ambitieuses.

Grappes de stations de travail

Ces *clusters*, tels qu'on les nomme habituellement, sont des architectures apparues il y a une dizaine d'années[2] et qui prennent de plus en plus d'importance. En effet, il y a 80 clusters dans la liste du Top500 de juin 2002.

La particularité de ces architectures [4] est qu'elles sont construites à partir de composants bon marché du commerce. En effet, elles consistent à regrouper un certain nombre de stations de travail autour d'un réseau de communication. Cette architecture ressemble beaucoup à celle des supercalculateurs décrits ci-dessus. En voici les différences les plus significatives à mon avis :

- *nœuds de calcul :* les nœuds des clusters sont des ordinateurs à part entière alors que ceux des supercalculateurs ne sont pas autonomes, ils sont directement conçus pour être connectés au réseau d'interconnexion ; le nombre de processeurs (se partageant la mémoire) par nœud est en général moins important dans les clusters, un, deux ou quatre ;
- *réseau d'interconnexion :* ceux des clusters sont la plupart du temps des bus, certes à haut débit mais supportant moins de charge que ceux des supercalculateurs ; des architectures à base de commutateurs proposent cependant des performances comparables à celles des réseaux des supercalculateurs ;
- *système d'exploitation :* sur l'immense majorité des clusters, c'est le système libre GNU/Linux qui est utilisé en raison de la disponibilité d'outils, de la possibilité de le modifier pour l'adapter à chaque système et de sa gratuité, alors que les supercalcu-

[2] Le premier cluster est la machine *Beowulf* construite en 1994 au *Center for Excellence in Space Data and Information Sciences (CESDIS)*.

lateurs sont en général livrés avec le système du constructeur, certes optimisé pour la machine mais moins flexible (et beaucoup plus cher).

En résumé les clusters sont moins couplés que les supercalculateurs et permettent d'obtenir un rapport puissance/prix très avantageux au prix de performances un peu moins bonnes, en particulier sur les applications fortement couplées.

Un cluster nécessite des outils logiciels le faisant apparaître comme un unique ordinateur vis-à-vis des applications. Ces intergiciels permettent de programmer les clusters avec les mêmes langages que les supercalculateurs. La frontière entre clusters et supercalculateurs est de ce fait assez floue.

Systèmes distribués et grilles de calcul

Les systèmes distribués, a priori destinés à des applications mettant en relation des entités réparties ont évolués vers le métacalcul consistant à exécuter une unique application sur un tel système. Deux façons assez répandues de considérer le métacalcul sont les grilles de calcul et les calculs distribués sur l'internet.

Les grilles de calcul [42] sont une architecture en plein développement. Elles consistent en un réseau d'ordinateurs faiblement couplés et ont pour but d'offrir une très grande puissance de calcul à leurs utilisateurs de la façon la plus transparente possible. Ces ordinateurs peuvent être des supercalculateurs, des clusters ou des stations de travail ordinaires. Ils sont reliés par un réseau à très grande échelle, le plus souvent l'internet. De ce fait, les grilles sont plus un ensemble de logiciels permettant de répartir et d'exécuter des applications sur un parc de machines hétérogènes et placées sous des autorités administratives différentes.

Le support logiciel nécessaire à l'utilisation de grilles de calcul est encore plus important que celui des clusters. Il faut en particulier gérer les transparences de localisation, de nommage, d'accès et d'architecture des divers ordinateurs composant la grille. Ce genre d'architecture permet l'exécution d'applications nécessitant une puissance de calcul considérable mais faiblement couplées.

Pour des applications encore plus faiblement couplées, les systèmes de distribution de calculs sur l'internet sont devenus populaires. Il est fait ici appel à la bonne volonté du public pour la mise à disposition de temps de calcul. Chaque participant télécharge un logiciel client qui se charge de communiquer avec un serveur centralisé chargé de lui octroyer une partie du calcul. Les applications de ces systèmes[3] sont, par exemple, des simulations distribuées (astronomie, génome, chimie, etc) ou des calculs mathématiques (recherche de nombres premier, problèmes de factorisation).

Ces applications nécessitent de telles quantités de calcul que l'utilisation d'un maximum de ressources de calcul prime sur l'optimisation du code. Les problèmes de partitionnement et de distribution des calculs, de tolérance aux pannes sont centraux à leur développement.

Systèmes sur silicium

Les systèmes sur silicium, ou SoC, sont des architectures hautement intégrées regroupant sur une même puce au moins un processeur programmable, de la mémoire et des unités de traitement accélératrices câblées. Cette puce intègre aussi les interfaces aux matériels périphériques ou au monde extérieur. Ces systèmes sont particulièrement adaptés aux applications intensives embarquées qui comportent une partie de traitement intensif du signal ou de l'image.

[3] Voir `http://www.aspenleaf.com/distributed/` pour une liste de tels projets.

Ces SoC sont composés de composants virtuels réutilisables, tels que des cœurs de processeurs, des unités de traitement du signal (DSP), du matériel de contrôle de protocoles, des blocs analogiques, des unités matérielles dédiées ou des bus intégrés sur la puce. Certaines de ces unités peuvent être parallèles, en particulier celles qui sont dédiées au traitement de signal intensif.

1.1.3 Difficulté de développement sur de telles machines

Voyons maintenant les caractéristiques de ces différentes architectures pour le traitement intensif qui les rendent difficiles à programmer efficacement.

Parallélisme

Toutes les architectures présentées précédemment possèdent plusieurs unités de traitement. Ce parallélisme peut aller de plusieurs milliers de processeurs pour les plus imposants supercalculateurs à quelques unités pour les petits clusters ou les SoC. Un autre parallélisme est aussi présent dans chaque unité avec les pipelines et les unités SIMD ou VLIW.

Pour tirer le meilleur parti de la puissance de calcul disponible, il faut transformer l'application pour qu'elle utilise à chaque instant toutes les ressources de calcul disponibles. Ceci est très difficile à obtenir. Ce processus de placement spatio-temporel des opérations constituant l'application sur l'architecture d'exécution est NP-complet dans tous les cas non triviaux [31].

Différentes approches tentent de proposer des outils pour exploiter le parallélisme potentiel de ces machines :

- *La parallélisation automatique* propose de transformer sans intervention humaine un programme séquentiel en un programme parallèle. Elle suppose l'analyse des dépendances de données du programme séquentiel pour en exhiber le parallélisme puis le placement sur l'architecture d'exécution. Même si des solutions existent pour des noyaux de calcul, c'est encore largement un problème ouvert pour une application réelle.
- *La programmation parallèle explicite* est pour le moment réservée aux spécialistes. Elle impose la plupart du temps la gestion de bas niveau du partage des calculs et des données. Elle permet l'obtention de bonnes, voire d'excellentes performances au prix d'efforts importants. Notons que des langages comme High Performance Fortran ou OpenMP permettent une parallélisation incrémentale de l'application. Des performances raisonnables sont alors possibles avec relativement peu d'efforts.
- *La programmation parallèle implicite* oblige le programmeur à n'exprimer aucune dépendance de données superflue dans son programme. Ainsi, le programmeur laisse le maximum de liberté au compilateur pour exploiter le parallélisme potentiel de l'application. On peut la rapprocher de la parallélisation automatique où la phase d'analyse du programme est supprimée et où l'information disponible est la meilleure possible.
- *La conception conjointe* du logiciel et du matériel diffère des approches précédentes dans le sens où l'architecture d'exécution n'est pas figée lors de l'écriture du programme. En effet, elle est conçue conjointement au programme, et donc une adaptation mutuelle du programme et de l'architecture permettent une meilleure adéquation entre les deux et donc de meilleures performances.

Le choix de telle ou telle méthode de conception dépendra de plusieurs facteurs : l'expertise des programmeurs, la pré-existance ou non d'un programme séquentiel, l'architecture cible, les outils disponibles, les ressources humaines... Les travaux que je présente au chapitre 2

concernent la parallélisation automatique, ceux du chapitre 3 se classent dans la programmation explicite, ceux des chapitres 4 et 5 se rapprochent de la programmation parallèle implicite et les perspectives présentées au chapitre 6 tendent vers la conception conjointe.

Hétérogénéité

L'hétérogénéité éventuelle de l'architecture cible ajoute un niveau de difficulté (et de liberté) supplémentaire. Cette hétérogénéité est une des caractéristiques fondamentales des systèmes distribués et des SoC. Elle peut aussi intervenir dans les supercalculateurs et les clusters, en particulier au niveau des performances relatives des différents nœuds de calcul. En effet, une telle machine est en générale homogène lors de sa conception mais devient hétérogène lors de son évolution par l'ajout ou le remplacement par des nœuds plus puissants. Le système d'exploitation et les outils logiciels restent cependant en général homogènes.

Il y a deux grandes classes d'hétérogénéité :

- *hétérogénéité de performances,* lorsque les différentes unités de calcul ont les mêmes fonctions mais avec des performances variables ;
- *hétérogénéité de fonction,* lorsque tous les calculs ne peuvent pas être effectués sur toutes les unités.

Le travail présenté au paragraphe 2.4.2 prend en compte l'hétérogénéité de performances en proposant un découpage statique des calculs permettant l'équilibre des charges alors que les perspectives du chapitre 6 s'attachent plus à l'hétérogénéité de fonction.

Contraintes

Enfin, la recherche de performances des applications de calcul intensif est motivée par le respect de contraintes de temps et de ressources de calcul. Dans mes travaux, ces contraintes sont souvent abstraites pour être traduites en « calculer le plus vite possible avec les ressources disponibles ». Nous verrons dans les perspectives que la formalisation des contraintes matérielles sera plus importante dans la suite de mes travaux.

1.2 Positionnement

1.2.1 Domaine scientifique

Toutes mes recherches concernent les *outils de développement*. Mon but est de proposer des outils permettant la programmation efficace des applications de calcul intensif. Ceux-ci peuvent être des compilateurs, des paralléliseurs ou des environnements de spécification.

Mes travaux ont commencé par les compilateurs paralléliseurs (chapitre 2) et s'orientent de plus en plus vers les environnements de spécification (chapitre 5). Ces environnements comportent des compilateurs paralléliseurs. Cette évolution consiste donc à remplacer le langage de programmation et la phase d'analyse des programmes par une spécification des algorithmes de plus haut niveau et facilitant la phase d'analyse de dépendances. En effet, le but de cette analyse est de retrouver l'essence de l'algorithme codé par le programme analysé. La spécification de l'algorithme par les seules dépendances de données permet d'éliminer l'analyse de dépendances et d'avoir toute l'information nécessaire pour les optimisations du compilateur parallèliseur.

1.2.2 Idées directrices

Quatre principes dirigent mes recherches :

1. *Programmer au plus haut niveau possible.* Il ne devrait pas être de la responsabilité du programmeur de gérer les détails de l'exécution de son application. Idéalement, il devrait exprimer son algorithme et les compilateurs devraient générer le code le plus efficace possible sur l'architecture visée. Bien sûr, cette situation est pour l'instant utopique dans le cas du calcul intensif, mais j'essaie d'aller dans ce sens.
2. *Promouvoir le parallélisme de données.* Ce paradigme de programmation permet justement une programmation de haut niveau dans bien des cas. Il est bien adapté au calcul intensif où les traitements sont souvent réguliers et la quantité de données manipulées importante.
3. *Optimiser au plus tôt dans la chaîne de développement.* Je suis convaincu que plus les informations de performances et les optimisations sont faites tôt dans le développement d'une application, plus ce développement sera rapide et l'application efficace. L'environnement de conception doit donc faire apparaître ces informations si elles sont disponibles, y compris avant compilation de l'application (voir les paragraphes 2.3 et 5.3). Les optimisations statiques (à la compilation) donnent de bonnes performances dans le cas où à la fois l'application et l'architecture sont connues car on ne paye qu'une fois le coût de l'optimisation et rien à l'exécution (voir les paragraphes 2.4 et 5.4). Dans certains cas cependant, on peut être amené à se reposer sur des optimisations à l'exécution (équilibrage de charge dynamique par exemple), en particulier dans le cas où les charges de calcul ou l'architecture d'exécution évoluent au cours du temps (voir le chapitre 4).
4. *Restreindre le domaine d'application.* Il est très difficile d'optimiser tous les programmes en général. Le domaine du calcul intensif lui-même est déjà ambitieux. En effet, la plupart des optimisations concernent les parties les plus intensives du code, à savoir les boucles [2,31] qui concentrent en peu de lignes une grande partie du temps d'exécution, encore faut-il que la forme de ces boucles soit comprise par l'optimiseur. En se focalisant sur un domaine d'application précis, on peut espérer réduire la variété des applications et ainsi proposer des optimisations adaptées. C'est la voie que j'ai suivie dans le chapitre 5 et que je poursuivrai dans la suite de mes recherches (voir le chapitre 6) en restreignant le domaine d'application au traitement de signal intensif.

1.3 Organisation du manuscrit et activités d'encadrement

Entre ma soutenance de thèse en janvier 1996 et mon arrivée dans l'équipe WEST du LIFL en septembre 1998 j'ai fait un an et demi de travail postdoctoral dans les équipes de Paul Feautrier au PR*i*SM à Versailles et d'Yves Robert au LIP à Lyon. Les travaux présentés au paragraphe 2.4 sont le fruit de ce travail. Le reste de mes travaux a été réalisé dans l'équipe WEST du LIFL où j'ai progressivement encadré de jeunes chercheurs sous la direction de Jean-Luc Dekeyser.

Le reste de ce manuscrit présente en quatre temps les travaux que j'ai effectués depuis ma soutenance de thèse. Dans un premier temps, le chapitre 2 exposera mes recherches concernant la parallélisation automatique et la compilation effectuées de 1996 à 2000. J'ai sur ce thème

encadré le stage de DEA de Denis Ruckebusch sur l'évaluation du volume de communications produit par une distribution de données en High Performance Fortran.

Dans le chapitre 3, je présenterai la parallélisation d'une application de recherche en électromagnétisme. Ce travail de parallélisation en vraie grandeur correspond à la thèse d'Emmanuel Cagniot dont j'ai suivi les travaux de 1998 à 2001.

Le chapitre 4 développera un support d'exécution distribué réalisé en CORBA sur le modèle des réseaux de processus. Ce travail est le fruit du stage de DEA et de la thèse d'Abdelkader Amar que je co-encadre depuis début 2000.

Le chapitre 5 exposera mes travaux autour des environnements de programmation visuels pour le traitement de signal intensif. Je co-encadre à ce propos la thèse de Philippe Dumont débutée en septembre 2001 sur le placement et la génération de code sur SoC.

Enfin, je conclurai et présenterai des perspectives à ces travaux dans le chapitre 6. Je parlerai en particulier des travaux envisagés dans la thèse de Mickaël Samyn qui vient de débuter sur la simulation répartie d'algorithmes de traitement de signal intensif sur SoC.

Chapitre 2

Parallélisation automatique

Dans la poursuite de mes travaux de thèse (intitulée « outils pour la parallélisation automatique »), de 1996 à 2000, j'ai continué à m'intéresser à différentes phases de la parallélisation et de la compilation de programmes de calcul à haute performance.

2.1 Philosophie et but des différentes études

Le but général est ici de fournir des outils pouvant être intégrés dans un compilateur paralléliseur d'un langage comme Fortran ou encore dans un environnement de programmation d'un tel langage. Les programmes étudiés font partie de la famille des programmes à contrôle statique dont on peut déterminer les dépendances de données à la compilation.

On s'intéresse ici en général à optimiser les boucles imbriquées qui regroupent en peu de lignes de code la majeure partie du temps de calcul. Le modèle sous-jacent utilisé dans la communauté de la parallélisation automatique est le modèle polyédrique. Dans ce modèle les bornes de boucles sont des fonctions affines des indices des boucles englobantes et les fonctions d'accès aux tableaux sont, elles aussi, des fonctions affines des indices de boucles. Ce modèle permet la représentation des espaces d'itération par des polyèdres et permet par calcul polyédrique de calculer un grand nombre d'informations concernant ces programmes comme l'expression des dépendances de données [24, 25, 37, 73], des calculs d'ordonnancement [29, 30, 38, 39, 86] et de placement de tâches parallèles [33, 40], des optimisations de génération de code [11, 27, 54], etc.

Je présenterai dans un premier temps, au paragraphe 2.2, un outil de calcul polyédrique symbolique, SPPoC [13] que j'ai développé avec Xavier Redon et qui a été utilisé pour les calculs des paragraphes 2.3 et 2.4.1. Ensuite, au paragraphe 2.3, je montrerai comment obtenir dans l'environnement de programmation, avant compilation, une estimation du volume de communication engendré par une opération de tableaux en High Performance Fortran selon la distribution des données choisie. Enfin, le paragraphe 2.4 s'intéressera à la parallélisation automatique avec une synthèse des algorithmes de parallélisation vus sous l'angle de la génération de code et une première étude concernant la répartition des calculs sur une architecture d'exécution hétérogène. Tous ses travaux, s'ils paraissent un peu décousus sont à la base de l'évolution vers mes recherches plus récentes exposées aux chapitres 4 et 5. J'essaierai donc ici de faire ressortir la démarche sans trop entrer dans les détails. Ceux-ci sont disponibles en annexe.

2.2 SPPoC : un outil de calcul polyédrique

Il existe de nombreux outils de calcul polyédrique numérique, mais nous n'en connaissons que trois qui permettent de manipuler des expressions paramétriques. Il s'agit de PIP [36,41], de la PolyLib [26, 85] et de l'Omega Library [53, 72]. PIP et la PolyLib étant fortement utilisés dans la communauté de la parallélisation automatique, nous avons réalisé la couche symbolique qui les rendra plus agréables et accessibles. Ces développements sont regroupés dans une bibliothèque : SPPoC pour *Symbolic Parameterized Polyhedral Calculator*. À la suite de demandes de la dite communauté, l'Omega Library est aussi intégrée dans SPPoC bien que disposant déjà d'une interface symbolique.

PIP permet de résoudre des problèmes de programmation linéaire paramétrée en nombres entiers. Un tel problème est la donnée d'un polyèdre définissant le domaine de recherche, d'un autre polyèdre définissant le domaine des valeurs des paramètres et d'une liste de variables. PIP retourne le minimum lexicographique de ces variables dans leur domaine de validité sous la forme d'un QUAST (*Quasi Affine Selection Tree*). La PolyLib, quant à elle, permet la manipulation de polyèdres paramétrés (définition, opérations ensemblistes, application de fonctions affines et de leur réciproque, comptage du nombre de points, etc). Enfin l'Omega Library qui manipule l'arithmétique de Presburger est partiellement redondante avec PIP et la PolyLib mais apporte d'autres fonctionnalités comme le calcul d'une fermeture transitive de relation.

Le premier inconvénient de ces outils (du moins en ce qui concerne PIP et la PolyLib) est leur difficulté d'utilisation venant principalement des structures de données de bas niveau utilisées. En effet, les polyèdres et autres structures de données sont représentés par des matrices de coefficients. Les variables et les paramètres sont repérés par leur indice de ligne ou de colonne. Ces représentations, bien qu'efficaces pour le calcul, ne sont que très peu lisibles. Le second inconvénient (et certainement le plus important) est que les résultats ne sont pas forcément sous leur forme la plus simple. Cela se fait cruellement sentir quand on enchaîne plusieurs calculs. La complexité des objets traités augmente alors rapidement, rendant impossibles de nouveaux calculs.

Pour résoudre ces deux problèmes, nous avons développé une interface unifiée à ces outils. Elle est complètement symbolique, donc beaucoup plus utilisable, et fait des simplifications qui permettent des enchaînements de calculs jusqu'alors impossibles.

Plus qu'une simple interface, SPPoC offre des fonctionnalités supplémentaires : prise en charge des changements de variables lors des appels à PIP, amélioration de certaines fonctions de la PolyLib (gestion automatique des dimensions et comptage de points dans un produit cartésien), et un moteur de simplification d'expressions arithmétiques et de systèmes de contraintes.

Le premier objectif de SPPoC est atteint : permettre l'implantation des deux applications présentées aux paragraphes 2.3 et 2.4.1. Pour bien comprendre le chemin parcouru, il faut savoir qu'une tentative de calcul du volume des communications sur l'exemple donné au paragraphe 2.3 a été faite en utilisant directement PIP et la PolyLib. Dans un premier temps, aucun résultat n'a pu être obtenu (mémoire insuffisante, temps de calcul prohibitif). Dans un second temps, après découpage manuel du problème, un résultat est finalement sorti. Le volume calculé tenait sur deux pages de texte et était, bien entendu, inexploitable.

Enfin, SPPoC est un logiciel libre diffusé sur l'internet à l'adresse `http://www.lifl.fr/west/sppoc/`. Le lecteur intéressé par les détails du fonctionnement de SPPoC pourra se reporter à l'annexe A.

2.3 Informations dans l'environnement de programmation

Dans le but de donner le maximum d'information permettant au programmeur d'optimiser au plus tôt son programme, je me suis intéresser à construire un indicateur d'un des paramètres cruciaux des performances d'un programme parallèle, le volume de communication induit par une affectation à un tableau distribué.

Je me suis placé dans le cas de la programmation data-parallèle et plus précisément en High Performance Fortran. Cette étude faisait suite aux travaux de Christian Lefebvre sur un outil, HPF-Builder [57] permettant la visualisation de la distribution des données choisie par le programmeur. Le but en était d'intégrer à HPF-builder un estimateur du volume de communication induit par une opération en fonction de la distribution des données. Ce problème a été étudié par d'autres chercheurs [35,46,55]. L'originalité de notre démarche a été de travailler au niveau du langage, de rester indépendant du compilateur et de l'architecture d'exécution. En effet, un programme efficace au niveau du langage ne devrait pas être mauvais lors de son déploiement ou plutôt, un mauvais programme au niveau du langage ne peut pas produire une exécution performante. L'objectif est de fournir au programmeur le plus tôt possible dans le cycle de développement de son application des informations sur les volumes de communication générés lui permettant d'éviter les plus gros écueils lors du choix de sa distribution des données.

Notre approche est complètement symbolique. Nous avons gardé comme paramètres du problème les tailles des tableaux manipulés pour fournir la réponse pour toutes les tailles. Le niveau d'abstraction de la machine d'exécution que nous avons choisi est le même que celui du langage, à savoir une grille multidimensionnelle de processeurs virtuels. Cette grille est appelée `PROCESSORS` en High Performance Fortran ; son placement sur l'architecture d'exécution est de la responsabilité du compilateur et le programmeur n'a aucun moyen de le contrôler. Toutes les distributions sont considérées, par blocs comme cycliques, avec réplication ou placement en mémoire. Les détails de cette modélisation sont présentés dans l'article [12].

Pour pouvoir pousser les calculs jusqu'au bout, nous avons restreint le champ d'application de cette évaluation aux programmes à contrôle statique. Nous sommes alors capables par une suite de calculs polyédriques en utilisant SPPoC de donner une expression symbolique du nombre de données devant être communiquées par une instruction d'affectation de tableaux. Sans perte de généralité, considérons une instruction lisant des données dans un tableau et écrivant les résultats d'un calcul sur ces données dans un autre tableau. Ces éléments de tableaux sont indicés par des fonctions affines des indices de boucles englobantes. Le domaine d'itération de ces indices forme un polyèdre. Dans le modèle de programmation à parallélisme de données, c'est le processeur qui possède la donnée résultat qui calcule l'opération (*owner computes rule*). Basé sur cette règle, on considère qu'il y a communication d'une donnée quand l'élément lu n'est pas sur le même processeur que l'élément écrit.

Un alignement HPF peut se concevoir comme la composition de l'inverse d'une fonction affine avec une fonction affine. La fonction inverse permet de formaliser l'éventuelle réplication de données. La modélisation d'une distribution HPF se base sur une projection ρ_T qui permet de sélectionner les dimensions du template qui sont distribuées sur les processeurs et sur un vecteur de paramètres κ_T. Les dimensions du template sont distribuées suivant le schéma `CYCLIC(k)`, le paramètre `k` étant donné, pour chaque dimension, par le vecteur de paramètres. Il est possible d'obtenir une distribution par blocs en choisissant correctement le paramètre `k`. Si les bornes inférieures et supérieures du template et du tableau de processeurs sont notées T_{min}, T_{max}, P_{min} et P_{max}, la fonction donnant les coordonnées du processeur sur lequel est

distribué un élément de template donné est

$$\pi_T(J) = P_{min} + (\rho_T(J - T_{min}) \div \kappa_T)\%(P_{max} - P_{min} + \mathbf{1}) \ .$$

Pour préciser la formule de calcul du volume de communications nous introduisons quelques notations :

- le domaine d'itération englobant l'instruction d'affectation est appelé $\mathcal{D}$ et le domaine de définition du template T est appelé $\mathcal{D}_T$;
- les fonctions d'accès aux éléments de template Φ_E (pour le tableau en écriture) et Φ_L (pour le tableau en lecture) sont définies comme la composition des fonctions d'alignement et des fonctions d'accès de ces tableaux ;
- la fonction π_T est, bien entendu, la fonction de distribution du template T.

Le volume de communications est alors donné par le nombre d'éléments de l'ensemble

$$\left\{(I, J) \mid J \in \mathcal{D}_T,\ I \in \Phi_E^{-1}(J),\ \forall K \in \Phi_L(I),\ \pi_T(J) \neq \pi_T(K)\right\} \ .$$

La façon de calculer ce nombre avec SPPoC est donnée au paragraphe 7.2 de l'annexe A.

2.4 Parallélisation automatique et optimisations

2.4.1 Algorithmes de parallélisation automatique et génération de code

Dès la fin de ma thèse je me suis intéressé au problème de la génération de code d'itération après transformation de boucles [9]. J'ai poursuivi dans cette voie pendant 2 ans environ. Ce travail a d'abord consisté en un état de l'art des algorithmes de parallélisation automatique avec une focalisation sur les transformations mises en œuvres et les techniques de générations de code associées [6]. J'ai ensuite, avec Paul Feautrier, proposé un algorithme de génération de code de bas niveau s'appliquant à tous les cas [11].

Tous les algorithmes de génération de code qui sont présentés dans [6] font appel à des techniques fortement liées aux types de transformations qu'ont subies les boucles. Dans le modèle polyédrique chaque instruction est associée à un domaine d'itération correspondant à l'ensemble des valeurs des indices des boucles qui l'englobent. En traduisant les bornes de ces boucles en inégalités on obtient un polyèdre paramétré pour chaque domaine d'itération. Les transformations de boucles reviennent à des transformations affines sur ces polyèdres. Le problème de la génération de code se ramène donc au problème du parcours des points entiers de polyèdres paramétrés. Les algorithmes présentés dans l'état de l'art utilisent des méthodes faisant un compromis entre l'efficacité du code généré et sa simplicité. Tous ces algorithmes proposent un code sous forme de boucles imbriquées.

La méthode présentée dans l'annexe B propose une démarche générale pour générer un code d'itération d'unions de polyèdres paramétrés, et même de l'union $\mathcal{U}(z) = \bigcup_{1 \leq i \leq p} \mathcal{L}_i(z)$ de réseaux linéairement bornés paramétrés définis par

$$\mathcal{L}_i(z) = \{x \in \mathbb{Z}^n \mid \exists y \in \mathbb{Z}^{m_i}, A_i x + B_i y + C_i z \leq d_i\} \ .$$

où $m_i, n, p_i, q \in \mathbb{N}^*$, $A_i \in \mathbb{Z}^{p_i} \times \mathbb{Z}^n$, $B_i \in \mathbb{Z}^{p_i} \times \mathbb{Z}^{m_i}$, $C_i \in \mathbb{Z}^{p_i} \times \mathbb{Z}^q$ et $d_i \in \mathbb{Z}^{p_i}$, avec z un vecteur de paramètres à valeurs dans un polyèdre $\mathcal{D} = \{z \in \mathbb{Z}^q \mid Ez \leq f\}$. Ces unions de réseaux sont capables de représenter les espaces d'itération de boucles non forcément parfaitement imbriquées ayant subi toutes sortes de transformations, y compris celles proposées par tous les algorithmes de parallélisation décrits dans [6].

Les points entier de $\mathcal{U}$ doivent être parcourus dans l'ordre lexicographique. L'idée de base de notre algorithme de génération de code est de construire une constante, first, et une fonction, next, définies par

$$\text{first} = \min_{\prec}\{y \in \mathcal{U}(z)\}$$

et

$$\begin{aligned}\text{next} : \mathcal{U}(z) &\to \mathcal{U}(z) \cup \{\bot\} \\ x &\mapsto \min_{\prec}\{y \in \mathcal{U}(z) \mid x \prec y\} \ ,\end{aligned}$$

où $\prec$ représente l'ordre lexicographique. Le calcul de la constante (paramétrée par z !), first, se fait par résolution du système linéaire en nombres entiers qui la définit. Pour next, le calcul est plus compliqué. Il fait appel à de multiples résolutions de problèmes linéaires. L'utilisation de SPPoC pour ce calcul est expliquée au paragraphe 7.1 de l'annexe A.

L'originalité du code généré est que nous n'avons pas essayé de reconstruire des boucles mais nous générons un code de plus bas niveau faisant intervenir principalement des tests (`IF`) et des sauts (`GOTO`). Cela nous donne plus de liberté et permet quand même l'écriture de code aussi efficace que celui des boucles quand c'est possible. Ce code n'est pas destiné à être lu par un humain, mais à être utilisé comme code intermédiaire par un compilateur. Les exemples présentés en annexe B montrent que les performances du code obtenu restent bonnes malgré le domaine d'application très étendu de la méthode.

Ces travaux trouveront une suite dans le cas particulier du traitement de signal intensif comme indiqué dans les perspectives au paragraphe 6.3.2.

2.4.2 Partitionnement des calculs sur une architecture hétérogène

Lors d'une visite à l'université du Tenessee à Knoxville dans l'équipe de Jack Dongarra j'ai étudié avec lui-même, Yves Robert et Frédéric Vivien une méthode de répartition des calculs sur un réseau hétérogène [10]. L'hétérogénéité considérée ici est seulement de performances, tous les nœuds du réseau de stations de travail utilisé étant capables des mêmes calculs. De plus, le problème considéré est la répartition des calculs issus d'une transformation de boucles appelée « pavage par des parallélépipèdes » (ou *tiling* en anglais). Les calculs sont donc ici uniformes. Je présenterai au chapitre 4 un modèle d'exécution permettant une distribution de calculs non uniformes. L'hétérogénéité de l'architecture d'exécution prend de plus en plus d'importance dans mes travaux (voir les perspectives au chapitre 6).

Le tiling [22, 49] est une technique de partitionnement des calculs utilisée pour augmenter la granularité des calculs et augmenter la localité des références. Cette technique s'applique aux boucles parfaitement imbriquées. Les avantages du tiling sont une meilleure efficacité du calcul par une meilleure exploitation des unités de calcul pipelinées et de la hiérarchie mémoire, ainsi qu'une diminution du volume de communication. Des inconvénients possibles sont une éventuelle augmentation de la latence et une propension à un déséquilibre de la charge de calcul dû à la granularité élevée. Ce problème d'équilibrage devient fondamental lorsque l'architecture d'exécution est hétérogène.

La répartition des blocs aux processeurs est étudiée dans l'annexe C où après une solution exacte mais de temps de calcul prohibitif par programmation linéaire, nous présentons une heuristique asymptotiquement optimale et une heuristique pratique de coût de calcul très raisonnable. Cette heuristique est évaluée sur un réseau de stations de travail. Les résultats

obtenus sont tout à fait satisfaisants. Le calcul de la répartition des blocs (par colonne) est si peu coûteux que cette technique peut être utilisée à la compilation si les performances relatives des nœuds de calcul sont connues ou à l'exécution si elles ne le sont pas. L'équilibrage de charge peut même être dynamique si la charge des machines change au cours du temps.

Cette étude clos ce chapitre consacré à mes travaux dans le domaine de la compilation et de la parallélisation automatique. Ils trouveront des échos dans les perspectives, en particulier au paragraphe 6.3.

Chapitre 3

Simulation numérique

Pour garder prise avec les applications réelles, j'ai suivi les travaux d'Emmanuel Cagniot concernant la parallélisation d'une application d'électromagnétisme. Ces travaux se sont placés dans le cadre d'une collaboration entre le LIFL et le L2EP Lille (Laboratoire d'Électrotechnique et d'Électronique de Puissance de Lille) entre 1998 et 2001 et ont donné lieu à de nombreuses publications [16–21]. L'application en question (résolution d'équations différentielles par gradient conjugué) a été parallélisée en High Performance Fortran en utilisant une bibliothèque pour les accès irréguliers aux tableaux. Une technique de décomposition de domaines a été appliquée pour augmenter le degré de parallélisme. Les résultats ont été très positifs et la collaboration avec le L2EP continue sur un autre sujet cette fois-ci lié à la simulation répartie de réseau de distribution d'électricité.

3.1 Approche génie logiciel

Le but poursuivi dans cette étude est, au-delà de la recherche de performances par parallélisation, d'essayer de construire une approche « génie logiciel » pouvant s'appliquer à d'autres applications. En effet, le code que nous avons parallélisé est un code de recherche en constante évolution. Il a donc fallu faire un compromis entre la recherche de performances et la maintenabilité de code.

3.1.1 Choix d'High Performance Fortran

La plupart des applications de calcul scientifique sont écrites en Fortran pour des raisons de performances et de bonne adaptation de ce langage à l'algèbre linéaire (notations de tableaux, fonctions intrinsèques, compilateurs optimiseurs, ...). L'application étudiée était écrite en Fortran 77 qui, s'il est très bien optimisé par les compilateurs, ne dispose d'aucun mécanisme permettant la structuration du code en vue de son évolutivité. S'est alors posé le choix du langage d'implémentation de la version parallèle. Les possibilités au moment du début des travaux étaient :

- Fortran 77 avec MPI [60] ou PVM [43] ;
- Fortran 95 avec MPI ou PVM ;
- High Performance Fortran [48] ;
- OpenMP [23].

Les priorités que nous nous sommes fixées étaient :

1. fournir un code maintenable par un non spécialiste ;
2. assurer une bonne portabilité du code sur différentes plateformes (station de travail, cluster, supercalculateur) ;
3. obtenir de bonnes performances.

Fortran 77 a été éliminé très rapidement et le travail de parallélisation a d'ailleurs commencé par un nettoyage du code et une réécriture en Fortran 95. OpenMP a été jugé manquant de compilateurs performants (en 1999) et n'assurant pas la portabilité sur des clusters à cause de son modèle à mémoire partagée. High Performance Fortran, bien qu'en fin de vie assurait une programmation de haut niveau et des compilateurs performants sur de nombreuses plateformes, en particulier le compilateur Adaptor [14] du GMD. Nous avons d'ailleurs collaboré avec son auteur principal, Thomas Brandes lors de cette étude.

3.1.2 Communications irrégulières

Comme nous le détaillerons dans la section suivante, l'application consiste en la construction et l'inversion d'une matrice creuse de grande taille. Pour des raisons de place mémoire, la matrice est stockée en mémoire sous forme CSR[1]. Cette disposition en mémoire des données provoque des communications irrégulières qu'High Performance Fortran ne gère pas efficacement. Cependant, bien que le motif de communication dépende des données, il évolue peu au cours d'une exécution de l'application.

Des mécanismes de gestion de ce type de communication tirant parti de la stabilité des motifs ont été développés. Nous avons choisi d'utiliser la bibliothèque Halos [5] qui optimise ce genre de communications tout en proposant une interface de relativement haut niveau. De plus Halos a été intégrée à Adaptor, ce qui nous a permis une utilisation facile est unifiée du couple High Performance Fortran/Halos.

3.2 Algorithmique pour l'électromagnétisme

Je vais présenter ici l'application que nous avons parallélisée et les deux parallélisations que nous avons réalisées.

3.2.1 Structure de l'application

L'étude porte sur la simulation du fonctionnement de machines électromagnétiques comme les moteurs électriques, par exemple. Je passe ici sous silence la modélisation et la formulation mathématique détaillée pour ne retenir que les caractéristiques principales du problème. Le lecteur intéressé pourra lire les détails dans [16].

Il s'agit de la résolution d'un système d'équations aux dérivées partielles, dérivé des équations de Maxwell, par la méthode des éléments finis. Ce système d'équations présente un certain nombre de caractéristiques qui le rendent difficile à résoudre :

– les formulations magnétostatiques et magnétodynamiques des équations de Maxwell sont des équations aux dérivées partielles respectivement elliptiques et paraboliques pour lesquelles la méthode des éléments finis est la plus appropriée ;

[1] *Compressed Sparse Row*

- la méthode de discrétisation fait appel aux éléments de Whitney faisant porter différentes grandeurs physiques par les nœuds, arêtes, facettes ou encore volume des éléments du maillage ; cette formalisation permet de garder les propriétés de continuité entre éléments du maillage ;
- le maillage fait appel à différents volumes élémentaires : tétraèdres, prismes et hexaèdres, cela offre de la souplesse au mailleur mais complique la construction du système et sa résolution ;
- les matrices obtenues par discrétisation par la méthode de Galerkin sont creuses, symétriques et définies ou semi-définies positives ; ce genre de système est résolu par la méthode du gradient conjugué ;
- les machines complexes doivent être modélisées en trois dimensions, ce qui produit un grand nombre d'inconnues ;
- la modélisation n'est pas linéaire, elle fait donc intervenir une boucle supplémentaire dans la résolution pour traiter cette non linéarité ;
- il y a évolution dans le temps de la géométrie de la machine simulée, ce qui implique un remaillage éventuel et surtout une reconstruction du système d'équation.

Toutes ces caractéristiques produisent des problèmes de grande taille nécessitant un temps de calcul important. La version séquentielle du programme n'a permis d'étudier que des exemples de taille relativement modeste. Une parallélisation a donc été envisagée pour des raisons à la fois de taille mémoire et de temps de calcul.

3.2.2 Approche parallèle « légère »

Objectif et travail préliminaire

Dans une première étape, détaillée dans l'annexe D, nous avons voulu construire une version parallèle de l'application impliquant le moins de modifications possibles du code d'origine. Nous avons mis la priorité sur la maintenabilité du code par les électromagnéticiens, la rapidité de parallélisation et la simplicité de la méthode. Il fallait en particulier ne changer ni les structures de données utilisées, ni les algorithmes, ni la structure générale du code. Nous ne nous sommes autorisé que l'ajout de directives High Performance Fortran et de quelques lignes de code.

Une réécriture quasi-complète de l'application a été nécessaire afin de la rendre compatible avec nos objectifs de portabilité. Outre une plus grande structuration et efficacité (en terme d'utilisation de la mémoire et de temps de calcul), la version Fortran 95 a ajouté la possibilité d'utiliser les trois éléments de discrétisation (et non seulement le tétraèdre) et le calcul des arêtes et facettes. Nous avons aussi optimisé un certain nombre d'algorithmes. L'utilisation de tableaux de taille dynamique a aussi permis la résolution de problèmes de toutes tailles.

Cette réécriture peut paraître contradictoire avec l'objectif de modifications mineures avancé juste avant... Nous ne la considérons pas comme de la parallélisation mais « simplement » comme un nettoyage du code le rendant maintenable et modifiable. Cette version a d'ailleurs été très vite utilisée par les électromagnéticiens du L2EP.

Parallélisation

Une analyse du temps pris par les différentes étapes du calcul nous ont conduit à paralléliser les plus coûteuses en temps de calcul : l'assemblage et la résolution du système d'équations.

Vu la structure de données représentant la matrice creuse, il a été décidé de la répliquer sur tous les processeurs pour éviter des coûts de communication prohibitifs.

Il a suffit de quelques directives de placement de données, d'identification de boucles parallèles et de l'ajout de quelques instructions utilisant la bibliothèque Halos pour gérer les communications irrégulières pour paralléliser le code. Moins de 1 % du code Fortran 95 a été modifié.

Résultats et analyse

Des mesures sur deux machines parallèles d'architectures différentes (IBM SP2 et Origin 2000 de Silicon Graphics) nous ont donné des résultats correspondant à nos attentes, à savoir une accélération acceptable pour un petit nombre de processeurs et une dégradation nette de cette accélération avec l'augmentation de ce nombre.

La raison principale de ce manque de scalabilité est la réplication des données due à l'impossibilité de découper de façon satisfaisante la matrice du système. Pour aller plus loin il faut changer l'algorithme de résolution pour se rapprocher d'une résolution par blocs qui devrait permettre un partitionnement plus aisé des données et des calculs. C'est ce que nous avons appelé l'approche parallèle « lourde ».

3.2.3 Approche parallèle « lourde »

Nous avons ici [17] changé les priorités dans notre recherche d'un compromis maintenabilité/performances en permettant des changements plus conséquents du code séquentiel en vue d'obtenir des temps de calcul réduits.

Pour obtenir un degré de parallélisme plus important que précédemment, nous nous sommes intéressés aux méthodes de décomposition de domaines. Compte tenu des caractéristiques mathématiques de l'application toutes les méthodes additives de Schwarz sont utilisables. Après un choix par élimination nous avons implémenté la méthode du complément de Schur, une méthode de sous-domaines sans recouvrement. Les raisons de ce choix sont principalement liées à des considérations numériques, aux caractéristiques des maillages manipulés et aux algorithmes de discrétisation employés dans l'application.

Partitionnement des données

Le parallélisme recherché est maintenant au niveau des domaines. La décomposition en domaines doit être faite au moment de la construction du maillage. La méthode du complément de Schur nécessite des interfaces régulières entre les domaines. Il est donc impossible d'utiliser des outils de partitionnement automatique. L'utilisateur doit donc réaliser cette phase fastidieuse manuellement. C'est le plus gros inconvénient de cette méthode.

Les conséquences du partitionnement manuel sont un petit nombre de sous-domaines assez bien équilibrés et séparés par des plans. Il est alors possible d'affecter chaque sous-domaine à un processeur et de traiter toutes les frontières sur un dernier processeur. Si le nombre de sous-domaines est plus important que le nombre de processeurs, il est toujours possible de placer plusieurs sous-domaines sur un même processeur. Nous voyons donc que l'équilibrage de charge reste la responsabilité de l'utilisateur.

Au niveau de l'implémentation, un système de copies fantômes de nœuds frontières a été mis en place pour gérer les communications. En effet, les calculs se font comme si les données nécessaires à chaque sous-domaine (y compris la frontière) sont présentes localement. Il n'y a

plus qu'à synchroniser au bon moment les copies fantômes avec les processeurs distants qui en sont responsables. Cela permet une structuration du code où la gestion des communications est découplée des phases de calcul (pré-chargement et post-stockage des données fantômes).

Le gros du travail de parallélisation a en fait été réalisé lors de l'écriture d'une version séquentielle de l'application basée sur la méthode de Schur. En effet, cette version a été pensée pour être facilement parallélisable. Il a suffit de l'ajout de 120 lignes de code sur 11 000 pour obtenir une version parallèle à l'aide d'High Performance Fortran et Halos.

Résultats et analyse

Les résultats obtenus confirment nos prévisions. Tout le code ayant été parallélisé et les données réparties sur les divers processeurs, nous avons obtenu une bien meilleure scalabilité.

L'effort de génie logiciel a été soutenu dans la réalisation de cette version du code. En effet, la gestion des communications a été cloisonnée dans un module et est complètement transparente dans le reste de l'application grâce au mécanisme de pré-chargement et de post-stockage. L'effort de prise en main par les électromagnéticiens est cependant plus important car les algorithmes fondamentaux de l'application ont dû être changés. Cependant au vu de la validité des résultats et du gain de performances, ils ont apprécié cette parallélisation. Des extensions (calcul d'erreurs) sont en cours de réalisation.

Le gros point noir de cette version du code est l'équilibrage de code manuel lors du maillage. L'étude du partitionnement automatique dépasse nos objectifs (et nos compétences). Une possibilité pour résoudre ce problème serait de construire de nombreux sous-domaines et de les regrouper sur les processeurs de la machine cible. L'équilibrage pourrait alors se faire par migration de sous-domaines d'un processeur à l'autre. Le manque de temps ne nous a pas permis de pousser plus avant cette voie.

3.2.4 Bilan

L'originalité de cette étude est la recherche d'un compromis entre maintenabilité et performances. Deux approches de la parallélisation ont été présentées privilégiant l'un ou l'autre de ces deux objectifs.

La base commune a été l'utilisation du paradigme du parallélisme de données pour permettre une parallélisation à haut niveau de l'application. L'utilisation d'une bibliothèque optimisant les communications irrégulières nous a permis d'obtenir simplement de bonnes performances malgré le stockage creux des matrices manipulées.

Les bonnes performances obtenues et la prise en main de l'application parallélisée par les électromagnéticiens ont validé notre approche.

3.3 Vers la simulation distribuée

La collaboration avec le L2EP se prolonge autour d'un autre sujet d'étude. En effet, le L2EP travaille avec différents partenaires universitaires et industriels dans le cadre du CNRT Futurelec[2] sur le thème des **Réseaux et Machines Électriques du Futur**.

Dans le cadre des accords entre Tractebel et le CNRT, nous collaborons avec l'équipe de Francis Piriou et Tractebel Engineering sur le thème de la simulation distribuée de réseau

[2] Voir `http://www.recherche.gouv.fr/technologie/cnrt/liste.htm`.

électrique. Lors du stage de DEA de Ouahid SAMET, nous avons étudié la parallélisation d'une application pédagogique de simulation de réseau électrique de Tractebel, FAST [44]. Cette application consiste à l'inversion d'une matrice représentant la discrétisation de grandeurs électriques sur un réseau électrique (générateurs compris). La structure particulière de la matrice permet une résolution par blocs (liés à la structure physique du réseau). Ce travail préliminaire nous laisse espérer une parallélisation assez efficace avec, comme dans l'étude de magnétodynamique, relativement peu de modifications du code séquentiel.

L'objectif à moyen terme de cette collaboration est d'étudier la parallélisation sur un cluster de machines SMP d'un simulateur plus complexe et complet que FAST, Eurostag [64]. Une évolution de ce logiciel est imposée par la dérégulation du marché de l'électricité qui provoque une modification de l'utilisation du réseau. Cette évolution sera très probablement coûteuse en temps de calcul, d'où la parallélisation d'Eurostag.

Notre objectif sera toujours de travailler au plus haut niveau possible et, si la structure de l'application le permet, de faire le lien avec les travaux sur la simulation distribuée présentés au chapitre suivant.

Chapitre 4

Réseaux de processus distribués

4.1 Motivation et domaine d'application

La motivation initiale des travaux de la thèse d'Abdelkader Amar est l'étude d'un support d'exécution distribué pour des applications de calcul scientifique [3]. Ce support doit cacher la complexité de construction d'applications distribuées au programmeur non spécialiste. En particulier, la gestion de bas niveau des communications doit être prise en charge par le système. Nous proposons de modéliser les applications comme un graphe de composants interconnectés.

Le domaine d'application comprend en particulier la simulation distribuée comme on l'entend dans le projet Sophocles (voir l'encart page suivante). Les applications sont ici des simulations de l'exécution d'applications complexes sur des architectures hétérogènes comme des systèmes sur silicium. Chaque composant de l'application est alors le simulateur d'un composant matériel de l'architecture et de l'exécution de la partie de l'application qui lui est attribuée. Le projet vise à la proposition d'une « cyber-entreprise » (voir l'encart page 31) permettant le choix des composants et le placement de l'application sur ces composants. La simulation est alors déclenchée et fonctionne de manière autonome.

Un concept clé que nous voulons garantir est la *dynamicité* qu'on peut voir sous plusieurs aspects :

- le développement incrémental de l'application ;
- la possibilité de remplacer à l'exécution un composant par un autre (pour corriger une erreur ou améliorer les performances) ;
- la migration d'un composant vers un autre nœud de calcul pour l'amélioration du matériel, l'équilibrage de charge ou encore la tolérance aux pannes.

Les applications visées peuvent en effet requérir de très longs temps de calcul, voire prendre en entrée un flux infini de données à traiter comme dans le cas des applications de traitement de signal intensif (voir la définition au paragraphe 5.1 page 35). Dans ce cas, la dynamicité à la fois du logiciel et du matériel prend tout son sens.

4.2 Modèle : les réseaux de processus

Pour représenter le parallélisme et la distribution de l'application, nous avons choisi d'utiliser le modèle des réseaux de processus proposé par Kahn [50, 51]. Ce modèle est souvent utilisé (en particulier par nos partenaires industriels dans le projet Sophocles) pour modéliser les applications de traitement de signal.

Projet ITEA *Sophocles*

System level develOpment Platform based on HeterOgeneous models and Concurrent LanguagEs for System applications implementation.
`http://www.sophocles-itea.org/`

But du projet. Le but du projet Sophocles est une validation conceptuelle d'une méthodologie, de plateformes et de technologies supportant le développement de systèmes complexes dans un environnement distribué. Ces systèmes sont construits à partir de composants virtuels (destinés à être assemblés sur un SoC) et leur développement comprend des phases de spécification, validation et intégration. Cette méthodologie devrait permettre la création de « cyber-entreprises » (voir l'encart page ci-contre) fournissant des services d'intégration via le web.
Les principaux utilisateurs de cette méthodologie seront les architectes de systèmes complexes, les fournisseurs de composants virtuels, les concepteurs de propriétés intellectuelles (IP) et les producteurs de systèmes finaux. Des interfaces cognitives proposeront un support intelligent aux activités de l'architecte système.
Ce projet est basé sur un certain nombre de nouveaux formalismes : SynchCharts Esterel, Array-OL, Evolving Grammars, Made, SIMPLE. Des technologies récentes sont utilisées : UML, XML, CORBA, JAVA, MPI, etc.

Partenaires. Le consortium est fondé sur la complémentarité.

France : THALES Communications (TCFR), THALES Underwater Systems (TUS), Esterel Technologies et LIFL
- pilotage (TCFR)
- techniques et environnement de simulation

Pays-Bas : Philips
- environnement pour l'analyse de performances

Italie : IPiTEC, ENEA
- cyber-entreprise
- techniques de simulation

Calendrier. Le projet a démarré début septembre 2001 et se terminera fin août 2003.

Cyber-entreprise

Lorsqu'une entreprise veut développer une application de traitement intensif sur un SoC, elle définit généralement l'architecture d'exécution en même temps que son application. Cette architecture est composée de différents composants matériels vendus par différents fournisseurs. Vu la complexité grandissante de ces applications et le coût des composants matériels, des simulations sont nécessaires avant de finaliser les choix.
Ces simulations se font à divers niveaux, fonctionnel d'abord, puis de plus en plus précis jusqu'à être valides au bit et au cycle d'horloge près. Ce n'est que lorsque toutes les simulations donnent un résultat satisfaisant que la décision de fondre le circuit peut être prise.
Cette approche pose des problèmes complexes de propriété industrielle : le fournisseur de composant ne doit pas avoir accès au code de l'application simulée et réciproquement, le développeur d'application ne doit pas avoir accès au code du simulateur. Le développement d'une *cyber-entreprise* permettant le couplage via l'internet des différents simulateurs (restant chez le fournisseur de composants) peut permettre un tel modèle de développement. Se posent alors les questions de la définition de l'interface des *composants virtuels* (VC) et de leur interaction.
Les constructeurs se sont regroupés au sein de la *Virtual Socket Interface Alliance* (`http://www.vsi.org/`) afin de définir des standards facilitant la réutilisation de VC pour la conception de systèmes sur silicium.

Dans les réseaux de processus les communications se font exclusivement à travers des files d'attentes unidirectionnelles et non bornées. Les deux seules primitives de communication à la disposition des processus sont l'écriture et la lecture (bloquante quand la file est vide) dans une file d'attente. Un processus peut ainsi être vu comme une application de ses flux d'entrée vers ses flux de sortie.

La valeur et le nombre de jetons produits ne dépendent que de la définition du réseau et sont indépendants de l'ordonnancement choisi des processus. Le choix de cet ordonnancement ne détermine que la taille des files d'attentes et si le calcul se termine. Certains réseaux ne permettent pas une exécution bornée. Parks [70] a étudié ces problèmes d'ordonnancement et a proposé un ordonnancement combinant une politique à la demande et à la disponibilité pour permettre une exécution complète et non bornée d'un réseau de processus lorsque c'est possible.

Il existe plusieurs implémentations des réseaux de processus dans différents domaines : pour la modélisation hétérogène avec PtolemyII [56], pour la modélisation d'applications de traitement de signal et l'étude de leur performances avec Yapi [32] et pour le métacalcul dans le domaine des systèmes d'information géographique avec Jade/PAGIS [83, 84]. Seule l'implémentation de Jade/PAGIS est distribuée, celles de PtolemyII et de Yapi utilisent des fils d'exécution pour représenter les différents processus. De plus, aucune de ces implémentations n'est dynamique ni ne permet le couplage de codes écrits dans des langages différents.

4.3 Choix techniques et réalisations

Comme exposé dans l'annexe E nous avons implémenté une version distribuée et dynamique des réseaux de processus. Pour réaliser nos objectifs d'interopérabilité nous avons eu besoin d'utiliser un intergiciel. Nous avons choisi CORBA [66] qui a été développé dans cet objectif et pour lequel de nombreuses implémentations existent.

Le modèle que nous avons implémenté est légèrement restreint par rapport aux réseaux de processus décrits ci-dessus. Les légères différences (restrictions sur la forme des processus et files d'attentes bornées) présentées en détail en annexe le rendent plus adapté à un gros grain de concurrence nécessaire pour une utilisation efficace d'un réseau à grande échelle et plus compatible avec notre objectif de couplage dynamique de codes.

Notre implémentation repose sur le concept de demies files d'attentes. Chaque file est décomposée en une partie attachée au processus qui écrit et en une deuxième partie attachée à celui qui lit. Ces deux parties s'échangent des éléments par négociation. Hormis la phase de déploiement des différents composants de l'application, toutes les communications se font exclusivement par ces files d'attente et ne nécessitent aucun point de contrôle centralisé. L'utilisateur écrit ses composants et une interface IDL de ceux-ci. Un générateur automatique exploite alors ces données pour construire l'application distribuée. Le déploiement de l'application est pour le moment manuel via une console interactive qui permet de lier les demies files d'attentes. Cette console permet aussi la gestion de la dynamicité par le remplacement d'un composant par un autre. Les détails de ces mécanismes sont présentés en annexe.

4.4 Travaux en cours

Basés sur le prototype présenté au paragraphe précédent nous en avons entamé des extensions.

4.4.1 Interfaçage avec d'autres implémentations

En collaboration avec Philips, qui a développé Yapi [32], une implémentation du modèle des réseaux de processus monoprocesseur, nous étudions la distribution d'un réseau à grain fin. Dans ce cas, il est inefficace d'utiliser un intergiciel comme CORBA pour encapsuler chaque processus du réseau. Le but est donc de proposer une distribution d'un réseau en regroupant dans un même composant les processus s'exécutant sur la même machine. En gros, cela revient à utiliser Yapi à l'intérieur de composants CORBA comme ceux du paragraphe précédent.

Une première évaluation de la faisabilité de cette approche est en cours sur une application proposée par Philips. Basés sur les problèmes rencontrés lors de cette expérience, dus principalement à l'ordonnanceur de Yapi, nous allons adapter notre prototype en vue d'obtenir un « Yapi distribuable » où il suffira d'indiquer le placement des processus pour générer l'application distribuée.

D'un autre côté, une implémentation distribuée du modèle de réseaux de processus de PtolemyII a été réalisée par THALES Communications au sein de Sophocles. Nous allons interfacer leur implémentation et la nôtre grâce à CORBA. Cela permettra de profiter des atouts des deux implémentations, à savoir leur intégration dans la plateforme de spécification hétérogène PtolemyII, et la dynamicité et le protocole de communication de notre système.

4.4.2 Retour d'information

Un autre des objectifs de nos travaux est d'obtenir des informations sur l'exécution distribuée. En effet, un certain nombre de paramètres sont observables, tels que la charge des machines, la longueur des files d'attente et les débits de ces files d'attentes.

Nous ajoutons à notre prototype la collecte de ces informations et la construction d'indicateurs caractérisant les déséquilibres dans l'exécution. On peut envisager par la suite un

déclenchement automatique des migrations de composants pour équilibrer dynamiquement la charge de calcul.

4.4.3 Parallélisme de données

Afin de rapprocher notre modèle d'exécution du traitement de signal intensif, nous allons en proposer une extension dataparallèle. Il s'agira de permettre le parallélisme de données à l'intérieur des composants distribués sur des machines multiprocesseurs. Ce modèle se rapproche d'autres techniques comme celles proposées pour un CORBA pour le calcul intensif [52, 71].

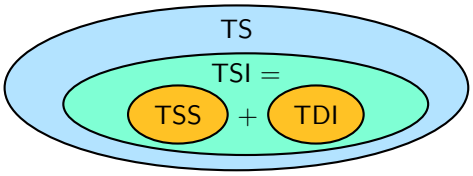
Chapitre 5

Environnement de développement pour le traitement de signal intensif

Depuis 1998 mes travaux s'orientent vers un cadre applicatif particulier : le traitement de signal intensif. Les propriétés de régularité de ce domaine nous laissent espérer de grandes possibilités d'optimisation.

J'ai pour l'instant mené des études dans deux directions complémentaires. D'une part la spécification visuelle d'applications de traitement de signal systématique, qui a donné lieu à la réalisation d'un prototype logiciel. D'autre part, j'encadre à 50 % la thèse de Philippe Dumont sur les techniques de compilation (placement spatio-temporel par transformations de code) pour la génération de code de traitement de signal intensif sur systèmes sur silicium.

5.1 Contexte : le traitement de signal intensif



La partie du traitement de signal (TS) qui nous intéresse est sa partie la plus intensive, composée du traitement de signal systématique (TSS) et du traitement de données intensif (TDI) plus irrégulier. Le TSS correspond à la première phase de traitement des signaux et consiste principalement à l'application de filtres et à des traitements très réguliers (indépendants de la valeur des signaux) appliqués systématiquement aux signaux d'entrée pour en extraire les caractéristiques intéressantes. Celles-ci sont ensuite traitées par des calculs plus irréguliers (dépendants de la valeur de ces grandeurs) dans la phase de TDI.

Ce schéma en deux phases se retrouve dans beaucoup d'applications de traitement de signal ou de l'image. En voici quelques exemples représentatifs venant de nos partenaires industriels.

Récepteur de radio numérique. Cette application en émergence fait appel à une partie frontale de TSS consistant à la numérisation de la bande de réception, la sélection du canal et l'application de filtres permettant d'éviter les parasites. Les données fournies par ces traitements systématiques sont ensuite envoyées dans le décodeur dont le traitement est plus irrégulier (synchronisation, démodulation, etc).

Traitement sonar. Une chaîne de traitement sonar classique se compose d'une première étape systématique : la veille bande large suivie d'un traitement de données : la poursuite. La première phase prend en entrée les signaux produits par les hydrophones (microphones répartis autour du sous-marin) et, par une suite de traitements systématiques produit des voies, couples (direction, intensité), représentant les échos captés. Ces échos sont ensuite analysés par la poursuite pour identifier et suivre au cours du temps les objets les produisant.

Encodeur/décodeur JPEG-2000 JPEG-2000 est un nouveau format standard de compression d'images. Le fonctionnement de l'encodeur [1] suit le même schéma en deux phases. La première partie (du prétraitement à la décomposition en ondelettes) est systématique. C'est dans la deuxième partie de l'encodage qu'apparaissent des traitements irréguliers (quantification, deux étages d'encodage). Le décodeur fonctionne exactement à l'inverse de l'encodeur et fait donc se suivre une phase de TDI et une phase de TSS.

5.2 Vue d'ensemble : de l'environnement de programmation au support d'exécution

Nos travaux sur le traitement de signal intensif ont démarré lors d'une collaboration avec Thomson Marconi Sonar (maintenant THALES Underwater Systems) autour de la compilation du langage Array-OL.

Array-OL [8] est un langage dédié au TSS. Il est basé sur la constatation que la complexité de telles applications vient des accès aux données (toujours des tableaux) et non des fonctions de calcul. On exprime donc visuellement le contrôle de l'application et textuellement les fonctions élémentaires de calcul. Array-OL est un langage à assignation unique où seules les dépendances de flot de données sont exprimées et où les dimensions spatiales et temporelles des tableaux sont banalisées.

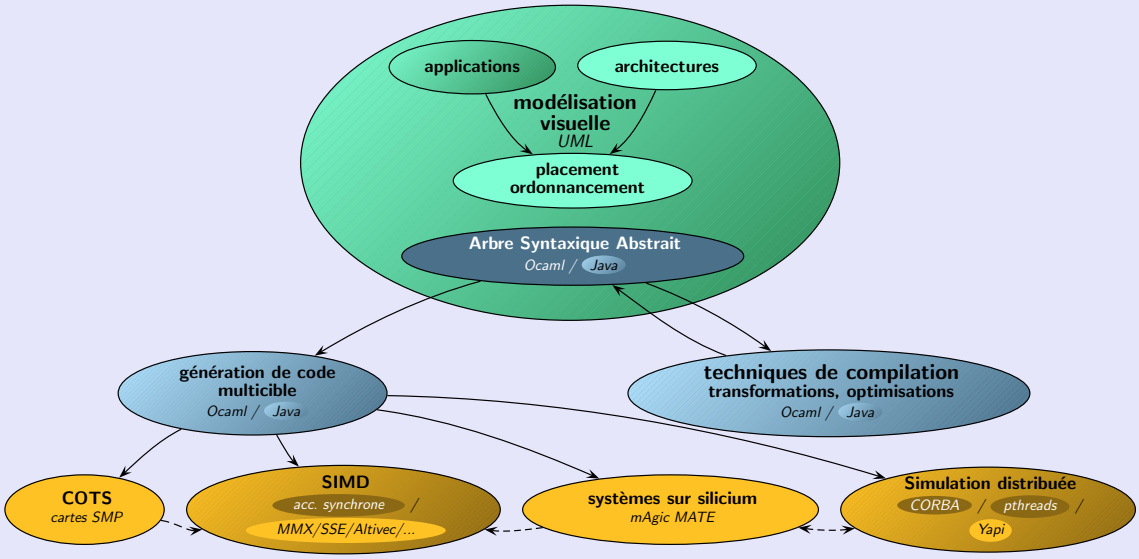
Autour de ce langage, nous avons développé une plateforme de prototypage, Gaspard (voir l'encart page suivante). Nous y avons intégré des outils de spécification visuelle, un compilateur, un outil de transformation de code qui permet diverses optimisations et un générateur de code multicible. Je présenterai dans ce chapitre la première version de Gaspard centrée autour de la spécification d'algorithmes en Array-OL pour aller jusqu'à la compilation sur divers supports matériels (il s'agit de la version réalisée en Objective Caml). Une nouvelle version est en cours de développement en Java. Elle est plus ambitieuse et servira de plateforme de prototypage pour tous les futurs travaux présentés au chapitre 6.

5.3 Outils visuels

Array-OL se prête bien à une spécification visuelle du contrôle d'une application de TSS. Il est constitué en deux niveaux : le modèle global et le modèle local. Le *modèle global* est un graphe de tâches tout à fait classique. Les seules données manipulées sont des tableaux et ces tableaux sont produits et consommés par les tâches. On exprime dans ce modèle global des dépendances entre tableaux. L'originalité tient ici au fait que le graphe de tâches est acyclique et qu'on travaille en assignation unique. Les tableaux ne sont produits que par une seule tâche. Pour pouvoir représenter les flux de données habituels en TSS, les tableaux peuvent avoir des dimensions infinies qui représentent alors le temps. Le *modèle local* représente la façon

Gaspard

Gaspard est notre plateforme de prototypage. Il est structuré autour d'un arbre syntaxique abstrait représentant une application placée sur une architecture. Cet arbre est obtenu par modélisation visuelle (suivant le schéma en « Y », voir le paragraphe 6.2.2 page 44). La compilation se fait par des transformations de cet arbre puis génération de code multicible.



Dans ce schéma, les couleurs foncées représentent les réalisations implémentées, les couleurs claires les projets et les dégradés les travaux en cours.

dont les tâches élémentaires utilisent les éléments des tableaux. On y spécifie des itérateurs dataparallèles permettant une exécution de type SPMD des tâches élémentaires. Ces itérateurs sont en fait les tâches du modèle global. Les dépendances exprimées à ce niveau local sont des dépendances entre éléments de tableaux. La description des itérateurs se fait par la construction d'un pavage par des motifs identiques de chaque tableau lu ou produit par la tâche. L'ajustage permet de préciser comment construire un motif à partir de son origine et le pavage décrit les origines des motifs. Ces deux opérations sont caractérisées par des matrices, M_a pour l'ajustage et M_p pour le pavage. En notant q le vecteur parcourant l'espace des motifs, d celui qui parcours les points du motif, m celui des tailles des dimensions du tableau et o les coordonnées du motif d'indice 0, l'équation qui donne les coordonnées, dans le tableau, du point correspondant aux coordonnées d dans le motif q est $\left(M_p.q + M_a.d + o\right) \bmod m$. Le modulo permet de représenter des dimensions physiques toriques, tels des hydrophones autour d'un sous-marin. Il n'est pas nécessaire dans tous les cas mais nous fait sortir du modèle polyédrique.

La taille de l'espace d'ajustage est donnée par la taille du motif, et la taille de l'espace d'itération (unique pour l'itérateur et donc commun à tous les pavages) est déterminée par le remplissage des tableaux de sorties. Un exemple de spécification pour le produit de matrice construit par application parallèle de produits de vecteurs est donné sur la figure 5.1 pour l'ajustage et la figure 5.2 pour le pavage.

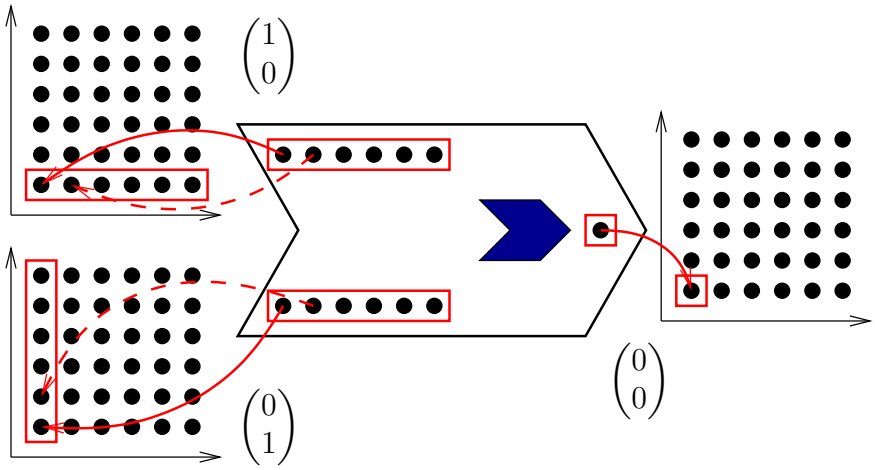


Fig. 5.1 – Matrices d'ajustage

Ces deux niveaux, global et local, peuvent être hiérarchisés. Une tâche élémentaire d'un niveau local peut être décrite comme un modèle global. Les tâches élémentaires du plus bas niveau sont implémentées dans le langage cible du compilateur. Une bibliothèque pour le TSS comprendrait peu de ces tâches : des produits scalaires, des transformées de Fourier et des sommes.

Nous avons prototypé un environnement visuel de spécification pour le langage Array-OL. Ce développement a été réalisé en Objective Caml pour le compilateur et TCL/Tk pour l'interface graphique. Ce prototype est présenté dans l'annexe F. Il a ensuite évolué avec une interface plus performante en GTK+, avec en particulier un modèle local graphique et une métaphore visuelle de circuit imprimé, pour donner le prototype décrit dans l'article [7]. La cible de compilation est dans cette version un système de métacalcul et l'interface visuelle se

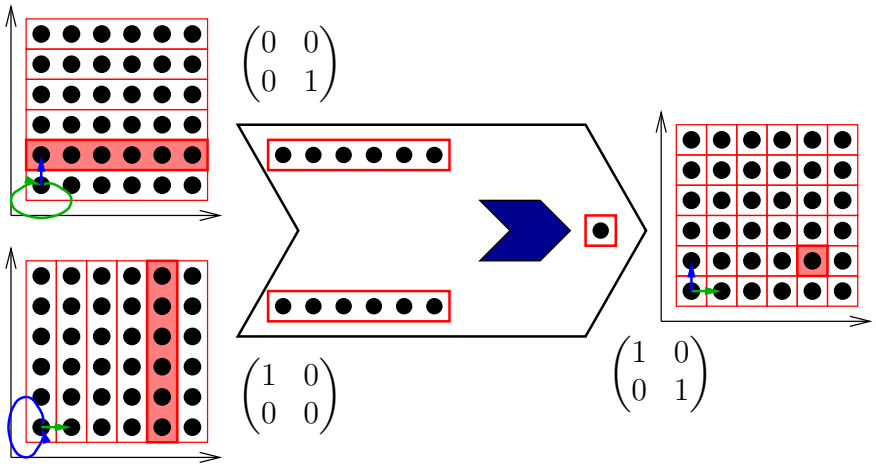


FIG. 5.2 – Matrices de pavage

veut à la fois une interface de spécification et une interface de surveillance de l'exécution. Ce travail est encore en cours.

La nouvelle version en cours de développement en Java aura sa partie visuelle prise en charge par un outil UML (voir le paragraphe 6.2.1). Nous allons ainsi vers un niveau plus élevé que la spécification d'algorithme en utilisant un langage de modélisation. Cette interface UML servira aussi à spécifier l'architecture d'exécution et le placement de l'algorithme sur cette architecture.

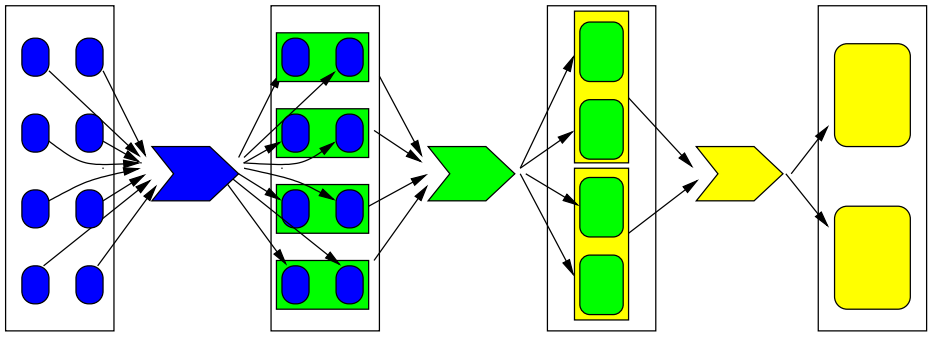
5.4 Techniques de compilation

Lors de la thèse de Julien SOULA [8, 79], nous avons mis au point des transformations de code Array-OL vers Array-OL. De telles transformations sont nécessaires pour exprimer le placement d'une application Array-OL. Elles permettent en effet de choisir la granularité de l'application, de factoriser le graphe de composants ou de faire un compromis entre occupation mémoire et temps de calcul.

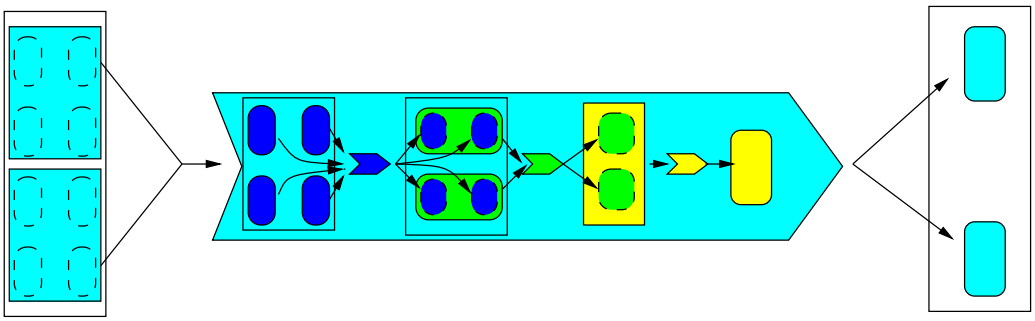
Un formalisme adhoc, les *opérateurs de description de tableaux*, a été développé pour représenter les itérateurs Array-OL. Ce formalisme met en relation les éléments des tableaux résultats avec ceux des tableaux opérandes (pour une description complète voir [79]). Des transformations de code élémentaires reposant sur ce formalisme ont été formalisées et implémentées dans Gaspard. L'intérêt principal d'utiliser un formalisme si spécialisé est que les programmes obtenus par transformation restent des programmes Array-OL. Ils sont donc ainsi compilables par le même compilateur et candidats à de nouvelles transformations.

Schématiquement, la fusion permet le regroupement de plusieurs tâches consécutives dans une même hiérarchie. Un itérateur dataparallèle entoure alors la séquence de tâches et permet l'expression d'un plus gros grain de parallélisme. De plus, si l'exécution de ces tâches est en partie séquentialisée, on peut réutiliser l'espace mémoire utilisé pour le stockage des tableaux intermédiaires. Ceci est crucial dans le cas des tableaux infinis. Une autre transformation, le changement de pavage, joue directement sur la granularité du calcul. Un exemple de fusion

de trois tâches successives est présenté sur la figure 5.3. En supposant qu'on utilise un mode d'exécution où chaque tableau doit être produit en entier avant d'être consommé, la version hiérarchisée par fusion nécessite deux fois moins de mémoire pour stocker les tableaux intermédiaires. De plus, si tous les tableaux étaient infinis dans leur dimension verticale (cas classique d'un flux de données), la version non hiérarchique devrait allouer une quantité de mémoire infinie alors que dans l'autre cas, on peut exécuter de façon pipelinée le niveau hiérarchique produit. Il faut noter ici qu'on ne fait qu'exprimer autrement les mêmes dépendances. L'inté-



3 tâches dataparallèles successives.



Après fusion des 3 tâches, 2 niveaux de parallélisme de données.

FIG. 5.3 – Fusion de trois tâches

rêt des transformations est de mettre l'application dans une forme facilement exploitable par la phase de placement et d'ordonnancement, et d'orienter ceux-ci.

Ces travaux sont le point de départ de futures recherches. Il faut en effet maintenant définir une métrique pour pouvoir quantifier les effets d'une telle transformation de code (sur l'espace mémoire, le temps d'exécution, la facilité de placement, etc.) et des stratégies pour piloter leur utilisation. Ces travaux ont commencé dans le cadre restreint du placement d'applications Array-OL sur systèmes sur silicium. À plus long terme, nous allons étudier la généralisation de ces transformations pour placer et optimiser les applications de TS définies dans notre modèle.

5.5 Support d'exécution distribué

Le prototype Gaspard supporte la compilation pour plusieurs architectures : C++ séquentiel, C++ multithreadé (bibliothèque pthread) et l'accélérateur synchrone (une puce SIMD dédiée au traitement de signal systématique dessinée par THALES Underwater Systems).

Une extension du générateur de code C++ est en cours de réalisation pour produire du code pour le support d'exécution distribué présenté au chapitre 4. À ce propos, le modèle des réseaux de processus peut servir de modèle d'exécution pour Array-OL en alimentant les tâches élémentaires par des files d'attentes de motifs. Il est alors possible dans certains cas de pipeliner l'exécution de ces tâches (il faut que les tableaux soient lus et produits « dans le même sens »). La formalisation des cas favorables et une proposition de méthode systématique de traduction du modèle de spécification Array-OL vers le modèle d'exécution réseaux de processus est encore à l'étude.

Chapitre 6

Bilan et perspectives

6.1 Bilan

Dans les chapitres précédents j'ai exposé divers travaux concernant la programmation d'applications de calcul intensif. Une des lignes directrices de mes recherches a été la quête d'outils de plus en plus haut niveau pour faciliter à la fois la tâche du programmeur et, paradoxalement, celle du compilateur. En effet, plus le programmeur exprime ses algorithmes simplement, plus le compilateur a de facilité à extraire les informations de dépendances de données qui sont la clé d'une optimisation efficace sur les architectures modernes.

Le parallélisme de données est au centre de toutes ces recherches, depuis la parallélisation automatique, la programmation en High Performance Fortran, jusqu'aux environnements dédiés au traitement de signal intensif. Il permet l'expression simple du parallélisme dans les applications manipulant de grandes quantités de données. Cette expression simple est ensuite utilisable aisément par le compilateur pour produire du code efficace.

J'ai montré des optimisations à tous les niveaux de la chaîne de développement, depuis l'environnement de programmation jusqu'au système d'exécution en passant par les différentes phases de compilation. Le but de toutes ces optimisations est de faire remonter l'information au plus près de l'utilisateur en lui masquant leur complexité.

Les perspectives que je présente ci-après sont intégrées à l'action DaRT[1] de l'unité de recherche Futurs de l'INRIA portée par l'équipe WEST du LIFL à laquelle j'appartiens. Elles font directement suite aux travaux présentés aux chapitres 4 et 5.

6.2 Vers un niveau encore plus haut : la modélisation

Nos contacts industriels nous ont fait nous intéresser à la conception d'applications de traitement de signal intensif et par conséquent au développement conjoint logiciel/matériel de telles applications, en particulier sur des systèmes sur silicium. Dans ce cadre, nous avons progressivement étendu nos activités de spécification d'applications à la spécification du déploiement d'une application sur une architecture.

Travaillant près d'industriels, nous avons cherché à nous rapprocher des standards de spécification. Nous recherchions une notation visuelle standard, bien acceptée dans l'industrie et autour de laquelle de nombreux outils existeraient. Notre attention s'est portée sur la

[1] Voir `http://www.lifl.fr/west/dart/`.

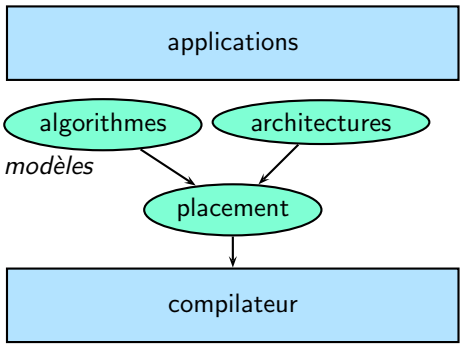
notation visuelle UML [67]. Nous sommes en cours de développement d'une spécialisation d'UML pour le traitement de signal intensif : ISP UML (*Intensive Signal Processing UML*). J'en précise ci-dessous les objectifs, le modèle fondamental et les principes de base. Ce modèle englobera le modèle RTE UML (*Real-Time Embedded UML*) qui sera développé au sein du projet Prompt*2Implementation* dont nous faisons partie (voir l'encart page ci-contre).

6.2.1 Objectifs d'ISP UML

Dans notre spécialisation d'UML en ISP UML, nous voulons satisfaire les objectifs suivants :

- *S'affranchir des langages de programmation,* en proposant une spécification visuelle au moins au niveau des expressions des dépendances si ce n'est au niveau du code des tâches élémentaires.
- *Réduire les temps de développement :*
  - en permettant la réutilisation totale ou partielle d'applications existantes, ou d'architectures existantes ;
  - en proposant des bibliothèques de composants prêts à l'emploi.
- *Respecter les standards* et donc l'adéquation avec divers outils qui les respectent.
- *Exprimer complètement le parallélisme potentiel* d'une application : parallélisme de tâches ou de données.
- *Spécifier à différents niveaux* une architecture, autant au niveau réseau d'échange d'une grappe qu'au niveau transfert de données sur une carte SMP ou dans un SoC.

6.2.2 Le modèle « Y »



Le modèle « Y » repose sur un environnement permettant une spécification visuelle des applications TSI, des architectures cibles et du déploiement des applications sur les architectures. Ce modèle de construction particulier permet de différencier la spécification, le support d'exécution et l'exécution proprement dite de l'application sur ce support. Séparer ces modèles correspond à une habitude de programmation dans ce domaine d'application. Le modèle « Y » est présent dans des environnements dédiés pour le *co-design* (conception conjointe logicielle/matérielle), pour la validation, pour la simulation et pour la génération de code. Il est à la base du projet ITEA Prompt*2Implementation*.

La séparation des modèles ouvre les voies de la réutilisabilité. Une même architecture ou application pourra apparaître dans plusieurs projets, en particulier on doit pouvoir réutiliser une application sur une nouvelle architecture, développer une nouvelle application ou transformer une application sur une architecture existante. Le placement reste bien sûr lié à l'application et à l'architecture. La séparation permet également un partage des tâches de développement par métier ; définir une application ne demande pas les mêmes compétences que celles nécessaires pour définir une architecture : assignation à la personne la plus compétente dans le domaine, pas d'interférence avec les autres domaines. La réutilisation ainsi que

Projet ITEA Prompt*2Implementation*

Parallel processing dedicated, Rapid Optimised Mapping Platform for Telecom applications to Implementation

But du projet. Dans le but de concevoir un système de développement conjoint logiciel/matériel pour des applications de télécommunications embarquées, le projet P2I propose de mettre en place le modèle « Y » en adaptant des outils de spécification d'applications, d'architectures, et de placements existants. Ayant pour objectif la définition d'une méthodologie non-ambiguë pour la spécification, la simulation, le test et l'implémentation de tels systèmes prouvés, le projet Prompt*2Implementation* reprend l'approche « Y » avec comme outils de spécification Esterel Studio [34] pour la spécification de l'application et les tests fonctionnels couplé à SynDEx [78] pour la spécification de l'architecture et du placement. Ces deux approches devraient être unifiées dans un environnement RTE UML (*Real-Time Embedded UML*). Partant de briques existantes, on peut espérer rapidement valider le modèle « Y » et son intégration dans le langage UML. Il doit alors être possible à partir de la spécification, de vérifier et de tester une application puis de générer du code efficace après placement par la méthode AAA [76], en particulier pour des applications de télécommunications. Notre action dans ce projet concerne effectivement la validation du modèle « Y » par prototypage via un profile/framework UML. Un effort particulier sera fait vers l'OMG (`http://www.omg.org/`) par le biais de propositions de standardisation. À la différence de l'objectif du projet DaRT, ces outils existants ne supportent pas explicitement le paradigme data-parallèle que nous tentons d'intégrer par expression des dépendances de données.

Partenaires. Ce projet est franco-finlandais. La participation des deux leaders mondiaux des télécommunications que sont THALES et Nokia permet aux autres partenaires de travailler sur des applications représentatives du domaine pour la réalisation des outils de développement.

France : Esterel Technologies, THALES Communications (TCFR), INRIA et LIFL
- pilotage (Esterel Technologies)
- proposition d'une méthodologie et d'un RTE UML (*Real-Time Embedded UML*)
- prototypage (évolutions et couplage d'Esterel Studio et SynDEx)
- validation sur une application de télécommunication

Finlande : Nokia, université de technologie de Tampere et université de Turku
- comparaison de plusieurs méthodologies de développement conjoint
- validation de la méthodologie sur une application de télécommunication

Calendrier. Le projet a été validé par l'ITEA. Le financement a été obtenu pour un démarrage en septembre 2002 pour deux ans.

la répartition vers les personnes les plus qualifiées contribuent à une diminution des temps de développement et permettent donc de réduire le temps de mise sur le marché.

Nous proposons un environnement visuel de spécification pour cette architecture « Y » construit autour d'une même métaphore. Il doit faciliter l'échange entre les différents métiers sans nécessiter la maîtrise de plusieurs langages. Il en va de même pour les environnements de spécification utilisés : pas d'apprentissage de plusieurs outils ou interfaces.

À chaque modèle on associe une méthodologie sachant que les interférences entre les trois modèles imposent par construction la définition d'une méthodologie collaborative et cohérente. Cette méthodologie transversale guide la réalisation complète d'une application TSI. Elle assure la cohérence entre les modèles, et en permet l'exploitation automatique par les outils de compilation, de génération de code, de transformation et de simulation.

Mon implication dans la définition d'ISP UML se place à la définition des objectifs de ce méta-modèle et du lien avec les techniques de compilation utilisées pour obtenir une exécution ou une simulation respectant les contraintes de temps (voir le paragraphe 6.3).

6.2.3 Points clés de la spécification d'applications en ISP UML

Notre objectif concerne la spécification d'algorithmes à un haut niveau d'abstraction. Nous avons déduit de l'observation des différents modèles de spécification utilisés par nos collaborateurs industriels qu'un système de traitement de signal intensif devait respecter les contraintes suivantes.

- *Assignation unique :* les données sont principalement des tableaux produits par une tâche élémentaire de traitement et consommés sans modification par d'autres tâches élémentaires. Cette assignation unique de tableaux facilite la spécification visuelle d'une application et permet de ne manipuler que de vraies dépendances de données (dépendances de flot).
- *Unification des dimensions temporelles et spatiales :* les dimensions des tableaux sont en général associées à des concepts propres à l'application (hydrophones, capteurs, énergie...). L'une d'entre-elles peut être associée au temps, elle permet alors l'identification des différentes valeurs de ces mêmes concepts durant la vie de l'application. Cette dimension devient alors de taille infinie pour une application embarquée.
- *Expression des dépendances temporelles et spatiales :* elle représente le seul lien qui unisse les différents objets manipulés par le programme. Elle fournit les dépendances entre les éléments des objets (tableaux) en entrée et en sortie de chaque tâche élémentaire. Elle recouvre autant les dimensions spatiales que temporelles. Elle guarantie l'expression du parallélisme potentiel maximal et permet la mise en œuvre directe des techniques de compilation.
- *Universalité d'un langage de programmation :* le domaine d'application doit couvrir le traitement de signal intensif. Au traitement systématique du signal est souvent associé un traitement de données intensif. Celui-ci est irrégulier, il manipule des structures de données dynamiques et les traitements sont à comportement variable. Afin de proposer un unique modèle pour ces deux types de spécification successifs, nous ajoutons deux caractéristiques essentielles:
  - *la récursivité,* elle permet en particulier de généraliser notre modèle afin de supporter des traitements de type réduction sur des tableaux de taille quelconque.
  - *les composants d'ordre supérieur,* cela augmente le domaine d'application de notre modèle en lui apportant l'universalité des langages fonctionnels.

AS Compilation pour les systèmes embarqués

Cette action spécifique du CNRS fait partie du réseau thématique pluridisciplinaire (RTP) architectures des machines et compilation (`http://www.archi-compil.org/`).
Son but est de fédérer les activités de recherche avec les objectifs suivants :
- optimisation dynamique des programmes et environnements de programmation ;
- optimisation des accès mémoire des programmes : réduction des espaces mémoire nécessaires aux applications, utilisation des caches pour le temps réel ;
- réduction de la consommation électrique : minimisation des instructions coûteuses, optimisation des communications réseau ;
- nouvelles architectures favorisant ces optimisations : circuits dédiés de faible coût, unités matérielles contrôlées par logiciel, interaction processeur/programme pour les optimisations dynamiques ;
- distribution, délégation, et utilisation distante de services à travers un réseau sans-fil afin de réduire la consommation des ressources locales.

Tous ces objectifs ont pour but de développer des mécanismes de compilation pour les applications dédiées aux systèmes embarqués.

Ces deux caractéristiques donnent à ISP UML la puissance d'un langage de programmation universel car il inclus de fait le λ-calcul. Il est cependant fortement orienté vers le traitement de signal intensif et s'en servir pour programmer d'autres types d'applications peut s'avérer impraticable.

Les deux autres parties de la spécification que sont l'architecture et le déploiement sont bien moins avancées. La spécification d'architecture sera commune avec celle de RTE UML (du projet Prompt*2Implementation* page 45) et sera basée sur les modèles existants (Array-OL architecture, mAgiSim, Vcc, SynDEx...) en faisant apparaître les éléments actifs, participant au traitement des données, et passifs, de mémorisation.

En ce qui concerne le déploiement, il faudra enrichir la version minimaliste proposée par UML pour exprimer les liens entre les éléments de modélisation de l'application et ceux de l'architecture. Ce déploiement pourra être explicite, semi-automatique, voire automatique selon les techniques de placement et d'ordonnancement mises en œuvre dans l'outil (voir ci-dessous).

6.3 De la spécification à l'exécution

À partir d'une spécification de haut niveau d'une application, d'une architecture et du déploiement de la dite application sur cette architecture, il reste un gros travail de compilation et d'optimisation pour obtenir un code (embarqué ou de simulation) avec les meilleures performances possibles pour garantir des temps de réponse rapides. De nombreuses difficultés scientifiques et techniques sont à étudier dans ce domaine. Les travaux que nous y menons sont soutenus par l'action spécifique du CNRS « compilation pour les systèmes embarqués » (voir l'encart).

- *Compilation efficace d'un modèle de haut niveau :* il faut tirer parti de la représentation des dépendances de données pour générer le code le plus efficace possible. De nombreuses techniques d'optimisations sont connues. La difficulté réside dans l'intégration de ces optimisations au sein du même code.
- *Placement et ordonnancement :* c'est le point clé de l'utilisation efficace des multiples unités d'exécution de la cible matérielle. Cette cible étant hétérogène, il peut y avoir

de fortes contraintes sur le placement (manuel, semi-automatique, voire complètement automatique). Des transformations de code (factorisation du graphe de composants, changement de granularité, etc.) peuvent être nécessaires pour exprimer de tels placements. Le placement et l'expression des dépendances permettent alors de choisir un ordonnancement de l'application. Les points difficiles sont ici les choix interdépendants d'un placement et d'un ordonnancement assurant le temps réél.

- *Génération de code :* nous voulons à partir d'un modèle d'une application être capables de générer un code d'exécution pour diverses architectures telles que des systèmes sur silicium, des architectures à base de COTS ou même des systèmes de métacalcul. Mis à part les problèmes de placement et d'ordonnancement, l'hétérogénéité des cibles matérielles et leur nature répartie font de la génération de code un problème techniquement difficile.
- *Simulation distribuée :* dans ce cas, les contraintes de placement sont moins fortes mais le support peut être dynamique (évolution du matériel, du logiciel, tolérance aux pannes, etc.). Outre cette difficulté technique supplémentaire, se pose le problème du niveau de simulation. En effet, on peut vouloir simplement une simulation fonctionnelle de l'application pour valider l'algorithme mais on peut aussi bien vouloir simuler le fonctionnement de l'application sur un simulateur de la cible matérielle choisie. Se posent alors des problèmes très complexes de couplage des simulateurs de chaque composant matériel.

Après cette vue générale de la problématique, précisons maintenant les travaux envisagés à court et moyen termes.

6.3.1 Compilation et placement spatio-temporel

Une des idées directrices de notre modélisation est de donner suffisamment de liberté au programmeur pour qu'il puisse exprimer le parallélisme de son application. En particulier, il ne doit pas exprimer de fausses dépendances de données, dues à un ordre d'écriture séquentiel ou à la réutilisation de la mémoire. Ceci libère l'utilisateur des contraintes d'exécution de son application. Il y a deux conséquences sur le compilateur : d'une part il a toutes les informations nécessaires pour optimiser et tirer parti du parallélisme potentiel de l'application, mais de l'autre, il doit prendre les bonnes décisions et la qualité de ses optimisations est primordiale dans les performances obtenues. Nous pensons que, dans le cadre restreint des applications de traitement de signal intensif, les techniques d'optimisation sont suffisamment mûres pour permettre une compilation efficace.

Notre modèle ISP UML a les propriétés suivantes :

- assignation unique ;
- expression des seules dépendances de flot de données ;
- unification des dimensions temporelles et spatiales ;
- récursif ;
- composants d'ordre supérieur.

L'assignation unique, la récursivité et les composants d'ordre supérieur apparentent ce modèle aux langages fonctionnels purs, c'est-à-dire sans effet de bord. La compilation et la parallélisation de tels langages [47, 65, 69, 75] fait appel à des techniques telles que le typage statique, la manipulation des fonctions comme des citoyens de première classe du langage, la dérécursivation, l'évaluation partielle, etc. D'autre part, le fait d'exprimer uniquement les dépendances de flot permet l'utilisation de nombreuses techniques de parallélisation automatique [2, 6, 31, 59]

(transformation de boucles, tuilage, etc.). Des langages spécifiques au traitement de signal temps réél proposent d'autres types d'optimisations [74]. L'interaction entre ces différents types d'optimisations reste à étudier.

Le choix des structures de données utilisées (tableaux à assignation unique) pose le problème majeur de l'utilisation efficace de l'espace mémoire. En identifiant quelles dimensions représentent l'espace et laquelle représentent le temps, la réutilisation de l'espace de stockage devient possible. Des études allant dans ce sens existent [58, 62, 81].

Le placement et l'ordonnancement d'une application sont deux étapes intimement liées du développement de l'application. Il s'agit en effet de la même opération dans les espaces respectivement physique et temporel. Les meilleurs ordonnancements et placements choisis indépendamment peuvent conduire à des exécutions catastrophiques. Il faut donc tenir compte de l'un pour calculer l'autre ou, idéalement, trouver un critère d'optimisation permettant de les réaliser conjointement. La plupart du temps, on optimise l'un puis on utilise ce résultat pour calculer l'autre. Les deux choix d'optimiser d'abord le placement ou d'abord l'ordonnancement existent dans la littérature [28].

La stratégie choisie par la méthodologie AAA [45, 76, 77] (Adéquation Algorithme Architecture) consiste en l'optimisation simultanée du placement et de l'ordonnancement par des heuristiques gloutonnes. Dans le cadre du projet Prompt*2Implementation* (voir page 45) nous allons collaborer avec l'équipe de l'action INRIA Ostre qui a développé la méthodologie AAA et le logiciel SynDEx [78]. Le placement et l'ordonnancement sont alors calculés automatiquement par une heuristique dans le cadre de contraintes posées par l'utilisateur.

Cependant, comme le domaine d'application que nous étudions est restreint et que notre modèle de spécification unifie les dimensions temporelles et physiques des données, une optimisation conjointe plus précise du placement et de l'ordonnancement nous semble possible. Dans la suite de nos travaux, nous rechercherons donc une méthode unifiant le placement et l'ordonnancement au sein du même formalisme pour, à terme, automatiser plus efficacement le placement spatio-temporel d'une application sur une architecture donnée. La proposition de Thies, Vivien, Sheldon et Amarasinghe [80] dans le cas des ordonnancements affines et des vecteurs d'occupation nous encourage dans cette voie.

Nous partons des travaux sur les transformations de code Array-OL présentés au paragraphe 5.4 pour les étendre dans trois directions (thèse de Philippe Dumont) :

1. proposer des stratégies d'enchaînement des transformations élémentaires optimisant certains critères (minimisation de la taille mémoire, des recalculs induits, adaptation à l'architecture mémoire de la cible) ;

2. étendre leur champ d'application du traitement de signal systématique au traitement de données intensif ;

3. étudier plus précisément le lien entre les opérateurs de description de tableaux et le modèle polyédrique pour enrichir mutuellement les techniques proposées dans ces deux domaines.

Conjointement à ces extensions, une méthode adaptée à la partie irrégulière du modèle ISP UML comprenant l'enchaînement d'une phase d'optimisation « fonctionnelle » pour se ramener à une expression plus adaptée aux techniques de parallélisation et d'optimisation de langages du type Fortran sera évaluée.

6.3.2 Génération de code

La phase finale de la compilation d'une application est la génération d'un exécutable pour l'architecture matérielle sur laquelle elle a été placée. Cette génération de code est donc paramétrée par une description des langages supportés par les différents composants matériels.

Les architectures visées sont de plusieurs types partageant à des degrés divers des caractéristiques d'hétérogénéité et de distribution :

- les systèmes sur silicium sont très hétérogènes, mais vu l'absence de système d'exploitation, les transferts de données et les temps d'exécutions sont très finement contrôlables ;
- les architectures à base de cartes multiprocesseurs sont plus homogènes et disposent souvent de compilateurs pour des langages de plus haut niveau (C, C++) ;
- enfin, les systèmes de métacalcul proposent de nombreux services pour assurer l'interopérabilité et la communication entre les composants logiciels de l'application. Le problème majeur réside ici dans la synchronisation entre ces composants. Nous en parlons plus en détail dans la section suivante.

Les applications de TSI sont encore majoritairement écrites en assembleur pour obtenir un maximum de performance. Au vu de la complexité croissante des architectures d'exécution, le coût en temps de développement devient prohibitif. De plus l'hétérogénéité de l'architecture oblige à générer du code dans des langages différents pour chaque composant pour lesquels des assembleurs optimiseurs ou des compilateurs de langages de bas niveau existent généralement. L'objectif de cette phase de compilation est de générer du code pour les différents assembleurs ou compilateurs natifs en fonction du placement et de l'ordonnancement choisi. Rappelons ici que notre modèle n'exprime que l'enchaînement des tâches et le placement des données, pas les tâches élémentaires de calcul. Celles-ci continuent à être développées dans les langages natifs des architectures cibles. Par contre, l'expression du contrôle de l'application reste à la charge du générateur de code. Une bonne partie des difficultés sont reportées d'une part dans la phase d'ordonnancement et de placement et d'autre part dans l'écriture des noyaux de calcul que sont les composants élémentaires. L'expression du contrôle peut devenir complexe après transformation du code. Des méthodes comme celles que j'ai présentées au paragraphe 2.4.1 peuvent alors être utilisées. Cependant, elles sont inutiles dans la plupart des cas et une génération simple des itérateurs est souvent possible.

Dans une première version de Gaspard, nous avons pu générer du code pour plusieurs cibles (voir le paragraphe 5.5) : en C++ avec la bibliothèque pthreads pour une station de travail multithreadée, en C++ et CORBA pour un système de métacalcul et en macro-assembleur pour un processeur de traitement de signal SIMD, l'accélérateur synchrone. Ces premiers développements nous ont amené à une réflexion sur la modularisation et la paramétrisation du générateur de code.

Nos objectifs à court terme sont de finaliser un moteur de génération de code générique et paramétrable au sein de notre prototype Gaspard. Une autre étude en collaboration avec l'IPiTEC commence sur l'utilisation de grammaires évolutives pour générer du code distribué. Ce type de grammaires est déjà utilisé avec succès dans le cadre de la génération de code pour une famille de systèmes sur silicium par l'IPiTEC [15, 61, 68].

Selon les résultats de ces travaux, nous orienterons nos recherches afin de complètement automatiser cette phase de génération de code et de faciliter sa paramétrisation par le modèle de l'architecture cible.

6.3.3 Simulation distribuée

Les travaux sur le support d'exécution distribué se poursuivent autour des trois directions présentées au paragraphe 4.4, à savoir :

- l'interfaçage avec d'autres implémentations ;
- le retour d'information sur l'exécution ;
- l'intégration du parallélisme de données.

En parallèle, Mickaël Samyn démarre une thèse sous ma co-direction sur la simulation répartie d'applications de traitement de signal intensif sur systèmes sur silicium. Il s'agira de se rapprocher des standards, proposés par exemple par la Virtual Socket Interface Alliance (`http://www.vsi.org/`) et SystemC (`http://www.systemc.org/`).

La diversité des modèles (avec ou sans référence de temps) utilisés à différents niveaux de simulation pose des problèmes de couplage entre les simulateurs des composants virtuels intervenant dans le système. Nous nous intéresserons en particulier au couplage de simulateurs de plus bas niveau pour lesquels les problèmes de synchronisation deviennent particulièrement critiques. En effet, une simulation au bit et au cycle près d'une application sur un SoC suppose l'échange très fréquent de petites quantités de données. Des techniques d'anticipation et de regroupement des communications peuvent alors être envisagées pour réduire les délais de communication.

Bibliographie

[1] Mickael D. Adams. « The JPEG-2000 still image compression standard ». Rapport Technique N2412, ISO/IEC JTC 1/SC 29/WG 1, septembre 2001. `http://www.jpeg.org/wg1n2412.pdf`.

[2] Randy Allen et Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, octobre 2001. `http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-286-0`.

[3] Abdelkader Amar, Pierre Boulet, et Jean-Luc Dekeyser. « Assembling dynamic components for metacomputing using CORBA ». Dans *Parallel Computing 2001*, Naples, Italy, septembre 2001. Lecture Notes in Computer Science.

[4] Mark Baker. « Cluster computing white paper ». Rapport Technique, IEEE Task Force on Cluster Computing, décembre 2000. `http://www.dcs.port.ac.uk/~mab/tfcc/WhitePaper/`.

[5] Siegfried Benker. « Optimizing irregular HPF applications using Halos ». Dans *Parallel and Distributed Processing, Proceedings of IPPS/SPDP Workshops*, volume 1586 de *Lecture Notes in Computer Science*, pages 1015–1024, San Juan, Puerto Rico, USA, avril 1999. Springer.

[6] Pierre Boulet, Alain Darte, Georges-André Silber, et Frédéric Vivien. « Loop parallelization algorithms: From parallelism extraction to code generation ». *Parallel Computing*, 24(3-4):421–444, mai 1998.

[7] Pierre Boulet, Jean-Luc Dekeyser, Florent Devin, et Philippe Marquet. « A visual development environment for meta-computing applications ». Dans *HCI International 2001, 9th Int'l Conf. on Human-Computer Interaction*, volume 1, pages 983–987, New Orleans, LA, août 2001. Lawrence Erlbaum Associates, Publishers.

[8] Pierre Boulet, Jean-Luc Dekeyser, Jean-Luc Levaire, Philippe Marquet, Julien Soula, et Alain Demeure. « Visual data-parallel programming for signal processing applications ». Dans *9th Euromicro Workshop on Parallel and Distributed Processing, PDP 2001*, pages 105–112, Mantova, Italy, février 2001.

[9] Pierre Boulet et Michèle Dion. « Code generation in Bouclettes ». Dans *Proceedings of the Fifth Euromicro Workshop on Parallel and Distributed Processing*, pages 273–280, London, UK, janvier 1997. IEEE Computer Society Press.

[10] Pierre Boulet, Jack J. Dongarra, Yves Robert, et Frédéric Vivien. « Static tiling for heterogeneous computing platforms ». *Parallel Computing*, 25(5):547–568, may 1999.

[11] Pierre Boulet et Paul Feautrier. « Scanning polyhedra without DO-loops ». Dans *PACT'98*, pages 4–11. IEEE Computer Society, 1998.

[12] Pierre Boulet et Xavier Redon. « Communication pre-evaluation in HPF ». Dans *EUROPAR'98*, volume 1470 de *LNCS*, pages 263–272. Springer Verlag, 1998.

[13] Pierre Boulet et Xavier Redon. « SPPoC : manipulation automatique de polyèdres pour la compilation ». *Technique et Science Informatiques*, 20(8):1019–1048, 2001.

[14] Thomas Brandes. « *ADAPTOR Programmers Guide* ». GMD-SCAI, décembre 1999.

[15] S. Cabasino, P.S. Paolucci, et G.M. Todesco. « Dynamic parsers and evolving grammars ». *ACM SIGPLAN Notices*, 27, 1992.

[16] Emmanuel Cagniot. « *Algorithmes Data-parallèles Irréguliers Appliqués à la Simulation Éléctromagnétique Tridimensionelle* ». Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, décembre 2000.

[17] Emmanuel Cagniot, Thomas Brandes, Jean-Luc Dekeyser, et Francis Piriou. « Parallelization of a 3-D magnetostatic code using high performance fortran and the Schur complement method ». Dans *Conference on the Computation of Electromagnetic Fields, Compumag'13*, Évian, France, juillet 2001.

[18] Emmanuel Cagniot, Thomas Brandes, Jean-Luc Dekeyser, et Francis Piriou. « Une approche génie logiciel des codes de simulation irréguliers : Application au cas de l'electromagnétisme ». Dans *RenPar'13, Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, pages 115–120, Paris, France, juin 2001.

[19] Emmanuel Cagniot, Thomas Brandes, Jean-Luc Dekeyser, Francis Piriou, Pierre Boulet, et Stéphane Clenet. « High level parallelization of a 3D electromagnetic simulation code with irregular communication patterns ». Dans *4th International Meeting on Vector and Parallel Processing (VECPAR'2000)*, pages 519–528, Porto, Portugal, juin 2000. Lecture Notes in Computer Science vol. 1470.

[20] Emmanuel Cagniot, Thomas Brandes, Jean-Luc Dekeyser, Francis Piriou, Pierre Boulet, Stéphane Clénet, Yvonnick Le Menach, et Georges Marques. « Parallelization of a Fortran 90 program for electromagnetic problems ». Dans *3rd Annual HPF User Group Meeting, HUG'99*, Redondo Beach, CA, août 1999.

[21] Emmanuel Cagniot, Thomas Brandes, Jean-Luc Dekeyser, Francis Piriou, Pierre Boulet, et Georges Marques. « Parallelization of 3D magnetostatic code using High Performance Fortran ». Dans *International Conference on Parallel Computing in Electrical Engineering, PARELEC'2000*, pages 181–185, Trois-Rivières, Quebec, Canada, août 2000.

[22] Pierre-Yves Calland, Jack Dongarra, et Yves Robert. « Tiling with limited resources ». Dans L. Thiele, J. Fortes, K. Vissers, V. Taylor, T. Noll, et J. Teich, éditeurs, *Application Specific Systems, Achitectures, and Processors, ASAP'97*, pages 229–238. IEEE Computer Society Press, 1997.

[23] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, et Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, octobre 2000.

[24] Philippe Clauss. « Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs ». Dans *ACM Int. Conf. on Supercomputing*. ACM, mai 1996.

[25] Philippe Clauss et Vincent Loechner. « Parametric analysis of polyhedral iteration spaces ». Dans *IEEE Int. Conf. on Application Specific Array Processors, ASAP'96*. IEEE Computer Society, août 1996.

[26] Philippe Clauss, Vincent Loechner, et Doran K. Wilde. « Deriving formulae to count solutions to parameterized linear systems using ehrhart polynomials: Applications to the analysis of nested-loop programs ». Rapport Technique RR 97-05, Laboratoire Image et Calcul Parallèle Scientifique, April 1997. URL: `http://icps.u-strasbg.fr/PolyLib`.

[27] Jean-François Collard, Paul Feautrier, et Tanguy Risset. « Construction of DO loops from systems of affine constraints ». *Parallel Processing Letters*, 5(3):421–436, septembre 1995.

[28] Alain Darte, Claude Diderich, Marc Gengler, et Frédéric Vivien. « Scheduling the computations of a loop nest with respect to a given mapping ». *Lecture Notes in Computer Science*, 1900, 2001.

[29] Alain Darte et Yves Robert. « Constructive methods for scheduling uniform loop nests ». *IEEE Trans. Parallel Distributed Systems*, 5(8):814–822, 1994.

[30] Alain Darte et Yves Robert. « Affine-by-statement scheduling of uniform and affine loop nests over parametric domains ». *J. Parallel and Distributed Computing*, 29:43–59, 1995.

[31] Alain Darte, Yves Robert, et Frédéric Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000. `http://www.birkhauser.com/detail.tpl?isbn=0817641491`.

[32] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, et K. A. Vissers. « YAPI: Application modeling for signal processing systems ». Dans *37th Design Automation Conference*, Los Angeles, CA, juin 2000. ACM Press.

[33] Michèle Dion et Yves Robert. « Mapping affine loop nests: New results ». Dans Bob Hertzberger et Guiseppe Serazzi, éditeurs, *High-Performance Computing and Networking, International Conference and Exhibition*, volume LCNS 919, pages 184–189. Springer-Verlag, 1995. Extended version available as Technical Report 94-30, LIP, ENS Lyon (anonymous ftp to lip.ens-lyon.fr).

[34] Esterel Technologies. « Esterel suite for system-on-a-chip top level validation ». `http://www.esterel-technologies.com/v2/esterelSuiteForSystemOnAChipTopL%evelValidation/index.html`.

[35] Thomas Fahringer. « Compile-time estimation of communication costs for data parallel programs ». *Journal of Parallel and Distributed Computing*, 39(0153):46–65, 1996.

[36] Paul Feautrier. « Parametric integer programming ». *RAIRO Recherche Opérationnelle*, 22:243–268, septembre 1988.

[37] Paul Feautrier. « Dataflow analysis of array and scalar references ». *Int. J. Parallel Programming*, 20(1):23–51, 1991.

[38] Paul Feautrier. « Some efficient solutions to the affine scheduling problem, part I: one-dimensional time ». *Int. J. Parallel Programming*, 21(5):313–348, octobre 1992.

[39] Paul Feautrier. « Some efficient solutions to the affine scheduling problem, part II: multi-dimensional time ». *Int. J. Parallel Programming*, 21(6):389–420, décembre 1992.

[40] Paul Feautrier. « Towards automatic distribution ». *Parallel Processing Letters*, 4(3):233–244, 1994.

[41] Paul Feautrier et Nadia Tawbi. « Résolution de systèmes d'inéquations linéaires; mode d'emploi du logiciel PIP ». Rapport Technique 90-2, Institut Blaise Pascal, Laboratoire MASI (Paris), janvier 1990.

[42] Ian Foster et Carl Kesselman, éditeurs. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, juillet 1998. `http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-475-8`.

[43] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, et Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[44] S. Gissinger, P. Chaumes, J.-P. Antoine, A. Bihain, et M. Stubbe. « Advanced dispatcher training simulator ». *IEEE Computer Applications in Power*, 13(2), avril 2000.

[45] T. Grandpierre, C. Lavarenne, et Y. Sorel. « Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors ». Dans *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES99)*, pages 74–78, New York, mai 3–5 1999. ACM Press.

[46] Manish Gupta et Prithviraj Banerjee. « Compile-time estimation of communication costs of programs ». *Journal of Programming Languages*, 2(3):191–225, septembre 1994.

[47] Manish Gupta, Sayak Mukhopadhyay, et Navin Sinha. « Automatic parallelization of recursive procedures ». *International Journal of Parallel Programming*, 28(6):537–562, 2000.

[48] High Performance Fortran Forum. « High Performance Fortran Language Specification ». Rapport Technique 2.0, Rice University, janvier 1997.

[49] K. Högstedt, L. Carter, et J. Ferrante. « Determining the idle time of a tiling ». Dans *Principles of Programming Languages*, pages 160–173. ACM Press, 1997. Extended version available as Technical Report UCSD-CS96-489.

[50] Gilles Kahn. « The semantics of a simple language for parallel programming ». Dans Jack L. Rosenfeld, éditeur, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., août 1974.

[51] Gilles Kahn et David B. MacQueen. Coroutines and networks of parallel processes. Dans B. Gilchrist, éditeur, *Information Processing 77*, pages 993–998. North-Holland, 1977. Proc.IFIP Congress.

[52] Katarzyna Keahey. « PARDIS: Programmer-level abstractions for metacomputing ». *Future Generation Computer Systems*, 15(5–6):637–647, octobre 1999.

[53] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, et David Wonnacott. « The Omega Library Interface Guide ». Rapport Technique, Dept. of Computer Science, Univ. of Maryland, College Park, 1996. `http://www.cs.umd.edu/projects/omega/`.

[54] Wayne Kelly, William Pugh, et Evan Rosser. « Code generation for multiple mappings ». Dans *The 5th Symposium on Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, février 1995.

[55] Ken Kennedy et Ulrich Kremer. « Automatic data layout for High Performance Fortran ». Dans Sidney Karin, éditeur, *Proceedings of the 1995 ACM/IEEE Supercomputing*

Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. ACM Press and IEEE Computer Society Press.

[56] Edward A. Lee. « *Overview of the Ptolemy Project* ». University of California, Berkeley, mars 2001.

[57] Christian Lefebvre et Jean-Luc Dekeyser. « HPF-Builder: A visual environment to transform Fortran 90 codes to HPF ». *The Int'l Journal of Supercomputer Applications*, 11(2):95–102, Summer 1997 1997.

[58] Vincent Lefebvre et Paul Feautrier. « Optimizing storage size for static control programs in automatic parallelizers ». Dans *European Conference on Parallel Processing*, pages 356–363, 1997.

[59] Amy Wingmui Lim. « *Improving Parallelism and Data Locality with Affine Partitioning* ». PhD thesis, Stanford University, septembre 2001.

[60] Ewing Lusk et al.. « MPI-2: Extensions to the message-passing interface ». Rapport Technique, Message Passing Interface Forum, 1997. `http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0/mpi2-report%.htm`.

[61] Daniel Maufroid, Pier S. Paolucci, Philippe Kajfasz, et Alessandro Bertini. « mAgic FPU: VLIW floating point engines for "System-on-Chip" applications ». Dans *EMMSEC'99*, Stockholm, Sweden, 1999.

[62] Dror E. Maydan, Saman P. Amarasinghe, et Monica S. Lam. « Array-data flow analysis and its use in array privatization ». Dans ACM, éditeur, *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1993*, pages 2–15, New York, NY, USA, 1993. ACM Press.

[63] Hans Meuer, Erich Stromaier, Jack Dongarra, et Horst D. Simon. « Top500 supercomputers sites ». 19th edition, Mannheim University and University of Tennessee, juin 2002. `http://www.top500.org/`.

[64] B. Meyer et M. Stubbe. « EUROSTAG – a single tool for power system simulation ». *TRANSMISSION & DISTRIBUTION International*, mars 1992.

[65] Markus Mottl. « Automating functional program transformation ». Master's thesis, University of Edinburgh, septembre 2000. `http://www.ai.univie.ac.at/~markus/msc_thesis/`.

[66] Object Management Group, Inc., éditeur. *Common Object Request Broker Architecture (CORBA), Version 2.6*. `http://www.omg.org/technology/documents/formal/corba_iiop.htm`, décembre 2001.

[67] Object Management Group, Inc., éditeur. *Unified Modeling Language (UML), Version 1.4*. `http://www.omg.org/technology/documents/formal/uml.htm`, septembre 2001.

[68] Pier S. Paolucci, Philippe Kajfasz, Philippe Bonnot, Bernard Candaele, Daniel Maufroid, Elena Pastorelli, Andrea Ricciardi, Yves Fusella, et Eugenio Guarino. « mAgic-FPU and MADE: A customizable VLIW core and the modular vliw processor architecture description environment ». Dans *SIMAI 2000 Conference*, Ischia, Italy, juin 2000.

[69] Cristóbal Pareja, Ricardo Peña, Fernando Rubio, et Clara Segura. « Optimizing Eden by program transformation ». Dans *2nd Scottish Functional Programming Workshop, St. Andrews 2000*. Intellect, 2001.

[70] Thomas M. Parks. « *Bounded Scheduling of Process Networks* ». PhD thesis, University of California at Berkeley, 1995.

[71] Thierry Priol et Christophe René. « Cobra: A CORBA-compliant programming environment for high-performance computing ». Dans *Euro-Par'98*, numéro 1470 dans Lecture Notes in Computer Science, pages 1114–1122, Southampton, UK, novembre 1998.

[72] W. Pugh. « The Omega test: a fast and practical integer programming algorithm for dependence analysis ». Dans IEEE, éditeur, *Proceedings, Supercomputing '91: Albuquerque, New Mexico, November 18–22, 1991*, pages 4–13. IEEE Computer Society Press, 1991.

[73] William Pugh. « A practical algorithm for exact array dependence analysis ». *Communications of the ACM*, 8:27–47, août 1992.

[74] H. John Reekie. « *Realtime Signal Processing: Dataflow, Visual, and Functional Programming* ». PhD thesis, School of Electrical Engineering, University of Technology at Sydney, septembre 1995.

[75] Manuel Serrano et Pierre Weis. « Bigloo: A portable and optimizing compiler for strict functional languages ». Dans *Static Analysis Symposium*, pages 366–381, 1995.

[76] Yves Sorel. « Massively parallel computing systems with real time constraints - the "Algorithm Architecture Adequation" methodology ». Dans *Proceedings of the 1st International Conference on Massively Parallel Computing Systems*, pages 44–54, Los Alamitos, CA, USA, mai 1994. IEEE Computer Society Press.

[77] Yves Sorel et Christophe Lavarenne. « Modèle unifié pour la conception conjointe logiciel-matériel ». *Traitement du Signal (numéro spécial Adéquation Algorithme Architecture)*, 14(6):569–578, 1997.

[78] Yves Sorel et Christophe Lavarenne. « *SynDEx Documentation Index* ». INRIA, 2000. `http://www-rocq.inria.fr/syndex/doc/`.

[79] Julien Soula. « *Principe de Compilation d'un Langage de Traitement de Signal* ». Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, décembre 2001.

[80] William Thies, Frederic Vivien, Jeffrey Sheldon, et Saman P. Amarasinghe. « A unified framework for schedule and storage optimization ». Dans *SIGPLAN Conference on Programming Language Design and Implementation*, pages 232–242, 2001.

[81] Peng Tu et David Padua. « Chapter 8. automatic array privatization ». *Lecture Notes in Computer Science*, 1808, 2001.

[82] Aad J. van der Steen et Jack J. Dongarra. « Overview of recent supercomputers ». 12th edition, Top500 Supercomputers Sites, juin 2002. `http://www.top500.org/ORSC/2002/`.

[83] Darren Webb, Andrew Wendelborn, et Kevin Maciunas. « Process networks as a high-level notation for metacomputing ». Dans *Workshop on Java for Parallel and Distributed Computing (IPPS)*, Puerto Rico, avril 1999.

[84] Darren Webb, Andrew Wendelborn, et Julien Vayssière. « A study of computational reconfiguration in a process network ». Dans *IDEA'7*, Victor Harbour, South Australia, février 2000.

[85] Doran K. Wilde. « A library for doing polyhedral operations ». Rapport Technique Internal Publication 785, IRISA, Rennes, France, Dec 1993. Also published as INRIA Research Report 2157.

[86] Michael E. Wolf et Monica S. Lam. « A loop transformation theory and an algorithm to maximize parallelism ». *IEEE Trans. Parallel Distributed Systems*, 2(4):452–471, octobre 1991.

Annexe A

SPPoC : manipulation automatique de polyèdres pour la compilation

SPPoC : manipulation automatique de polyèdres pour la compilation

Pierre Boulet — Xavier Redon

Laboratoire d'Informatique Fondamentale de Lille (LIFL),
Université des Sciences et Technologies de Lille,
Bâtiment M3, Cité Scientifique,
59 655 Villeneuve d'Ascq cedex
{Pierre.Boulet, Xavier.Redon}@lifl.fr
http://www.lifl.fr/west/sppoc/

RÉSUMÉ. *Le modèle polyèdrique est très souvent utilisé pour l'analyse et la transformation de programme dans les compilateurs parallèliseurs. Les prototypes de recherche dans ce domaine utilisent donc souvent des outils comme PIP (résolution paramétrique de problèmes linéaires), la PolyLib (bibliothèque de manipulation de polyèdres) ou Omega (bibliothèque et interface de manipulation de formules de Presburger). Les deux principaux problèmes de ces outils sont leur manque de convivialité et des modules de simplification trop primitifs. Le manque de simplification fait que l'enchaînement de calculs conduit à des résultats incompréhensibles ou qui n'aboutissent pas pour des problèmes de mémoire ou de temps. La bibliothèque SPPoC (Symbolic Parameterized Polyhedral Calculator) résoud ces problèmes grâce à son interface totalement symbolique et à des modules de simplification des résultats plus poussés. Elle permet aussi d'unifier différents outils. La présentation de SPPoC est illustrée par deux applications : une application de génération de code et une application d'estimation de volume de communications.*

ABSTRACT. *The polyhedral model is quite popular in the field of parallelizing compilers. So, research prototypes tend to use tools like PIP (parametric integer programming solver), the PolyLib (library for polyhedra manipulation) or Omega (library and calculator for Presburger formulas). The two main drawbacks of these tools are a poor human-computer interface and a lack of aggressive simplification. This last deficiency leads to sequences of computations which give too complex results or even that cannot be completed due to memory exhaustion or time constraints. The SPPoC (Symbolic Parameterized Polyhedral Calculator) library brings a solution to these problems due to its completely symbolic interface and to its advanced simplification modules. It also allows the unification of different tools. We present two applications which use SPPoC : a code generator and a communication volume estimator.*

MOTS-CLÉS : *Polyèdres, Calcul symbolique, Génération de code, Transformations de boucles, Data-parallélisme, Volume de communications, PIP, PolyLib, Omega Library*

KEYWORDS: *Polyhedra, Symbolic Computation, Code Generation, Loop Transformations, Data-Parallelism, Communication Volume, PIP, PolyLib, Omega Library*

1. Introduction

Lors de la compilation de langages parallèles ou de la parallélisation automatique de programmes séquentiels, des analyses statiques du code sont nécessaires. Celles-ci font souvent appel à des calculs polyédriques lorsqu'on s'intéresse à des langages à contrôle statique (voir le chapitre 7 pour des exemples précis). Ce modèle polyédrique permet l'expression des dépendances de données [FEA 91, PUG 92, CLA 96b, CLA 96a], des calculs d'ordonnancement [WOL 91, FEA 92a, FEA 92b, DAR 94, DAR 95] et de placement de tâches parallèles [FEA 94, DIO 95], des optimisations de génération de code [COL 95, KEL 95, BOU 98a], etc.

Il existe de nombreux outils de calcul polyédrique numérique, mais nous n'en connaissons que trois qui permettent de manipuler des expressions paramétriques. Il s'agit de PIP [FEA 88, FEA 90], de la PolyLib [WIL 93, CLA 97] et de l'Omega Library [PUG 91, KEL 96]. PIP et la PolyLib étant fortement utilisés dans la communauté de la parallélisation automatique, nous avons réalisé la couche symbolique qui les rendra plus agréables et accessibles. Ces développements sont regroupés dans une bibliothèque : SPPoC pour *Symbolic Parameterized Polyhedral Calculator*. À la suite de demandes de la dite communauté, l'Omega Library est aussi intégrée dans SPPoC bien que disposant déjà d'une interface symbolique.

PIP permet de résoudre des problèmes de programmation linéaire paramétrée en nombres entiers. Un tel problème est la donnée d'un polyèdre définissant le domaine de recherche, d'un autre domaine définissant le domaine des valeurs des paramètres et d'une liste de variables. PIP retourne le minimum lexicographique de ces variables dans leur domaine de validité sous la forme d'un QUAST (Quasi Affine Selection Tree).

La PolyLib, quant à elle, permet la manipulation de polyèdres paramétrés (définition, opérations ensemblistes, application de fonctions affines et de leur réciproque, comptage du nombre de points, etc).

Enfin l'Omega Library qui manipule l'arithmétique de Presburger est partiellement redondante avec PIP et la PolyLib mais apporte d'autres fonctionnalités comme le calcul d'une fermeture transitive de relation.

Le premier inconvénient de ces outils (du moins en ce qui concerne PIP et la PolyLib) est leur difficulté d'utilisation venant principalement des structures de données de bas niveau utilisées. En effet, les polyèdres et autres structures de données sont représentés par des matrices de coefficients. Les variables et les paramètres sont repérés par leur indice de ligne ou de colonne. Ces représentations, bien qu'efficaces pour le calcul, ne sont que très peu lisibles. Le second inconvénient (et certainement le plus important) est que les résultats ne sont pas forcément sous leur forme la plus simple. Cela se fait cruellement sentir quand on enchaîne plusieurs calculs. La complexité des objets traités augmente alors rapidement, rendant impossibles de nouveaux calculs.

Pour résoudre ces deux problèmes, nous avons développé une interface unifiée à ces outils. Elle est complètement symbolique, donc beaucoup plus utilisable, et fait des simplifications qui permettent des enchaînements de calculs jusqu'alors impossibles.

La suite de cet article est structurée ainsi : après une présentation succinte de l'interface utilisateur dans le chapitre 2, nous décrivons successivement les quatre composants de SPPoC : l'interface à PIP (chapitre 3), l'interface à la PolyLib (chapitre 4), l'interface à l'Omega Library (chapitre 5) et le fonctionnement des simplifications symboliques (chapitre 6). Enfin, le chapitre 7 présente deux exemples d'applications bénéficiant de cette interface de haut niveau.

2. Interface utilisateur

2.1. *Choix d'implémentation*

SPPoC est une bibliothèque pour le langage Objective Caml [PRO]. Celle-ci est en fait constituée d'un module qui regroupe plusieurs sous-modules donnant accès à ses différentes fonctionnalités.

Nous avons choisi le langage Objective Caml pour ses capacités de traitement de structures de données complexes et symboliques, son expressivité et sa possibilité d'interfaçage direct avec le langage C, utilisé pour l'implémentation de PIP et de la PolyLib, l'Omega Library étant écrite en C++. Le code représente au final environ 9000 lignes de code Objective Caml et 6000 lignes de code C.

Pour une utilisation facile, il faut un moyen de décrire les données manipulées avec une syntaxe humainement lisible. Nous avons utilisé le préprocesseur camlp4 [RAU] pour construire des « quotations ». Il s'agit de chaînes de caractères entre <: s< et >> où s est un sélecteur qui indique le type de la quotation. D'autre part, un ensemble d'imprimeurs a été défini qui permet un affichage automatique « en clair » des structures de données symboliques.

Bien que SPPoC ait été conçu comme une bibliothèque destinée à être intégrée à des outils tels que des compilateurs, il peut aussi être utilisé comme une calculatrice. En effet, grâce à l'utilisation d'Objective Caml comme langage de programmation, la construction d'un système interactif pour SPPoC a été quasi immédiate : Objective Caml dispose d'une boucle interactive de compilation, évaluation, impression du résultat. Chaque commande tapée au clavier est typée, compilée puis exécutée. Il suffit de lier la bibliothèque contenant le module SPPoC avec ce système interactif et toutes les fonctions de la bibliothèque deviennent accessibles.

Nous présentons ci-dessous dans un premier temps la structure des modules composant la bibliothèque puis comment utiliser SPPoC de façon interactive en ligne de commande.

2.2. *Interface de programmation*

Le module SPPoC propose un ensemble de types abstraits (chacun représenté par un module distinct) qui permettent les manipulations de calcul symbolique. Les

principaux types abstraits sont :

– les formes affines (module Form) ;

– les expressions arithmétiques générales (module Expr) ;

– les (in)équations affines (module Ineq) ;

– les QUAST (QUasi Affine Selection Tree) (module Quast) ;

– les QUAST étendus (module Equast), qui sont des QUAST dans lesquels on peut avoir des variables qui représentent des restes de divisions euclidiennes d'une forme linéaire par un entier.

On définit ensuite le module PIP qui permet la résolution de problèmes de programmation linéaire paramétrée (par l'appel à PIP). Ces problèmes sont caractérisés par un système d'inéquations affines et par une question (module Question). La solution d'un tel problème est donnée sous la forme d'un QUAST étendu. Le module Polyhedron implante toutes les fonctions de la PolyLib et le module Presburger implante certaines fonctions de l'Omega Library.

Tous les modules définis ici, sauf le module PIP représentent des types abstraits et sont construits selon le même schéma : un type abstrait `t`, des constructeurs, des destructeurs, éventuellement des opérateurs sur ce type, et des fonctions d'entrée-sortie : un constructeur à partir d'une chaîne de caractères, `read`, à base d'une grammaire gérée par camlp4 et un imprimeur, `print`.

2.3. *Utilisation interactive*

Les apports particuliers pour une utilisation interactive de SPPoC sont d'une part les mécanismes d'entrées-sorties (quotations et imprimeurs) et la présence d'une fonction `help` documentant chaque module. Les différentes quotations définies sont décrites dans la table 1. Leur syntaxe est très permissive. En effet, l'utilisateur peut séparer une liste d'objets en utilisant une virgule ou un point-virgule, il peut les encadrer par des délimiteurs optionnels (accolades ou crochets), ce qui permet des notation ensemblistes (accolades et points-virgules), de style listes Caml (crochets et points-virgules) ou libre (pas de délimiteurs et virgules). Certains opérateurs arithmétiques ont plusieurs notations, par exemple le modulo (`mod` ou `%`) ou l'élévation à une puissance (`^` ou `**`). Le but de cette permissivité est la facilité d'utilisation. L'utilisateur doit se concentrer sur son problème et non sur la syntaxe à employer.

Dans toute la suite de l'article, nous illustrons les fonctionnalités de SPPoC par des exemples. Ceux-ci sont des extraits de sessions d'utilisation de l'outil interactif, ils sont présentés comme ci-dessous :

```
# let d = <:s< 1<=i<=n, 1<=j<=i >>;;
val d : SPPoC.System.t = i >= 1, i <= n, j >= 1, i >= j
# Polyhedron.of_system d;;
- : SPPoC.Polyhedron.t =
```

donnée	type	exemples
liste de variables	`string list`	`<:v<a,a',a''>> <:v<{i;j;k;l}>>`
forme affine	`Form.t`	`<:f<7+4*i-j+3*(i+j-k)>>`
expression	`Expr.t`	`<:e<(3i-k^x+n) div (2p) - (3j-k-m) % q>>`
(in)équation	`Ineq.t`	`<:i<2*i-(j div 3)<n*m>>` `<:i<i=j>> <:i<0<=2*k>>`
système	`System.t`	`<:s<2i+j=n, 0<=i<=n, j<=n, j<i>>` `<:s<{1<=i1<=n, 1<=j1<=n, i1=j1, j1%8=k1%8}>>` `<:s<[a+b<c<=2*a; a>=0]>>`
fonction	`Function.t`	`<:fct<x;y <- 2*i % p, i^(p%2)>>` `<:fct<[p;q] <- {i+j, n-i-j}>>`
rayons	`Rays.t`	`<:r<{line(3*i-2*j), ray(-j), vertex((35*j+12*k)/21)}>>`

Tableau 1. *Quotations disponibles dans SPPoC*

```
{({i >= j, j >= 1, i <= n},
  {vertex(i+j+n), ray(n), ray(i+n), ray(i+j+n)})}
```

Le caractère `#` est le message d'invite du système interactif, chaque phrase tapée par l'utilisateur est imprimée en italique et terminée par `;;`. La réponse du système se décompose en 3 parties :

– avant les `:`, le mot clé `val` suivit du nom de la valeur définie par une construction du style « `let` nom `=` valeur » ou `-` s'il s'agit d'une simple évaluation sans nommage ;

– entre les `:` et le signe `=`, le type de l'expression résultant de l'évaluation de la phrase de l'utilisateur (inféré par le système de typage d'Objective Caml) ;

– après le signe `=`, la valeur telle qu'affichée par l'imprimeur associé à son type.

3. Programmation linéaire en nombres entiers paramétrée

Nous présentons ici la façon dont l'interface à PIP a été réalisée et ce qu'elle apporte en facilité d'utilisation.

3.1. *PIP*

PIP est l'acronyme de *Parameterized Integer Programming* ou programmation linéaire paramétrée en nombres entiers. Cet outil permet de calculer le minimum lexicographique d'un domaine polyédrique paramétré.

La solution d'un tel problème est représentée par un QUAST (Quasi Affine Selection Tree), c'est-à-dire un arbre de sélection dont les prédicats sont des inégalités affines et les feuilles des listes de formes affines ou $\perp$ signifiant qu'il n'y a pas de solution pour la branche considérée. Une branche de QUAST définit un domaine de paramètres et la valeur du minimum lexicographique du polyèdre pour des paramètres à valeurs dans ce domaine.

Étant donné que PIP peut travailler en nombres entiers, pour exprimer la solution, on peut avoir besoin de divisions entières. Pour conserver des formes affines dans les prédicats de sélection et les feuilles, PIP peut introduire des « nouveaux paramètres » définis comme la division entière d'une forme affine par un entier.

Voici un exemple d'utilisation de PIP tiré de [FEA 90] : minimiser (i, j) tels que

$$\begin{cases} i \leq n \\ j \leq p \\ 2i + j = 2n + p - m \end{cases}, \tag{1}$$

toutes les variables et tous les paramètres étant positifs ou nuls. Voici sous quelle forme il est traité par PIP :

problème posé	réponse

`((commentaire)`
` 2 3 4 0 -1 1`
` (#[-1 0 0 0 1 0]`
`  #[0 -1 0 0 0 1]`
`  #[-2 -1 0 -1 2 1]`
`  #[2 1 0 1 -2 -1])`
` ()`
`)`
	`((commentaire -1 )`
	` (if #[ -1 2 1 0]`
	`  (if #[ 1 -2 0 0]`
	`  (list #[ 0 0 0 0]`
	`  #[ -1 2 1 0])`
	`  (newparm 3 (div #[ 1 0 0 0] 2))`
	`  (if #[ -1 0 1 2 0]`
	`  (list #[ 0 1 0 -1 0]`
	`  #[ -1 0 1 2 0])`
	`  ()))`
	`  ()))`

L'interprétation de la réponse est

si $-m + 2n + p \geq 0$
alors | **si** $m - 2n \geq 0$
| **alors** $[0, -m + 2n + p]$
| **sinon** | **si** $-m + p + 2(m/2) \geq 0$
| | **alors** $[n - (m/2), -m + p + 2(m/2)]$
| | **sinon** $\perp$
sinon $\perp$. (2)

3.2. *Limitations de PIP*

Comme on peut le voir sur l'exemple précédent, la syntaxe utilisée n'est pas très lisible, bien qu'efficace pour le calcul. Une autre limitation de PIP, qui tient à l'al-

gorithme utilisé (et qui garantit une solution), est que toutes les variables et tous les paramètres sont considérés à valeurs positives. Ce qui peut amener l'utilisateur à faire des changements de variables pour pouvoir résoudre son problème. De même il est facile en utilisant cette même méthode de faire calculer à PIP des maxima lexicographiques et des minima et des maxima de fonctions objectives affines.

3.3. *Interface à PIP*

Ce que SPPoC propose dans son interface répond aux problèmes évoqués ci-dessus. En effet, la syntaxe proposée est complètement symbolique comme on peut le voir dans la version SPPoC de notre exemple [1] :

```
# let p = PIP.make <:s<i<=n, j<=p, 2*i+j=2*n+p-m>>
                   (Question.min_lex <:v<i, j>>);;
val p : SPPoC.PIP.t =
  Solve : MIN(i, j)
  Under the constraints :
  {i <= n, j <= p, 2*i+j+m = 2*n+p}
# PIP.solve_pos p;;
- : SPPoC.EQuast.t =
Parameters :
np1 = m mod 2
Quast :
if m <= 2*n+p then
  if m >= 2*n then
    [0; -m+2*n+p]
  else if np1 <= p then [(-m+2*n+np1)/2; -np1+p] else _|_
else _|_
```

Le résultat obtenu est de forme légèrement différente (mais équivalente, bien sûr) de celle qui a été obtenue en utilisant PIP directement (voir équation (2)) car SPPoC représente les nouveaux paramètres sous forme de modulos plutôt que sous forme de divisions entières. Cela a pour effet de simplifier les expressions obtenues et en particulier de faciliter les changements de variables. Cette constatation est empirique et sa cause en est encore mal comprise.

Les changements de variables pour se libérer de la contrainte de positivité et permettre les maxima et les fonctions objectives sont faits automatiquement. Par exemple, pour calculer le maximum de $i-j$ sur le domaine $-n \leq i \leq n, 0 \leq j \leq n-i, n \geq 10$, il suffit d'écrire:

```
# PIP.solve (PIP.make <:s<-n<=i<=n; 0<=j<=n-i; n>=10>>
                      (Question.max_fun <:f<i-j>>));;
- : SPPoC.EQuast.t = [n]
```

1. Nous utilisons ici la fonction *PIP.solve_pos* qui suppose que les variables et les paramètres sont tous positifs. On évite ainsi des changements de variables inutiles.

Le dialogue entre SPPoC et PIP se fait à travers un appel de fonction C[2]. SPPoC autorise aussi la manipulation des QUAST et permet ainsi l'enchaînement de calculs avec plusieurs appels à PIP. Pour une description des opérations possibles sur les QUAST, voir la section 6.4. Le chapitre 7.1 présente une utilisation intensive de cette interface où plusieurs dizaines d'appels à PIP sont nécessaires pour produire un résultat.

4. Opérations sur les polyèdres

Pour les opérations sur les polyèdres SPPoC repose presque entièrement sur la bibliothèque polyédrique connue sous le nom de PolyLib [WIL 93]. À l'origine, la PolyLib a été développée à l'IRISA de Rennes par Doran Wilde dans le cadre du projet Alpha. Une seconde version de la PolyLib a vu le jour en collaboration avec Philippe Clauss et Vincent Loechner de l'ICPS à Strasbourg [CLA 97]. Cette version permet de manipuler des polyèdres paramétrés, en particulier de trouver leurs sommets et de compter le nombre de points entiers qu'ils contiennent.

L'interface entre SPPoC et la PolyLib utilise le fait que des routines écrites en C peuvent être utilisées en Objective Caml par simple édition de liens. Le plus difficile est d'écrire les fonctions de conversion des structures Caml en structures C et vice-versa.

4.1. *Construction des polyèdres*

La PolyLib manipule en fait des unions de polyèdres, chaque polyèdre étant défini par un couple formé de ses contraintes et d'une représentation duale à base de sommets, de rayons et de droites. Ces informations sont redondantes, la forme duale pouvant être obtenue à partir des contraintes et réciproquement, mais elles permettent d'effectuer les calculs plus rapidement. Pour la même raison SPPoC manipule des listes de couples.

Il est possible de créer un polyèdre directement à partir des deux informations mais c'est déconseillé car aucun contrôle de cohérence n'est effectué. On peut cependant vouloir utiliser cette possibilité pour des raisons de performance. Il est beaucoup plus pratique de créer un polyèdre à partir d'un simple système de contraintes :

```
# let p = Polyhedron.of_system <:s<3*i+j>=10, 1<=i<=n, 1<=j<=m>>;;
val p : SPPoC.Polyhedron.t =
  {({j <= m, j >= 1, i <= n, i >= 1, 3*i+j >= 10},
    {ray(m), ray(n), vertex(i+7*j+7*m+n),
     ray(j+m), ray(i+n), vertex(3*i+j+m+3*n)})}
```

2. Nous utilisons ici une version modifiée de PIP qui permet cet appel de fonction.

La présence de rayons dans la forme duale (calculée à partir des contraintes) nous apprend qu'il ne s'agit pas d'un polyèdre borné. En effet, le translaté d'un polyèdre suivant la direction d'un de ses rayons est inclus dans le polyèdre original.

Il est aussi possible de créer un polyèdre en spécifiant sa forme duale. Prenons l'exemple d'une bande diagonale dans un espace à deux dimensions dont les bords passent par les points $(0, 1)$ et $(1, 0)$. Sa représentation duale est constituée de ces deux sommets et de la ligne de vecteur directeur $(1, 1)$:

```
# let p = Polyhedron.of_rays <:r<vertex(i), vertex(j), line(i+j)>>;;
val p : SPPoC.Polyhedron.t =
  {({i-j <= 1, i-j >= -1}, {line(i+j), vertex(j), vertex(-j)})}
```

La représentation de `p` fait bien apparaître les deux inéquations définissant la bande.

De plus SPPoC permet la création de polyèdres ne contenant aucun point ou tous les points d'un espace défini par le nom des variables associées à ses dimensions. Enfin, des fonctions sont disponibles pour créer directement des *unions* de polyèdres et pour récupérer la liste des contraintes ou la liste des représentations duales de telles unions. Les unions de polyèdres sont représentées par une liste de polyèdres. La calculatrice SPPoC présente toujours des unions préalablement simplifiées par la PolyLib : les polyèdres des listes sont disjoints.

4.2. *Opérations ensemblistes*

SPPoC reprend les opérations sur les polyèdres offertes par la PolyLib : intersection, union, soustraction, image ou pré-image par une fonction affine, calcul de l'enveloppe convexe, simplification dans le contexte d'une autre union de polyèdres, test d'inclusion et enfin test de vacuité.

Il est important de comprendre que SPPoC construit les polyèdres en ne tenant compte que des variables qui interviennent dans les contraintes ou la représentation duale. Supposons qu'un utilisateur souhaite calculer l'intersection entre une droite horizontale dans le plan et un polyèdre plus complexe de dimension deux. La définition de la droite peut se faire comme suit :

```
# let d = Polyhedron.of_system <:s< i=4 >>;;
val d : SPPoC.Polyhedron.t = {({i = 4}, {vertex(4*i)})}
```

Ceci définit un polyèdre de dimension 1 (il est possible de s'en convaincre en demandant la liste des variables ; elle se limite à `i`). Le convexe de dimension deux peut, lui, se définir par :

```
# let p = Polyhedron.of_system <:s<2*i+j>1, i-4*j<10, j<=20>>;;
val p : SPPoC.Polyhedron.t =
  {({j <= 20, i-4*j <= 10, 2*i+j >= 1},
    {vertex((14*i-19*j)/9), vertex((-19*i+40*j)/2), vertex(90*i+20*j)})}
```

Lors du calcul d'intersection de la droite `d` et du polyèdre `p`, SPPoC se rend compte que les deux polyèdres n'ont pas la même dimension et cherche à les plonger dans un espace commun. En l'occurrence un espace à deux dimensions (associées aux variables `i` et `j`). Le résultat de l'intersection est un segment de droite :

```
# Polyhedron.inter d p;;
- : SPPoC.Polyhedron.t =
{({i = 4, j <= 20, 2*j >= -3}, {vertex((8*i-3*j)/2), vertex(4*i+20*j)})}
```

Dans tous les cas, il est possible de trouver un espace commun : SPPoC récupère les ensembles de variables des unions de polyèdres, en calcule l'union V et plonge les deux unions de polyèdres dans un espace ayant autant de dimensions que V a d'éléments. L'identification des dimensions communes se fait par égalité des noms de variables les caractérisant. Il n'y a pas de renommage automatique.

Pour les images et pré-images de polyèdres par une fonction affine, un mécanisme semblable est mis en place. Pour comprendre l'utilité de l'auto-ajustement en présence de fonctions, prenons l'exemple de la projection du polyèdre ci-dessous selon la direction associée à `i` :

```
# let q = Polyhedron.of_rays
          <:r< vertex(i+5j), vertex(2i-j),
               vertex(3j), vertex(2i) >>;;
val q : SPPoC.Polyhedron.t =
  {({i <= 2, 5*i+j <= 10, 2*i-j >= -3, 2*i+j >= 3},
    {vertex(i+5*j), vertex(2*i-j), vertex(3*j), vertex(2*i)})}
```

La fonction à utiliser est `<:fct< i <- i >>`.[3] Notre fonction de projection se retrouve donc avec un espace de départ et d'arrivée à une dimension. SPPoC va ajuster automatiquement la dimension de cet espace lors du calcul de l'image :

```
# Polyhedron.image q <:fct< i <- i >;;
- : SPPoC.Polyhedron.t = {({i >= 0, i <= 2}, {vertex(0), vertex(2*i)})}
```

Si, au contraire, on souhaite plonger `q` dans un espace avec un plus grand nombre de dimensions (par exemple, en utilisant `<:fct< i,j,k <- i,j,k >>`), SPPoC va automatiquement ajuster la dimension de `q` avant d'appliquer la fonction. L'utilisateur n'a donc pas, en principe, à se préoccuper des dimensions de ses fonctions et de ses

3. Il faut noter que les fonctions de SPPoC permettent de faire du renommage de variables au vol : à gauche du symbole `<-` se trouve la liste des nouveaux noms de variables et à droite apparaissent les expressions correspondant aux nouvelles variables. Ces expressions s'écrivent, bien entendu, en fonction des anciens noms de variables. Ainsi la fonction `<:fct< i',j' <- j,i >>` représente une symétrie par rapport à la diagonale principale avec renommage des variables. Le fait de ne pas pouvoir introduire de nouvelle variable non contrainte dans l'ensemble d'arrivée de la fonction n'est pas une limitation grâce l'auto-ajustement réalisé par SPPoC.

polyèdres. Cependant SPPoC permet de forcer le plongement d'une fonction ou d'une union de polyèdres dans un espace de dimension supérieure.

Par contre, l'utilisateur doit absolument utiliser les mêmes variables pour référencer les dimensions des espaces dans lesquels sont définis ses polyèdres. Si pour une raison ou une autre, ce n'est pas possible lors de la définition des polyèdres, SPPoC permet, par la suite, de renommer les variables liées aux dimensions.

4.3. *Opérations sur les polyèdres paramétriques*

Même sans opération spécifique, la PolyLib permet de manipuler des polyèdres paramétriques en ajoutant des dimensions supplémentaires pour les paramètres. Par exemple il est possible de définir un carré paramétré par la taille `n` de ses cotés :

```
# let carre = Polyhedron.of_system <:s< 0<=i<=n, 0<=j<=n >>;;
val carre : SPPoC.Polyhedron.t =
  {({j <= n, j >= 0, i <= n, i >= 0},
    {vertex(0), ray(n), ray(i+n), ray(i+j+n), ray(j+n)})}
```

Il est possible de faire des opérations ensemblistes sur ces polyèdres paramétriques. Par contre, la représentation duale de ces objets est déconcertante ; elle ne donne plus les sommets des polyèdres bornés. En effet, pour « paramétrer » le polyèdre, des dimensions représentant les paramètres sont ajoutées. La représentation duale tient compte de ces dimensions et, comme les paramètres sont généralement non bornés, on se retrouve avec des lignes ou des rayons suivant les dimensions paramétriques.

Il existe une version étendue de la PolyLib qui est celle à laquelle SPPoC propose une interface et qui permet de trouver les sommets d'un polyèdre paramétrique et de calculer le nombre de points entiers contenus dans un tel polyèdre. Voici un appel à l'extension de la PolyLib qui donne les sommets d'un polyèdre paramétrique sous sa forme habituelle, c'est à dire eux-mêmes paramétrés :

```
# Polyhedron.vertices
    carre ( Polyhedron.of_system <:s< n>=0 >> );;
- : (SPPoC.Polyhedron.t * SPPoC.Expr.t list list) list =
[{({n >= 0}, {ray(n), vertex(0)})}, [[n; n]; [n; 0]; [0; n]; [0; 0]]]
```

On reconnait bien ici les sommets du carré : $(0,\ 0)$, $(0,\ n)$, $(n,\ 0)$ et $(n,\ n)$, ainsi que la définition du domaine du paramètre n : $n \geq 0$.

Pour compter le nombre de points entiers contenus dans un polyèdre paramétrique borné[4] l'extension de la PolyLib utilise les polynômes de Ehrhart. Les polynômes de

4. le comptage des points d'un polyèdre non borné produit un résultat indéterminé

Ehrhart sont des polynômes à coefficients périodiques. En fait les coefficients sont des couples formés d'une liste de nombres et d'un paramètre. La valeur du coefficient est le nombre de la liste de rang le paramètre modulo la taille de la liste. Intuitivement, ces coefficients périodiques sont utiles pour construire des expressions dépendant de paramètres en évitant une définition au cas par cas suivant les valeurs de ces paramètres. Prenons l'exemple du polynôme $nm + [\frac{1}{2}, \frac{2}{3}]_n m + [0, \frac{1}{3}]_m n + 1$, sa valeur pour $n = 2$ et $m = 3$ est $2 \times 3 + \frac{1}{2} \times 3 + \frac{1}{3} \times 2 + 1$. Deux exemples de calcul du nombre de points entiers sont donnés ci-après. Les résultats sont, bien entendu, donnés en fonction des paramètres.

```
# Polyhedron.enum
    carre
    (Polyhedron.of_system <:s< n>=0 >>);;
- : SPPoC.Polyhedron.expr_quast =
[{({n >= 0}, {ray(n), vertex(0)})}, 1+2*n+n**2]
# let rectangle =
    Polyhedron.of_system <:s< 0<=i<=n, 0<=j<=n/3 >>;;
val p : SPPoC.Polyhedron.t =
  {(3*j <= n, j >= 0, i <= n, i >= 0,
    vertex(0), ray(n), ray(i+n), ray(3*i+j+3*n), ray(j+3*n))}
# Polyhedron.enum
     rectangle
    (Polyhedron.of_system <:s< n>=0 >>);;
- : SPPoC.Polyhedron.expr_quast =
[{({n >= 0}, {ray(3*n), vertex(0)})},
 1/3*n**2+n*[4/3, 1, 2/3]n+[1, 2/3, 1/3]n]
```

L'opération de comptage de points entiers est coûteuse en mémoire et en temps. En appelant directement la fonction de comptage de la PolyLib, seuls des polyèdres de petite dimension peuvent être traités. Pour parvenir à traiter des polyèdres plus complexes, SPPoC offre une fonction de plus haut niveau `Polyhedron.enumeration`. Cette fonction utilise le fait que la plupart des polyèdres manipulés peuvent s'écrire sous la forme d'un produit cartésien. Un résultat bien connu est que le cardinal d'un produit cartésien est le produit des cardinaux des ensembles utilisés dans le produit. Le fait de lancer le calcul du nombre de points sur plusieurs polyèdres de faible dimension plutôt que sur le polyèdre original permet de limiter l'explosion combinatoire. L'exemple le plus frappant est celui d'un hyper-cube de dimension d ; un tel polyèdre est le produit cartésien de d polyèdres de dimension 1.

L'algorithme de décomposition d'un polyèdre en produit cartésien consiste à partitionner l'ensemble des contraintes définissant le polyèdre. On regroupe deux contraintes dans le même sous-ensemble si et seulement si elles ont au moins une variable en commun. À la partition de l'ensemble des contraintes correspond donc une partition de l'ensemble des variables (toutes les variables apparaissent dans les contraintes puisque le polyèdre doit être borné). Le polyèdre original est le produit cartésien des polyèdres définis par les sous-ensembles de contraintes sur les sous-ensembles de variables correspondants.

5. Opérations sur les relations en nombres entiers

SPPoC intègre aussi une bibliothèque de manipulation de relations sur des vecteurs d'entiers nommé Omega Library (voir [PUG 91]). Cette bibliothèque permet de faire des opérations sur les polyèdres entiers comme la PolyLib mais apporte aussi des fonctions supplémentaires comme le calcul de la fermeture transitive d'une relation. Le principal problème rencontré lors de l'intégration de l'Omega Library dans SPPoC tient au fait que l'Omega Library est écrite en C++. L'interfaçage d'Objective Caml avec ce langage est un peu moins aisé que l'interfaçage avec C. Pour l'instant le support de l'Omega Library dans SPPoC est expérimental.

5.1. *Construction des relations*

Il a déjà été dit que la structure de base de l'Omega Library est la relation entre vecteurs d'entiers. Voici un exemple de relation de l'Omega Library telle que l'affiche SPPoC :

```
{ [i; j] -> [i'; j'], {i = i', j <= m, j >= 1, i' <= n, i' >= 1} }
```

Celle-ci met en relation un vecteur d'entrée `[i; j]` et tous les vecteurs de sortie `[i; j']` avec `j'` compris entre `1` et `m`. Il faut noter, qu'à l'opposé de la PolyLib, l'Omega Library permet de manipuler directement des expressions paramétriques. Par exemple, dans la relation ci-dessus, les symboles `n` et `m` n'apparaissent pas dans les variables d'entrée ou de sortie, ils sont donc considérés comme des paramètres.

Les relations de l'Omega Library peuvent être définies à l'aide de formules de Presburger, c'est à dire avec les opérateurs logiques $\neg$, $\vee$ et $\wedge$, des contraintes affines en les variables et des quantificateurs $\exists$ et $\forall$. La méthode actuelle de construction des relations avec SPPoC ne permet pas d'introduire de quantificateur $\forall$, par contre les quantificateurs $\exists$ sont autorisés. En fait une relation de l'Omega Library est construite à partir de la liste des paramètres, de la liste des variables d'entrée, de la liste des variables de sortie et d'une liste de systèmes de contraintes. Toutes les variables apparaissant dans les systèmes mais dans aucune des trois listes de variables sont considérées comme des variables quantifiées au sens de $\exists$. Voici un exemple de construction d'une relation avec SPPoC :

```
# let p = Presburger.parameters_of_list <:v<n, m>>;;
val p : SPPoC.Presburger.parameters = ["n";"m"]
# let r = Presburger.of_systems p <:v<i, j>> <:v<i', j'>>
                [<:s<i=i',1<=i<=n,1<=j<=m,j=10q+3,q>=0>>];;
val r : SPPoC.Presburger.t =
  { [i; j] -> [i'; j'],  Exists alpha :
    {i = i', j <= m, j >= 3, i' <= n, i' >= 1, 10*alpha-j = -3} }
```

Les paramètres sont traités à part (type Caml `Presburger.parameters`) car plusieurs relations peuvent partager les mêmes paramètres.

5.2. *Opérations ensemblistes*

Il est possible de représenter un polyèdre avec une relation ; il suffit de prendre un espace d'arrivée de dimension nulle. Par exemple le carré déclaré à l'aide de la PolyLib dans la section 4 peut aussi se définir comme une relation :

```
# let p' = Presburger.parameters_of_list <:v<n>>;;
val p' : SPPoC.Presburger.parameters = ["n"]
# let r = Presburger.of_systems p' <:v<i, j>> []
            [<:s<0<=i<=n, 0<=j<=n>>];;
val r : SPPoC.Presburger.t =
  { [i; j], {j <= n, j >= 0, i <= n, i >= 0 } }
```

On peut alors utiliser l'Omega Library pour effectuer des opérations ensemblistes comme le montre l'exemple ci-dessous.

```
# let r' = Presburger.of_systems p' <:v<i, j>> []
            [<:s< i<j >>];;
val r : SPPoC.Presburger.t = { [i; j], {i <= j} }
# let i = Presburger.inter r r';;
val i : SPPoC.Presburger.t = { [i; j], {i <= j, j <= n, i >= 0} }
```

Cette utilisation de l'Omega Library est redondante avec celle de la PolyLib mais on peut imaginer de lancer les deux calculs en parallèle et d'utiliser le résultat le plus rapidement obtenu ou le plus simple. L'implantation de ce procédé dans SPPoC pourra faire l'objet de futurs travaux. Les opérations ensemblistes s'appliquent aussi aux relations générales c'est à dire celles possédant un espace d'arrivée de dimension non nulle.

5.3. *Opérations spécifiques*

L'Omega Library permet d'aller bien au-delà des opérations ensemblistes, en particulier grâce à la possibilité d'introduire des variables quantifiées. Pour le moment, SPPoC ne permet pas d'utiliser toute la puissance de l'Omega Library. Cependant il est possible, dès maintenant, d'utiliser une fonctionnalité intéressante de l'Omega Library : le calcul de la fermeture transitive d'une relation. L'extrait de session SPPoC ci-dessous donne un exemple de calcul de la fermeture transitive d'une relation sur un domaine rectangulaire paramétrique.

```
# let r = Presburger.of_systems p <:v<i, j>> <:v<i', j'>>
            [<:s<i'=i+1; j'=j>>]
  and d = Presburger.of_systems p <:v<i, j>> []
            [<:s<1<=i<=n; 1<=j<=m>>]
  in Presburger.transitive_closure r d;;
- : SPPoC.Presburger.t = { [i; j] -> [i'; j'], {j = j', i-i' <= -1} }
```

L'intégration de l'Omega Library dans SPPoC est récente, un travail non négligeable reste à faire pour tirer parti de cette bibliothèque. Citons la possibilité d'introduire des variables quantifiées au sens de $\forall$, l'interfaçage de *toutes* les fonctions de manipulation de relations, l'ajout de conversions de ou vers les autres types majeurs de SPPoC, etc.

6. Simplifications symboliques

Nous présentons ici la plus forte valeur ajoutée de SPPoC, à savoir le système de calcul formel et en particulier les simplifications d'expressions et de systèmes de contraintes.

6.1. *Présentation générale de l'implémentation*

Le but poursuivi lors de l'écriture de SPPoC était d'offrir une interface de haut-niveau au calcul polyédrique. Les objectifs que nous nous sommes fixés sont donc les suivants :

– avoir un outil complètement symbolique ;

– proposer la même interface pour PIP, la PolyLib et l'Omega Library ;

– simplifier automatiquement les systèmes d'(in)équations.

La structure de donnée centrale est le système de contraintes affines qui permet à la fois d'exprimer les contraintes des problèmes de programmation entière et de décrire des polyèdres. Comme ces contraintes doivent être affines, la forme affine à coefficients rationnels, somme d'une constante rationnelle et de produits d'une variable symbolique par un rationnel, joue un rôle central.

Comme les données que nous manipulons ont une forme très particulière, l'utilisation d'un système de calcul formel tel que Maple [MAP] ou Mathematica [WOL] serait d'une lourdeur inutile. En fait, SPPoC peut-être vu comme un composant spécialisé pour le calcul polyédrique qui pourrait être intégré dans un tel système. Nous avons donc choisi d'écrire un système de simplification adapté qui fait apparaître le plus de formes affines possible.

La suite de la présentation de ce système de simplification formelle est organisée ainsi : nous présentons d'abord la simplification des expressions arithmétiques, puis celle des systèmes de contraintes. Enfin, nous verrons comment manipuler des QUAST, structure de donnée qui a été introduite en section 3.

6.2. *Expressions arithmétiques*

En ayant à l'esprit le souci d'une utilisation la plus aisée possible, nous ne restreignons pas la forme des expressions arithmétiques que l'utilisateur peut entrer. C'est le

système qui simplifie ces expressions en essayant de les linéariser au maximum. Une autre raison d'autoriser des expressions quelconques est qu'une sous-expression non linéaire peut être linéarisée plus tard par instanciation d'une variable ou simplification symbolique.

L'utilisateur peut par exemple entrer les expressions suivantes, qui sont automatiquement simplifiées.

```
# <:e<2*((i*m)^2) % 3>>;;
- : SPPoC.Expr.t = (2*((i*m)**2)) mod 3
# <:e<(n mod 2)*(2i+3*(j-i))-((3/3*n+0) mod 2)*(3j-i)>>;;
- : SPPoC.Expr.t = 0
```

Les opérateurs arithmétiques reconnus par l'algorithme sont : l'addition, la soustraction, la multiplication, la division rationnelle, l'élévation à la puissance, la division entière et le modulo (reste de la division entière). Les calculs numériques faisant intervenir ces opérateurs sont effectués en arithmétique rationnelle de précision arbitraire [5], les éléments neutres et absorbants sont traités, les formes affines sont développées et les expressions non linéaires factorisées. Comme pour tout algorithme de simplification où une forme canonique n'est pas définie (développée ou factorisée ?), le résultat n'est pas garanti mais il fonctionne plutôt bien sur les expressions quasi-linéaires que nous rencontrons dans nos applications.

L'heuristique de simplification parcourt l'arbre représentant l'expression arithmétique en partant des feuilles et en remontant vers la racine en appliquant les simplifications au fur et à mesure. Ce parcours de l'arbre garantit sa terminaison et une complexité linéaire dans la taille de l'expression (nombre de nœuds de l'arbre par exemple).

6.3. *Systèmes de contraintes*

Nous avons constaté expérimentalement, en particulier dans [BOU 98b], que les deux outils, PIP et la PolyLib, sont très sensibles à la forme de leur entrée. En effet, deux représentations équivalentes d'un même polyèdre peuvent mener à des résultats plus ou moins compliqués (bien qu'équivalents) ou encore faire échouer le calcul. En particulier, l'enchaînement de calculs, comme dans l'exemple de la section 7.2, peut être très difficile si les résultats intermédiaires ne sont pas simplifiées au fur et à mesure du calcul.

Pour permettre une simplification la plus poussée possible, nous avons classé les variables symboliques, appelées symboles dans la suite, en trois catégories :

– les « vraies » variables, c'est-à-dire sur lesquelles porte le calcul ;

5. utilisation de la bibliothèque `Num` de la distribution d'Objective Caml

– les variables intermédiaires qui ne servent qu'à exprimer le problème, elles sont en fait quantifiées existentiellement ;

– les paramètres du calcul.

L'algorithme de simplification va essayer de supprimer les variables intermédiaires et d'exprimer les variables en fonction des paramètres.

6.3.1. *Linéarisation*

Les systèmes de contraintes doivent être affines pour définir des polyèdres. Il faut donc linéariser ces systèmes. Afin de ne pas s'interdire des simplifications possibles, cette linéarisation est faite le plus tard possible. Elle consiste principalement à utiliser la définition de la division euclidienne (équation 3 ci-dessous) pour faire disparaître les divisions entières et les modulos en introduisant une variable intermédiaire supplémentaire.

$$\left\{\begin{array}{l} r \equiv a \mod b \\ q = a \div b \end{array}\right. \iff \left\{\begin{array}{l} \text{si } b > 0 \text{ alors } \left\{\begin{array}{l} a = bq + r \\ 0 \leq r < b \end{array}\right. \\ \text{si } b < 0 \text{ alors } \left\{\begin{array}{l} a = bq + r \\ b < r \leq 0 \end{array}\right. \end{array}\right. \tag{3}$$

Cette transformation n'est utile que si b est un entier non nul. En effet, dans le cas contraire, on obtient un système non linéaire.

6.3.2. *Simplifications élémentaires*

Nous décrivons ici les simplifications élémentaires faites par notre algorithme. L'enchaînement de ces transformations et la preuve de terminaison de l'algorithme sont expliqués dans la section suivante.

Toutes les transformations sont en fait des substitutions d'un symbole par une expression représentant sa valeur, ce qui va de la propagation des constantes à des substitutions plus complexes en passant par la suppression d'un symbole en cas de proportionnalité entre deux symboles. À chaque fois qu'une substitution est faite dans une expression, celle-ci est systématiquement simplifiée par l'algorithme présenté ci-dessus.

Ces remplacements de définitions de symboles conduisent à la simplification de l'exemple suivant, qui en est linéarisé :

```
# System.simplify <:v<i,j,k>> <:v<n,m>>
  <:s<n=10; k=n*i+j; j=m*n; i<=j>>;;
- : SPPoC.System.t = {j = 10*m, n = 10, i <= 10*m, 10*i+10*m = k}
```

La fonction *System.simplify* prend comme arguments la liste des variables, la liste des paramètres et le système à simplifier. Tous les symboles n'apparaissant dans aucune des listes arguments sont considérés comme des variables intermédiaires.

Toutes ces substitutions peuvent faire apparaître des inéquations inutiles, du genre définition d'une variable intermédiaire qui n'est jamais utilisée. Ces inéquations sont

supprimées. En modifiant l'exemple précédent pour faire apparaître une variable intermédiaire, on en constate la disparition après simplification :

```
# System.simplify <:v<i,j,k>> <:v<n,m>>
  <:s<n=10; k=n*i+j; x=m*n; i<=x+j>>;;
- : SPPoC.System.t = {n = 10, i <= j+10*m, 10*i+j = k}
```

6.3.3. *Enchaînement des simplifications*

L'algorithme d'enchaînement des simplifications est assez simple : tant qu'une simplification est possible, on fait la transformation correspondante. Ce schéma est légèrement modifié par un ordre de priorité donné aux transformations. On essaye en priorité les substitutions qui simplifient le plus le système, à savoir la propagation des constantes, ensuite la proportionnalité puis les substitutions de définitions et enfin la suppression d'inéquations inutiles. Pour détecter un maximum de définitions, c'est à dire d'expressions pouvant se mettre sous la forme « symbole = expression », les inéquations affines sont systématiquement simplifiées par le plus grand diviseur commun des coefficients de la forme affine. Cette heuristique nous semble efficace. En effet le résultat obtenu est au moins aussi simple que ce que nous obtenons en simplifiant « à la main » sur tous les cas que nous avons rencontrés.

Comme chaque simplification consiste en le remplacement d'un symbole par une expression ne faisant pas intervenir ce symbole, nous marquons ce symbole comme inactif dans la suite de l'algorithme. En effet, une nouvelle substitution de ce symbole n'apporterait rien. Le nombre de symboles dans un système étant fini, nous pouvons déduire que l'algorithme de simplification s'arrête et fait au plus n substitutions où n est le nombre de symboles apparaissant dans le système de contraintes.

6.4. *Calculs avec les QUAST*

Un QUAST, ou arbre de sélection quasi-affine, représente un point paramétré. La valeur de ce point dépend de celles des paramètres qui permettent de sélectionner une des feuilles de cet arbre de sélection. Les prédicats de sélection sont des inégalités affines et les feuilles des listes de formes affines, une pour chaque dimension de l'espace.

Cette structure de données permet de représenter les résultats des appels à PIP. Suite à la présence possible de *nouveaux paramètres* définis comme des modulos de formes linéaires dans un tel résultat, nous avons définis des « QUAST étendus » autorisant ces paramètres modulaires. Toutes les fonctions décrites ci-dessous s'appliquent aussi bien aux QUAST qu'aux QUAST étendus.

Outre les fonctions de création et de destruction des types abstraits QUAST et QUAST étendu (implémentés par les modules Quast et EQuast), nous avons défini des fonctions permettant de calculer avec de telles structures de données :

– la simplification d'un QUAST ;

– le calcul du minimum ou du maximum de deux QUAST ;

– la « mise à plat » un QUAST, c'est-à-dire la récupération une liste de couples (système de contraintes, valeur). Cette fonctionnalité est nécessaire pour l'application de la section 7.2.

La simplification se fait d'après la remarque suivante : suivre une branche de sélection dans un QUAST construit un système de contraintes. On peut donc éliminer les branches inutiles parce qu'inatteignables en testant l'existence d'un point dans le domaine défini par ces contraintes. Cela peut se faire en cherchant un point particulier dans ce domaine, par exemple le minimum lexicographique qui se calcule par un appel à PIP. On fusionne de même les branches qui, après élagage, sont identiques pour faire disparaître des nœuds inutiles dans l'arbre de sélection.

C'est lors des calculs d'extrema que la taille des QUAST manipulés peut augmenter dangereusement et que des simplifications sont particulièrement nécessaires. D'ailleurs, si lors d'un tel calcul des paramètres définissant des modulos interviennent avec des noms différents dans les deux QUAST, ils sont unifiés pour offrir plus de possibilités de simplification.

7. Exemples d'utilisation

Nous présentons dans ce chapitre deux exemples d'utilisation de SPPoC dans des contextes différents : la génération de code et l'estimation de volumes de communication.

Ces exemples illustrent différents aspects de SPPoC : le premier fait exclusivement appel à de la programmation linéaire et à des traitements sur les résultats fournis par PIP ; le deuxième est beaucoup plus complet et illustre la nécessité et l'efficacité des simplifications intermédiaires.

7.1. *Génération de code*

L'exemple que nous présentons ici illustre la partie de SPPoC qui permet la programmation linéaire en nombres entiers, à savoir l'interface à PIP et le calcul sur les QUAST. Le problème consiste à générer un code d'itération qui parcourt les points d'une union de domaines paramétrés. Ces domaines ainsi que l'ensemble de définition des paramètres sont définis par des ensembles de contraintes affines.

Comme expliqué dans [BOU 98a], résoudre ce genre de problème permet de générer du code après transformation d'un nid de boucles à contrôle statique par n'importe quelle suite de transformations affines (ordonnancements linéaires, affines, affines par morceaux, placements par projection, tuilage, torsion, etc). La méthode utilisée permet de traiter les nids de boucles non parfaitement imbriqués avec des transformations différentes pour chaque instruction.

7.1.1. *Modélisation*

L'idée de base consiste à ne pas essayer de reconstruire un nid de boucle, mais de construire un code à base de conditionnelles et de sauts, comme on peut les trouver dans le code assembleur provenant de la compilation de boucles. Il suffit en fait de calculer deux fonctions : l'une, first, qui donne le premier point de l'itération, c'est-à-dire le minimum lexicographique du domaine considéré ; et l'autre, next, qui calcule le point suivant le point courant. Ce point suivant est le minimum lexicographique de tous les points du domaine d'itération qui sont plus grands lexicographiquement que le point courant. Si le domaine d'itération est

$$\mathcal{L}(z) = \{x \in \mathbb{Z}^n \mid \exists y \in \mathbb{Z}^m, Ax + By + Cz \leq d\} \tag{4}$$

où m, n, p, $q \in \mathbb{N}^*$, $A \in \mathbb{Z}^p \times \mathbb{Z}^n$, $B \in \mathbb{Z}^p \times \mathbb{Z}^m$, $C \in \mathbb{Z}^p \times \mathbb{Z}^q$ et $d \in \mathbb{Z}^p$, avec z un vecteur de paramètres à valeurs dans un polyèdre $\mathcal{D} = \{z \in \mathbb{Z}^q \mid Ez \leq f\}$,alors le calcul de next revient à résoudre le problème :

trouver le x' minimum vérifiant les contraintes :

$$\left\{\begin{array}{l} z \in \mathcal{D} \\ x \in \mathcal{L}(z) \\ x' \in \mathcal{L}(z) \\ x \prec x' \end{array}\right. \iff \left\{\begin{array}{l} Ez \leq f \\ Ax + By + Cz \leq d \\ Ax' + By' + Cz \leq d \\ x \prec x' \end{array}\right. \tag{5}$$

où $\prec$ dénote l'ordre lexicographique.

La difficulté de cette résolution tient au fait que la contrainte $x \prec x'$ n'est pas linéaire. Pour la linéariser, on utilise la définition de l'ordre lexicographique pour décomposer cette contrainte en une disjonction :

$$\begin{array}{ll} & x_1 < x'_1 \\ \text{ou} & x_1 = x'_1,\ x_2 < x'_2 \\ & \vdots \\ \text{ou} & x_1 = x'_1,\ \ldots,\ x_{m-1} = x'_{m-1},\ x_m < x'_m \ . \end{array} \tag{6}$$

La recomposition des résultats des différents sous-problèmes linéaires est simplement le calcul du minimum de ces sous-problèmes. Or, comme ces différents résultats peuvent être ordonnés (conséquence directe de la décomposition de l'ordre lexicographique), nous pouvons utiliser ici la fonction `EQuast.group` qui optimise le calcul du minimum dans ce cas précis (voir section 6.4).

7.1.2. *Exemple de résolution*

Nous considérons un nid de boucles sur les indices $0 \leq i \leq n$ et $0 \leq j \leq i$ qui est transformé en un nid de boucles sur les indices i et $k = i + j$.

Nous avons deux problèmes linéaires à résoudre correspondants à la disjonction de l'ordre lexicographique. Définissons d'abord la partie commune des systèmes de contraintes :

```
# let pc = <:s<0<=n; 0<=i<=n; 0<=j<=i; k=i+j;
               0<=i'<=n; 0<=j'<=i'; k'=i'+j'>>;;
val pc : SPPoC.System.t =
  {n >= 0, i >= 0, i <= n, j >= 0, i >= j, i+j = k, i' >= 0, i' <= n,
   j' >= 0, i' >= j', i'+j' = k'}
```

On veut obtenir i' et k' en fonction de i et k. Pour cela, il nous faut résoudre en considérant que i', k', j' et j sont des variables et i, k et n des paramètres. Nous devons ensuite supprimer les dimensions j et j' inutiles. Ceci peut être fait en utilisant la question `Question.min_lex_exist <:v<i',k'>> <:v<j',j>>`. La construction des deux problèmes linéaires à résoudre se fait alors comme suit :

```
# let p1 = PIP.make (System.combine pc <:s<i<i'>>)
     (Question.min_lex_exist <:v<i',k'>> <:v<j',j>>);;
val p1 : SPPoC.PIP.t =
  Solve : MIN(i', k', j', j)
  Under the constraints :
  {n >= 0, i >= 0, i <= n, j >= 0, i >= j, i+j = k, i' >= 0, i' <= n,
   j' >= 0, i' >= j', i'+j' = k', i > i'}
# let p2 = PIP.make (System.combine pc <:s<i=i';k<k'>>)
     (Question.min_lex_exist <:v<i',k'>> <:v<j',j>>);;
val p2 : SPPoC.PIP.t =
  Solve : MIN(i', k', j', j)
  Under the constraints :
  {n >= 0, i >= 0, i <= n, j >= 0, i >= j, i+j = k, i' >= 0, i' <= n,
   j' >= 0, i' >= j', i'+j' = k', i = i', k > k'}
```

Nous calculons ci-dessous les deux résultats intermédiaires et nous les combinons pour obtenir le QUAST étendu représentant la fonction next :

```
# let r1 = PIP.solve p1;;
val r1 : SPPoC.EQuast.t = if i-n <= -1 then [1+i; 1+i] else _|_
# let r2 = PIP.solve p2;;
val r2 : SPPoC.EQuast.t = if 2*i-k >= 1 then [i; 1+k] else _|_
# let next = EQuast.group System.empty r1 r2;;
val next : SPPoC.EQuast.t =
  if i-n <= -1 then [1+i; 1+i] else if 2*i-k >= 1 then [i; 1+k] else _|_
```

La fonction first se calcule aisément comme suit :

```
# let first =
    PIP.solve (PIP.make <:s<0<=n; 0<=i<=n; 0<=j<=i; k=i+j>>
               (Question.min_lex_exist <:v<i,k>> <:v<j>>));;
val first : SPPoC.EQuast.t = [0; 0]
```

Nous pouvons maintenant en déduire un pseudo-code d'itération :

```
     i=0;
     k=0;
BOUCLE:
     -- corps de la boucle --
     if n-i >= 1 then
       i=i+1;
       k=i+1
     else
       if k <= 2*i-1
       then k=k+1
       else goto FIN
       endif
     endif;
     goto BOUCLE
FIN:
```

7.1.3. *Compléments*

Bien que nous ayons présenté cette génération de code en utilisant SPPoC de manière interactive, elle est intégrée à un prototype de paralléliseur automatique développé au PR*i*SM à l'université de Versailles. Ce schéma de résolution y a été étendu et optimisé. Les exemples complets présentés dans [BOU 98a] ont produit plusieurs dizaines d'appels à PIP à la minute et ont ainsi permis de vérifier la robustesse de l'implémentation.

7.2. *Estimation de volumes de communications*

Un autre exemple d'application utilisant SPPoC est le calcul d'une estimation du volume de communications générées par une instruction d'affectation dans un programme HPF. La méthode est présentée dans [BOU 98b], nous ne rappelons ici que les éléments nécessaires pour d'apprécier l'utilité de SPPoC pour ce genre de calcul.

7.2.1. *Un exemple pour fixer les idées*

Commençons par donner un exemple de programme HPF :

```
Program MatInit
!HPF$ PROCESSORS P(8,8)
!HPF$ TEMPLATE T(n,m)
!HPF$ DISTRIBUTE T(CYCLIC,CYCLIC) ONTO P
  real A(n,m), B(n)
!HPF$ ALIGN A(i,j) WITH T(i,*)
!HPF$ ALIGN B(i) WITH T(i,1)
  do i=1,n
    do j=1,m
```

```
      A(i,j)=B(i)              (S1)
    end do
  end do
end
```

Ce programme se contente d'initialiser une matrice en recopiant un vecteur dans chacune de ses colonnes. Nous allons chercher à estimer le volume des communications générées par l'instruction `S1`. À ce propos on constate que `S1` est une affectation ne concernant qu'un seul tableau en lecture. Nous nous limiterons ici à de telles affectations. Il est facile de généraliser par simple addition de volumes au prix d'une sur-estimation en cas de références multiples en lecture au même tableau. La figure 1 représente le flot des données de l'exemple et la figure 2 indique comment le vecteur et le tableau sont alignés sur le *template*[6] `T`.

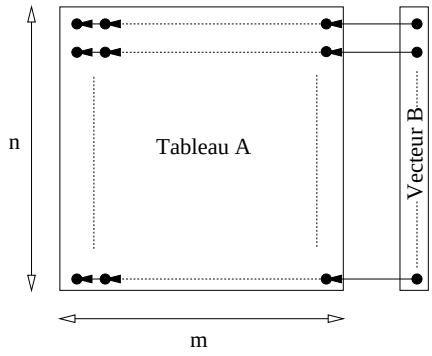


Figure 1. *Flot des données*

La figure 2 montre que les alignements HPF peuvent être utilisés pour répliquer les données sur le template et donc sur les processeurs au moyen de « `*` » dans les directives. Chaque colonne du tableau A est ici repliquée sur chaque colonne du template T. Une évaluation du volume de communication doit donc prendre en compte cette particularité. C'est pourquoi nous comptons les communications au niveau du template : il y a communication entre deux éléments de template s'il ne sont pas distribués sur le même processeur (au sens de la directive `PROCESSORS`) et si le calcul d'une valeur alignée sur le premier élément nécessite une valeur alignée sur le second. Nous définissons le volume de communications générées par une instruction d'affectation comme le nombre de communications entre éléments de template générées par cette instruction.

Des techniques de compilation telles que la vectorisation ou la factorisation des communications permettent d'éviter certaines des communications que nous prenons

6. Un *template* est un tableau virtuel sur lequel sont alignés les tableaux de données par des directives `ALIGN` et qui est réparti sur les processeurs par une directive `DISTRIBUTE`.

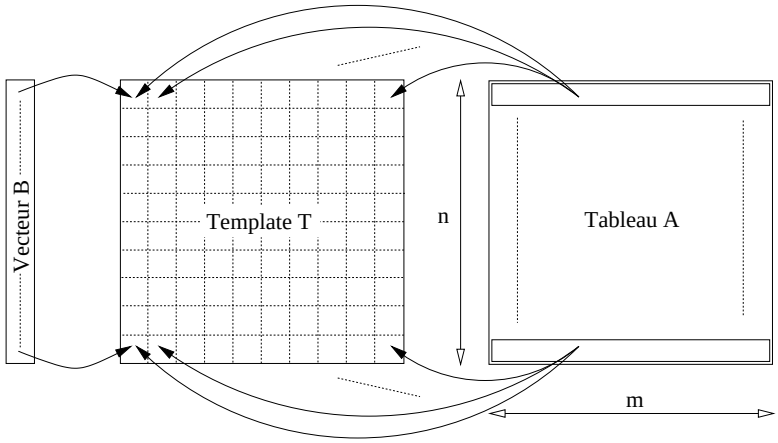


Figure 2. *Distribution des objets*

en compte. Cependant notre but est de donner une estimation pour un programme donné indépendamment du compilateur ou de la machine utilisée. Grâce à cette estimation il est possible de choisir une implantation d'algorithme plutôt qu'une autre.

7.2.2. *Représentation des alignements et des distributions HPF*

Pour donner une formule précise du volume de communications il faut pouvoir modéliser formellement les alignements et les distributions HPF. Un alignement HPF peut se concevoir comme la composition de l'inverse d'une fonction affine avec d'une fonction affine. La fonction inverse permet de formaliser l'éventuelle réplication de données.

Ainsi l'alignement du tableau `A` de notre exemple se modélise par la composition de l'inverse de la fonction $\delta_A \binom{i_1}{i_2} = i_1$ et de la fonction $\gamma_A \binom{i_1}{i_2} = i_1$. L'alignement du tableau `B` est lui modélisé par la composition de l'inverse de la fonction $\delta_B \binom{i_1}{i_2} = \binom{i_1}{i_2}$ et de la fonction $\gamma_B (i_1) = \binom{i_1}{1}$.

La modélisation d'une distribution HPF se base sur une projection ρ_T qui permet de sélectionner les dimensions du template qui sont distribuées sur les processeurs et sur un vecteur de paramètres κ_T. Les dimensions du template sont distribuées suivant le schéma `CYCLIC(k)` [7], le paramètre `k` étant donné, pour chaque dimension, par le vecteur de paramètres. Il est possible d'obtenir une distribution par blocs en choisissant correctement le paramètre `k`. Si les bornes inférieures et supérieures du template et du tableau de processeurs sont notées T_{min}, T_{max}, P_{min} et P_{max}, la fonction don-

7. `k` vaut 1 par défaut, ce qui est le cas dans notre exemple.

nant les coordonnées du processeur sur lequel est distribué un élément de template donné est

$$\pi_T(J) = P_{min} + (\rho_T(J - T_{min}) \div \kappa_T) \% (P_{max} - P_{min} + \mathbf{1}) \ . \tag{7}$$

Dans le cas de notre exemple, la fonction de projection est $\rho_T\binom{j_1}{j_2} = \binom{j_1}{j_2}$ et le vecteur de paramètres $\kappa_T = \binom{1}{1}$. La fonction de distribution est donc

$$\begin{aligned} \pi_T \begin{pmatrix} j_1 \\ j_2 \end{pmatrix} &= \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{pmatrix} j_1 - 1 \\ j_2 - 1 \end{pmatrix} \div \begin{pmatrix} 1 \\ 1 \end{pmatrix} \% \left(\begin{pmatrix} 8 \\ 8 \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \mathbf{1} \right) \\ &= \begin{pmatrix} 1 + (j_1 - 1)\%8 \\ 1 + (j_2 - 1)\%8 \end{pmatrix} \ . \end{aligned} \tag{8}$$

7.2.3. *Formule d'estimation du volume de communications*

Pour préciser la formule de calcul du volume de communications telle que définie dans le paragraphe 7.2.1, nous introduisons quelques notations :

– le domaine d'itération englobant l'instruction d'affectation est appelé $\mathcal{D}$ et le domaine de définition du template T est appelé $\mathcal{D}_T$;

– les fonctions d'accès aux éléments de template Φ_E (pour le tableau en écriture) et Φ_L (pour le tableau en lecture) sont définies comme la composition des fonctions d'alignement et des fonctions d'accès de ces tableaux ;

– la fonction π_T est, bien entendu, la fonction de distribution du template T.

Le volume de communications est alors donné par le nombre d'éléments de l'ensemble :

$$\left\{ (I, J) \mid J \in \mathcal{D}_T,\ I \in \Phi_E^{-1}(J),\ \forall K \in \Phi_L(I),\ \pi_T(J) \neq \pi_T(K) \right\} \tag{9}$$

7.2.4. *Utilisation de SPPoC pour le calcul de volume*

Notre but est de calculer de façon *automatique* le cardinal de l'ensemble décrit par la formule 9. Le présent paragraphe montre comment SPPoC peut être utilisé pour effectuer les différentes étapes de ce calcul.

Des opérateurs infixes sont offerts pour manipuler des vecteurs de formes linéaires. Ce sont les opérateurs arithmétiques et de comparaison classiques préfixés par le symbole `|`. Avec ces opérateurs, la construction des domaines de définition est très facile comme le montre cet extrait de code :

```
...
let dt = Polyhedron.of_system
           ( System.make ( ( j |>= t_min ) @ ( j |<= t_max ) )
...
```

Les contraintes d'appartenance comme $I \in \Phi_E^{-1}(J)$ ne sont pas exprimables directement par SPPoC. Par contre, il est possible de construire l'ensemble des (I, J)

respectant une telle contrainte. Pour cela, il suffit de prendre un ensemble paramétrique de paramètre J défini par la contrainte $I = J$ et de calculer une succession d'images et de pré-images de cet ensemble par les fonctions composant Φ_E (les deux fonctions composant l'alignement du tableau en écriture et la fonction d'accès au tableau en écriture). Voici un extrait de code de notre prototype de calcul de communications qui implante ce procédé :

```
...
Polyhedron.inter
  domaine_iteration
  ( Polyhedron.preimage
     ( Polyhedron.preimage
        ( Polydedron.image ensemble_parametrique delta_E )
        gamma_E )
     phi_E )
...
```

La contrainte $K \in \Phi_L(I)$ se traite de manière analogue.

La fonction de distribution est, elle aussi, implantée sous la forme d'un système d'équations. Le plus complexe est le calcul du résultat de la projection `rho` et du vecteur de paramètres `kappa`, SPPoC fait le reste :

```
...
let taille = p_max |- p_min |+ one in
  System.make ( p |= ( p_min |+ (( rho |/ kappa ) |% taille )))
...
```

Ce système est dupliqué pour pouvoir être appliqué sur J et sur K, l'égalité entre les deux fonctions de distribution est assurée par l'utilisation de variables intermédiaires représentant les coordonnées des processeurs (vecteur `p`).

Il faut aussi régler le problème de l'opérateur $\forall$ qui apparait dans (9). En fait il est préférable, plutôt que de calculer le nombre d'éléments de template donnant lieu à une communication, de calculer le nombre d'éléments de template ne donnant lieu *aucune* communication. Calculer le nombre d'éléments de l'ensemble (9) revient donc à calculer le nombre d'éléments de

$$\{(I, J) \mid I \in \mathcal{D},\ J \in \mathcal{D}_T\} \tag{10}$$

auquel on soustrait le nombre d'éléments de

$$\left\{(I, J) \mid J \in \mathcal{D}_T,\ I \in \Phi_E^{-1}(J),\ \exists K \in \Phi_L(I),\ \pi_T(J) = \pi_T(K)\right\} \tag{11}$$

Pour modéliser ce dernier ensemble, il suffit de considérer les variables liées à K comme des variables intermédiaires.

En définitive, l'ensemble (11) s'obtient par calcul de l'intersection des ensembles présentés plus haut. Le système résultat comporte des divisions entières et des modulos que SPPoC ne linéarise qu'en dernier recours. De cette façon, aucune information

n'est perdue ce qui permet des simplifications plus efficaces que dans un système linéarisé.

Dans le cas de notre exemple `MatInit`, la session SPPoC ci-dessous montre à la fois le système correspondant à l'ensemble (11) et la façon dont il est simplifié. Pour s'y reconnaître il faut savoir que lors de la génération automatique du système les notations suivantes ont été utilisées :

– les composantes de I sont `i1` et `i2` ;

– les composantes de J sont `j1` et `j2` ;

– les composantes de K sont `k1` et `k2` ;

– les composantes du vecteur des processeurs sont `p1` et `p2` ;

– et enfin les paramètres du programme HPF sont `n` et `m`.

```
# System.simplify
    <:v<i1,i2,j1,j2>> (* Variables *)
    <:v<n,m>>                (* Paramètres *)
    <:s< (-1+p1)+(-((-1+j1) mod 8)) = 0,
         (-1+p2)+(-((-1+j2) mod 8)) = 0,
         (-1+p1)+(-((-1+k1) mod 8)) = 0,
         (-1+p2)+(-((-1+k2) mod 8)) = 0,
         i1 = j1, k1 = j1, k2 = 1,
         m >= j2, j2 >= 1, i2 >= 1,
         n >= j1, m >= i2, j1 >= 1 >> ;;
- : SPPoC.System.t =
{i1 = j1, (-1+j2) mod 8 = 0, m >= j2, j2 >= 1, i2 >= 1,
 n >= j1, m >= i2, j1 >= 1}
```

Dans le cas général, il n'est pas possible de compter directement le nombre de points entiers dans le système simplifié par SPPoC. En effet, celui-ci peut encore comporter des variables intermédiaires utilisées pour modéliser le problème (les composantes de K et de `p`). Il ne faut compter que les couples (I, J) pour lesquels il existe un K et un `p` vérifiant les contraintes de (11). Cela peut être réalisé par un appel à PIP pour résoudre le programme linéaire de minimisation (ou de maximisation cela n'a pas d'importance) du vecteur (K,`p`) sous les contraintes de (11). Il s'agit d'une résolution paramétrique en fonction des paramètres I, J et des paramètres N du programme HPF. Dans le résultat de PIP seuls importent les domaines des paramètres I, J et N pour lesquels existent des solutions au programme linéaire. Le résultat recherché est la somme des nombres de points entiers dans ces domaines. Il est évident qu'il faut obtenir le nombre de points entiers en fonction des paramètres N.

Ce nombre de points est calculé via la fonction `Polyhedron.enumeration` de SPPoC. Pour notre exemple, il est inutile d'appeler PIP puisque la simplification opérée par SPPoC a déjà éliminé les variables intermédiaires. Le résultat du calcul de

la différence entre le nombre de points entiers dans les ensembles (10) et (11) est le suivant :

```
[{({m >= 1, n >= 1},
   {ray(n), ray(m), vertex(m+n)})},
 7/8*n*m**2-m*n*[0, 7/8, 3/4, 5/8, 1/2, 3/8, 1/4, 1/8]m]
```

Ce résultat indique qu'il n'existe de communication que si $n \geq 1$ et $m \geq 1$ (dans le cas contraire l'instruction ne génère aucune opération, il n'y a donc effectivement pas de communication). Il est possible de simplifier le résultat en supposant que m est divisible par 8, dans ce cas le coefficient périodique vaut zéro. Le nombre de communications est alors $\frac{7}{8}nm^2$, résultat auquel on pouvait s'attendre au vu de la figure 2.

8. Conclusion

Nous avons présenté ici SPPoC, un outil de calcul polyédrique et de programmation linéaire qui permet l'utilisation à travers une interface unifiée de PIP, de la PolyLib et de l'Omega Library. Plus qu'une simple interface, SPPoC offre des fonctionnalités supplémentaires : prise en charge des changements de variables lors des appels à PIP, amélioration de certaines fonctions de la PolyLib (gestion automatique des dimensions et comptage de points dans un produit cartésien), et un moteur de simplification d'expressions arithmétiques et de systèmes de contraintes.

Le premier objectif de SPPoC est atteint : permettre l'implantation des deux applications présentées dans cet article. Pour bien comprendre le chemin parcouru, il faut savoir qu'une tentative de calcul du volume des communications sur l'exemple donné dans la section 7.2 a été faite en utilisant directement PIP et la PolyLib. Dans un premier temps, aucun résultat n'a pu être obtenu (mémoire insuffisante, temps de calcul prohibitif). Dans un second temps, après découpage manuel du problème, un résultat est finalement sorti. Le volume calculé tenait sur deux pages de texte et était, bien entendu, inexploitable.

Le second objectif est de faire en sorte que SPPoC puisse être plus facilement utilisable pour les autres applications. Une première version est téléchargeable dès maintenant à partir de la page web `http://www.lifl.fr/west/sppoc/`, mais des améliorations sont nécessaires pour une plus grande diffusion. Le paquetage doit être perfectionné en supprimant de parties de code obsolète, en augmentant l'indépendance par rapport aux versions de PIP et de la PolyLib et en terminant l'interface à l'Omega Library. Il faut aussi ajouter des fonctionnalitées et, bien sûr, continuer la chasse aux bogues.

Le domaine d'application de SPPoC est très large puisqu'il couvre tout le champ des optimisations de compilation ou de parallélisation basées sur des analyses statiques de code. SPPoC permettant une plus grande facilité d'accès aux bibliothèques de calcul polyédrique, nous pensons que des analyses plus complexes pourront être menées à bien et que le prototypage de nouvelles applications sera plus rapide. No-

tons en particulier qu'il est prévu d'utiliser SPPoC dans le cadre d'une collaboration entre le LIFL et le PRiSM autour du thème de l'analyse de code pour vérifier l'équivalence de programmes.

9. Bibliographie

[BOU 98a] BOULET P., FEAUTRIER P., « Scanning polyhedra without Do-loops », *PACT'98*, IEEE Computer Society, 1998, p. 4-11.

[BOU 98b] BOULET P., REDON X., « Communication Pre-evaluation in HPF », *EURO-PAR'98*, vol. 1470 de *LNCS*, Springer Verlag, 1998, p. 263-272.

[CLA 96a] CLAUSS P., « Counting Solutions to Linear and Nonlinear Constraints through Ehrhart polynomials: Applications to Analyze and Transform Scientific Programs », *ACM Int. Conf. on Supercomputing*, ACM, 1996.

[CLA 96b] CLAUSS P., LOECHNER V., « Parametric Analysis of Polyhedral Iteration Spaces », *IEEE Int. Conf. on Application Specific Array Processors, ASAP'96*, IEEE Computer Society, 1996.

[CLA 97] CLAUSS P., LOECHNER V., WILDE D. K., « Deriving Formulae to Count Solutions to Parameterized Linear Systems using Ehrhart Polynomials: Applications to the Analysis of Nested-Loop Programs », rapport n° RR 97-05, April 1997, Laboratoire Image et Calcul Parallèle Scientifique, URL: `http://icps.u-strasbg.fr/PolyLib`.

[COL 95] COLLARD J.-F., FEAUTRIER P., RISSET T., « Construction of DO loops from systems of affine constraints », *Parallel Processing Letters*, vol. 5, n° 3, 1995, p. 421-436.

[DAR 94] DARTE A., ROBERT Y., « Constructive methods for scheduling uniform loop nests », *IEEE Trans. Parallel Distributed Systems*, vol. 5, n° 8, 1994, p. 814-822.

[DAR 95] DARTE A., ROBERT Y., « Affine-by-statement scheduling of uniform and affine loop nests over parametric domains », *J. Parallel and Distributed Computing*, vol. 29, 1995, p. 43-59.

[DIO 95] DION M., ROBERT Y., « Mapping Affine Loop Nests: New Results », HERTZBERGER B., SERAZZI G., Eds., *High-Performance Computing and Networking, International Conference and Exhibition*, vol. LCNS 919, Springer-Verlag, 1995, p. 184-189, Extended version available as Technical Report 94-30, LIP, ENS Lyon (anonymous ftp to lip.ens-lyon.fr).

[FEA 88] FEAUTRIER P., « Parametric Integer Programming », *RAIRO Recherche Opérationnelle*, vol. 22, 1988, p. 243–268, URL: `http://www.prism.uvsq.fr/parallel/softs/`.

[FEA 90] FEAUTRIER P., TAWBI N., « Résolution de Systèmes d'Inéquations Linéaires; mode d'emploi du logiciel PIP », rapport n° 90-2, 1990, Institut Blaise Pascal, Laboratoire MASI (Paris).

[FEA 91] FEAUTRIER P., « Dataflow analysis of array and scalar references », *Int. J. Parallel Programming*, vol. 20, n° 1, 1991, p. 23-51.

[FEA 92a] FEAUTRIER P., « Some efficient solutions to the affine scheduling problem, part I: one-dimensional time », *Int. J. Parallel Programming*, vol. 21, n° 5, 1992, p. 313–348.

[FEA 92b] FEAUTRIER P., « Some efficient solutions to the affine scheduling problem, part II: multi-dimensional time », *Int. J. Parallel Programming*, vol. 21, n° 6, 1992, p. 389–420.

[FEA 94] FEAUTRIER P., « Towards automatic distribution », *Parallel Processing Letters*, vol. 4, n° 3, 1994, p. 233-244.

[KEL 95] KELLY W., PUGH W., ROSSER E., « Code generation for multiple mappings », *The 5th Symposium on Frontiers of Massively Parallel Computation*, McLean, Virginia, 1995, p. 332-341.

[KEL 96] KELLY W., MASLOV V., PUGH W., ROSSER E., SHPEISMAN T., WONNACOTT D., « The Omega Library Interface Guide », rapport, 1996, Dept. of Computer Science, Univ. of Maryland, College Park, URL: `http://www.cs.umd.edu/projects/omega/`.

[MAP] MAPLESOFT, « Maple », URL: `http://www.maplesoft.com`.

[PRO] PROJET CRISTAL, « The Caml language », URL: `http://caml.inria.fr/`.

[PUG 91] PUGH W., « The Omega test: a fast and practical integer programming algorithm for dependence analysis », IEEE, Ed., *Proceedings, Supercomputing '91: Albuquerque, New Mexico, November 18–22, 1991*, IEEE Computer Society Press, 1991, p. 4–13.

[PUG 92] PUGH W., « A practical algorithm for exact array dependence analysis », *Communications of the ACM*, vol. 8, 1992, p. 27-47.

[RAU] DE RAUGLAUDRE D., « Camlp4 », URL: `http://caml.inria.fr/camlp4`.

[WIL 93] WILDE D. K., « A Library for Doing Polyhedral Operations », rapport n° Internal Publication 785, Dec 1993, IRISA, Rennes, France, Also published as INRIA Research Report 2157.

[WOL] WOLFRAM RESEARCH, « Mathematica », URL: `http://www.mathematica.com`.

[WOL 91] WOLF M. E., LAM M. S., « A loop transformation theory and an algorithm to maximize parallelism », *IEEE Trans. Parallel Distributed Systems*, vol. 2, n° 4, 1991, p. 452-471.

Article reçu le 6 décembre 1999.
Version révisée le 4 août 2000.
Rédacteur responsable : Catherine Mongenet

Pierre Boulet *est docteur en informatique de l'École Normale Supérieure de Lyon. Il est actuellement maître de conférences au Laboratoire d'Informatique Fondamentale de Lille. Ses travaux portent toujours sur la parallélisation automatique et la compilation de langages à parallélisme de données. Ils évoluent cependant vers les environnements de programmation parallèle visuelle et le meta-computing.*

Xavier Redon *est diplomé de l'école d'ingénieurs IIE (Institut d'Informatique d'Entreprise) d'Evry et docteur en informatique de l'Université de Versailles. Il est actuellement maître de conférences au Laboratoire d'Informatique Fondamentale de Lille et enseigne à l'EUDIL (Ecole Universitaire D'Ingénieurs de Lille). Il continue à travailler sur les outils utilisés dans la parallélisation automatique comme SPPoC mais se tourne plus vers l'analyse de programmes impératifs dans d'autres buts comme la preuve automatique d'équivalence d'algorithmes.*

Annexe B

Scanning Polyhedra without Do-Loops

Scanning polyhedra without Do-loops

Pierre BOULET
LIFL, USTL
Bâtiment M3, Cité scientifique
59655 Villeneuve d'Ascq Cedex, France
Pierre.Boulet@lifl.fr

Paul FEAUTRIER
Laboratoire PR*i*SM
Université de Versailles-St Quentin en Yvelines
Bâtiment Descartes, 45, av. des États-Unis
78035 Versailles Cedex, France
Paul.Feautrier@prism.uvsq.fr

Abstract

We study in this paper the problem of polyhedron scanning which appears for example when generating code for transformed loop nests in automatic parallelization. After a review of related works, we detail our method to scan affine images of polyhedra. After some experimental results we show how our method applies to unions of affine images of polyhedra.

We have taken the option to generate low level code without loops. This has allowed us to have a completely general and fully parameterized method without losing efficiency.

1. Introduction

In the field of automatic parallelization, efforts have been focused on parallelizing loops because they concentrate most of the computing time in a small number of statements. The polytope model has been devised in order to fully analyze loops. In this model, the iteration domain (the set of operations) of a statement is described as a parameterized polyhedron. Each operation is associated to an iteration vector whose components are the values of the surrounding loop counters. Translating the loop bounds into inequalities gives a polyhedron which the iteration vector must belong to.

In order to exhibit parallelism, one applies transformations to the iteration space. These transformations are usually deduced from attempts to optimize the efficiency of the transformed program. Scheduling, for instance, aims at minimizing the running time on a parallel computer, while mapping aims at minimizing communications. All in all, these transformations usually are affine transformations. They map the original iteration domain of each statement to a new iteration domain defined as the set of images of the points of the original domain. We will consider here only integral affine transformations, which means that the image domain is a set of integer points.

The program which results from a given transformation is obtained by writing code for enumerating, or scanning in a given order the resulting iteration domains. This problem has been the subject of a large number of papers, starting with Irigoin's thesis [9] and the seminal paper [1]. The order in which the points are enumerated is relevant since one of the constraints which are imposed on the transformations is that the image set must be enumerated in lexicographic order. In fact some dimensions of the image iteration space are time dimensions which are sequential while some others are space dimensions which are parallel. The originality of our method is that we do not try to generate a new loop nest to be compiled later, but rather to directly generate low level code. This gives us more freedom in the structure of the generated code while retaining good efficiency. Observe that the usual invective against `GOTO`'s does not apply in our case: the code we generate is not for human consumption. It is just an intermediate representation for the use of a compiler backend.

The organization of the paper is as follows: in Sect. 2, we present previous solutions to the same problem; in Sect. 3 we study the case of a single linearly bounded lattice, which we illustrate with some experiments in Sect. 4. We then show in section 5 how the method presented before can be extended to handle the general case of the union of linearly bounded lattices and we conclude in Sect. 6.

2. Related work

The problem of scanning polyhedra first arose in relation with the loop inversion transform. While it is evident that the "inverse" of

```
do i = 1,n          do j = 1,m
  do j = 1,m    is    do i = 1,n
    S                   S
  end do              end do
end do              end do
```

it requires some thought to see that the inverse of

```
do i = 1,n          do j = 1,n
  do j = i,n    is    do i = 1,j
    S                   S
  end do              end do
end do              end do
```

This kind of problem occurs either when the polyhedron to be scanned is given without any reference to a loop nest (for instance when one use specification languages like ALPHA) or when the loop nest is submitted to a unimodular transform. This situation is characterized by the fact that *all* integer points in the given polyhedron are to be visited. In this form, the problem was first solved by Irigoin in [9]; see also [8, 1, 5].

However, not all parallelizing transformations are unimodular. They may even be singular: this situation occurs when constructing communication loops [11]. The solution is to write the transformation $T = HU$ where U is unimodular and H is a Hermite normal form of T. One first scans the image of the iteration space by U, as above, then apply H, which has the property of being monotone, increasing with respect to lexicographic order. See [6, 12, 13].

In the general case, there are several statements which may be subjected to different transformations. One has to scan the union of the images of several polyhedra, not necessarily of the same dimension. All solutions which have been proposed [3, 10, 4] are compromises between code size and performance, with no clear way of selecting an optimum.

In all cases, the result is obtained by combining several algorithms: Hermite normal form construction, Fourier-Motzkin elimination, various methods for splitting domains. Our aim here is to give just one algorithm for handling all cases. In the interest of clarity, we will nevertheless split the presentation in two parts: firstly, the case of a linearly bounded lattice (LBL), and secondly the case of a union of linearly bounded lattices. Scanning a polyhedron is just a particular case of scanning an LBL.

3. Scanning Linearly Bounded Lattices

We first describe in this section the problem we consider (see section 3.1), then the algorithm used to solve it (see section 3.2) and finally the code generation scheme used by this algorithm (see section 3.3).

3.1. Problem specification

We consider here the problem of scanning a *linearly bounded lattice* in *lexicographic order*.

Definition 3.1 (Linearly Bounded Lattice). A linearly bounded lattice $\mathcal{L}$ is a set of integer points verifying a system of affine inequalities as follows:

$$\begin{aligned}&\mathcal{L} = \{x \in \mathbb{Z}^n \mid \exists y \in \mathbb{Z}^m, Ax + By \leq c\}\\ &\text{where } n, m, p \in \mathbb{N}^*, A \in \mathbb{Z}^p \times \mathbb{Z}^n,\\ &\qquad B \in \mathbb{Z}^p \times \mathbb{Z}^m \text{ and } c \in \mathbb{Z}^p \ .\end{aligned}$$

We will in fact handle parameterized linearly bounded lattices:

$$\begin{aligned}&\mathcal{L}(z) = \{x \in \mathbb{Z}^n \mid \exists y \in \mathbb{Z}^m, Ax + By + Cz \leq d\}\\ &\text{where } n, m, p, q \in \mathbb{N}^*, A \in \mathbb{Z}^p \times \mathbb{Z}^n,\\ &\qquad B \in \mathbb{Z}^p \times \mathbb{Z}^m, C \in \mathbb{Z}^p \times \mathbb{Z}^q \text{ and } d \in \mathbb{Z}^p \ .\end{aligned}$$

Here z is a vector of parameters whose values are in a polyhedron $\mathcal{D}$ defined as:

$$\mathcal{D} = \{z \in \mathbb{Z}^q \mid Ez \leq f\} \ .$$

In the following, all the LBLs we will encounter will be parameterized LBLs.

Definition 3.2 (lexicographic order).
The lexicographic order $\prec$ over $\mathbb{Z}^m$ can be expressed as:

$$\begin{aligned}&(x_1, \ldots, x_m) \prec (y_1, \ldots, y_m) \iff\\ &\exists k \in \{0, \ldots, m-1\}\\ &\qquad \mid x_1 = y_1, \ldots, x_k = y_k, x_{k+1} < y_{k+1} \ .\end{aligned}$$

We note $\min_\prec$ the lexicographic minimum.

3.2. Resolution method

The basic idea for the enumeration of the LBL $\mathcal{L}(z)$ is to build a function, "next", which, given a point in $\mathcal{L}(z)$, returns the next higher point in $\mathcal{L}(z)$ according to lexicographic order.

To initialize this process, we have to build a constant, "first", which is the lexicographic minimum of $\mathcal{L}(z)$. "first" is defined as:

$$\text{first} = \min_\prec \{y \in \mathcal{L}(z)\} \ .$$

Note. This kind of problem is a parameterized linear program that can be solved using a tool such as PIP [7]. PIP computes lexicographic minima of domains defined by integer linear inequations. The solutions PIP returns are in the form of a *quasi-affine selection tree* (QUAST).

Indeed, depending on the values of the parameters, a solution may have different values. It has been shown that these values can be expressed as linear expressions of the parameters in polyhedral subdomains. Thus the QUAST structure is a selection (`if then else`) tree describing these subdomains and the associated solutions. Two branches of this tree are separated by a linear predicate defining an hyperplane, thus delimiting two subdomains of the parameter domain. The leaves of this tree are linear expressions of the searched minimum. We note $\perp$ when there is no solution for a given subdomain. Some problems may require the presence of some integer modulos at some depth in the selection tree. These modulos add complexity to the generated code.

Once we know how to compute "first", "next" can be described in the following way:

$$\begin{aligned}\mathrm{next} : \mathcal{L}(z) &\to \mathcal{L}(z) \cup \{\perp\} \\ x &\mapsto \min_{\prec}\{y \in \mathcal{L}(z) \mid x \prec y\} \ .\end{aligned}$$

So, we have to solve the following problem:

find the minimum x' subject to the constraints:

$$\left\{\begin{array}{l} z \in \mathcal{D} \\ x \in \mathcal{L}(z) \\ x' \in \mathcal{L}(z) \\ x \prec x' \end{array}\right. \iff \left\{\begin{array}{l} Ez \leq f \\ Ax + By + Cz \leq d \\ Ax' + By' + Cz \leq d \\ x \prec x' \end{array}\right.$$

This problem is non-linear due to the constraint $x \prec x'$. We have to decompose it into smaller linear problems and to compose their results. To build these smaller problems, we use definition 3.2 to decompose the constraint $x \prec x'$ into the disjunction:

$$\begin{array}{ll} & x_1 < x'_1 \\ \text{or} & x_1 = x'_1, x_2 < x'_2 \\ & \vdots \\ \text{or} & x_1 = x'_1, \ldots, x_{m-1} = x'_{m-1}, x_m < x'_m \end{array}$$

We can now build the linear subproblems $\mathcal{P}^1, \ldots, \mathcal{P}^m$. Here is the general form of $\mathcal{P}^k$:

minimize x' subject to the constraints:

$$\left\{\begin{array}{l} Ez \leq f \\ Ax + By + Cz \leq d \\ Ax' + By' + Cz \leq d \\ x_1 = x'_1 \\ \vdots \\ x_{k-1} = x'_{k-1} \\ x_k < x'_k \end{array}\right.$$

Using the results of these parameterized linear problems, we can now generate the iteration code.

The abstract program looks like:

```
    x = first
1   if x = ⊥ then goto 2
    loop body
    x = next(x)
    goto 1
2
```

This prescription is sufficient when one just has to write sequential code, as for instance scatter/gather code in communication routines. But one may wonder how parallelism will be expressed in this framework. In the standard scheme, those loops whose counter is not one of the components of time are flagged as parallel. One then relies on a low level parallel compiler to write the actual parallel code.

In our context, the solution is to select some of the variables as virtual processor names, and to scan the remaining variables with the virtual processor names as parameters. This will naturally generate SPMD code. If p is the virtual processor name, and if we want to use blocking in the virtual processor space, we just add the constraints: $a \leq p \leq b$, a and b being new parameters. Our system then automatically writes the virtualization loop.

The next step is to insert synchronization primitives. In the case of a distributed memory machine, synchronization is a byproduct of message passing. For a global memory machine, one has to use a new synchronization primitive: `synch(t)`.

Each processor has its own virtual clock in shared memory. The `synch` primitive first sets this clock to `t`, then enters a busy waiting loop. At each iteration, the processor computes (redundantly) the minimum of all clocks. It leaves the waiting loop iff its clock is equal to the said minimum.

Note. The above is just a definition of the semantics of `synch`. Its performance can be enhanced in various ways: relinquishing the processor in coarse grained situations, logarithmic update of the minimum, and others.

One may extend this definition to multidimensional time. This being done, one just has to insert a `synch` in the above code at each point where a component of time is modified. Since the `synch` primitive has the ability of "jumping ahead" in time, our parallel program will be at least as efficient as an implementation with barriers, in which there are as many synchronization operations as there are time steps.

3.3. Code generation algorithm

Let us now consider the iteration code. Ideally, this code should be as efficient as a loop would be, in the case when a loop can be designed to iterate over the same set of points $\mathcal{L}(z)$.

First we can observe that any solution of the problem $\mathcal{P}^k$ is lexicographically greater than any solution of the subproblem $\mathcal{P}^l$ when $k < l$. This comes from the definition of these problems. Indeed, in any solutions x'^k of $\mathcal{P}^k$ and x'^l of $\mathcal{P}^l$, $\forall i < k, x'^k_i = x_i$, $x'^k_k > x_k$ and $\forall i < l, x'^l_i = x_i$. As $k < l$, $\forall i < k, x'^k_i = x'^l_i$ and $x'^k_k > x'^l_k$, which means $x'^k \succ x'^l$. This allows us to seek the next point in the domain by first looking for solutions of $\mathcal{P}^m$, then, if there is no solution, of $\mathcal{P}^{m-1}$, and so on until $\mathcal{P}^1$. If this last problem has no solution, then we have finished scanning the set $\mathcal{L}(z)$.

In the simple example where the given polyhedron is a cube, $\{(i,j,k) \mid 1 \leq i,j,k \leq n\}$, the scanning code is as follows.

```
        i = 1         \
        j = 1         | first
        k = 1         /
2       CONTINUE
c   --- loop body ---
        if (k.le.n-1) then    \
          k = 1+k             |
          GOTO 2              |
        endif                 |
        if (j.le.n-1) then    |
          k = 1               | next
          j = 1+j             |
          GOTO 2              |
        endif                 |
        if (i.le.n-1) then    |
          k = 1               |
          j = 1               |
          i = 1+i             |
          GOTO 2              |
        endif                 /
1       CONTINUE
```

Remark. The storage of the new values of `i`, `j` and `k` — which represent x_1, x_2 and x_3— is done in reverse order to avoid using temporary variables. This is not useful on this particular example but the advantage of this method can be seen in more complicated examples. We have also suppressed useless storage statements like `i = i`.

An other important speed factor in loops is that loop bounds are only evaluated once at the beginning of the loop. We can refine our code generation scheme to do the same here. In each predicate of a conditional, one has just to compute the invariant part beforehand and store it in some variable. Here is what our example finally looks like:

```
        i = 1
        j = 1
        k = 1
        b3 = -1+n
4       b2 = -1+n
3       b1 = -1+n
2       CONTINUE
c   --- loop body ---
        if (k.le.b1) then
          k = 1+k
          GOTO 2
        endif
        if (j.le.b2) then
          k = 1
          j = 1+j
          GOTO 3
        endif
        if (i.le.b3) then
          k = 1
          j = 1
          i = 1+i
          GOTO 4
        endif
1       CONTINUE
```

Note. If `i` is the time variable, one just has to insert a `synch(i)` just after statement 4.

The generated code has the following general form:

```
        x⃗ = first
        computation of level 1 constants
label_1 computation of level 2 constants
label_2 computation of level 3 constants
        ...
label_{m-1} computation of level m constants
label_m loop body
        next_m ↣ goto label_m
        next_{m-1} ↣ goto label_{m-1}
        ...
        next_1 ↣ goto label_1
```

where "$\text{next}_i \rightarrowtail$ `goto` label_i" corresponds to a selection tree (`if then else`) which translates the QUAST next_i. At each leaf of this selection tree, there are, first, the storage of the new values of the indices ($\vec{x}$), and next, a "`goto` label_i" statement to start the next iteration.

4. Experimental results

We study here how the code we generate compares with loop methods in terms of compactness and speed.

4.1. Simple 3D example: *permutation*

The domain we consider here is the affine image of the polyhedron:

$$\begin{cases} 1 \leq i \leq n \\ 1 \leq j \leq n \\ 1 \leq k \leq i+j \end{cases}$$

by the permutation

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

Any (good) loop transformation method would produce the following code:

```
        do x = 1, 2*n
          do y = max(1,x-n), n
            do z = max(1,x-y), n
c   --- loop body ---
            enddo
          enddo
        enddo
```

Here is the code we produce:

```
      x = 1
      y = 1
      z = 1
      b9 = -1+2*n
      b8 = 1-n
8     b7 = -1+n
      b5 = -2+x
      b6 = -1+x
7     b4 = -1+n
6     CONTINUE
c  --- loop body ---
      if (z.le.b4) then
        z = 1+z
        GOTO 6
      endif
      if (y.le.b7) then
        if (y.ge.b5) then
          z = 1
          y = 1+y
          GOTO 7
        else
          z = b6-y
          y = 1+y
          GOTO 7
        endif
      endif
      if ((n-x).ge.0) then
        z = x
        y = 1
        x = 1+x
        GOTO 8
      else
        if (x.le.b9) then
          z = n
          y = b8+x
          x = 1+x
          GOTO 8
        endif
      endif
5     CONTINUE
```

The two codes share a common structure. Hence, the execution times are similar with a slight advantage to the second one. The running times and code sizes are shown in Tab. 1.

time (s) / size (bytes)	loops	gotos
without opt.	1.87 / 2,192	1.75 / 3,300
with opt.	0.14 / 1,416	0.12 / 1,668

Table 1. Execution times and object code sizes for the permutation example

All the execution times have been measured on a SUN SPARCstation 20. All the programs have been compiled with the SUN f77 FORTRAN compiler. We have compiled our programs both without any compiler optimization and with the `-O` option.

These results confirm that the two codes share a common structure, but as we generate lower level code, we are able to have slightly better execution times.

Remark. Further optimizations such as replacing non strict inequalities by strict ones to remove some intermediate variables or merging some intermediate variables with the same value may reduce the non optimized running time but do not improve the optimized running time. This is due to the good optimizations done by the compiler. These optimizations can even increase the optimized running time because they reduce the number of variables, which reduces opportunities to allocate variables to registers.

Although the Fortran codes have different lengths, the object codes generated have nearly the same size with a slight advantage to the loop version.

4.2. Lefur's Example P1

This more complex example is detailed in an extended version of this article [2]. In this case, the goto code is slightly slower than the do-loop code.

5. Extension to unions of linearly bounded lattices

We study here how the method presented in Sect. 3 can be extended to deal with unions of LBLs, which appear when generating code for non perfect loop nests where each statement may be subjected to a different affine transformation: each LBL is the image of the iteration domain of some statement by an affine transformation.

5.1. Preliminary remarks

The basic idea is to apply the previous method to each LBL and to combine the results. There are however some issues to deal with:

LBLs of different dimensions. To be able to speak about union of sets, we must deal with sets of the same dimension. If one set has less dimensions than the others, one can just complete the missing dimensions with a constant. The choice of this constant is arbitrary; one can set it to 0.

Reducing the complexity of the computation. We have remarked that computing a minimum of several QUASTs can be costly, as much during code generation as during the execution of this code. So one of our aims when developing the code generation algorithm for unions of LBLs has been to avoid computing these minima.

Code duplication. Code duplication is also an issue when dealing with this kind of transformation. Though it can be costly, we have not focused on its complete elimination.

5.2. Formal method

Considering the previous remarks, we have designed a method to iterate over unions of LBLs.

Let us consider that we have p LBLs $\mathcal{L}_1(z), \ldots, \mathcal{L}_p(z)$, depending on some parameters $z \in \mathcal{D}$. These LBLs are defined by:

$$\mathcal{L}_i(z) = \{x \in \mathbb{Z}^n \mid \exists y \in \mathbb{Z}^{m_i}, A_i x + B_i y + C_i z \leq d_i\} \ .$$

We note $\mathcal{U}(z) = \bigcup_{1 \leq i \leq p} \mathcal{L}_i(z)$.

We use here the same method as in Sect. 3.2, by building a constant, "first", and a function, "next". "first" is defined as: $\mathrm{first} = \min_{\prec}\{y \in \mathcal{U}(z)\}$, and as before, this is an integer programming problem directly solvable by PIP. Function "next" is defined as:

$$\begin{aligned} \mathrm{next} : \mathcal{U}(z) &\to \mathcal{U}(z) \cup \{\bot\} \\ x &\mapsto \min_{\prec}\{y \in \mathcal{U}(z) \mid x \prec y\} \ . \end{aligned}$$

To compute this function, we decompose it into linear integer programming problems.

If we suppose that, for a given point x of $\mathcal{U}(z)$, we know which $\mathcal{L}_i(z)$ it belongs to, $\mathrm{next}(x)$ can be expressed as: $\mathrm{next}_i(x) = \min_{\prec j \in \mathbb{N}_p^*} \mathrm{next}_{i,j}(x)$ for any i such that $x \in \mathcal{L}_i(z)$ and where $\mathrm{next}_{i,j}$ is defined by:

$$\begin{aligned} \mathrm{next}_{i,j} : \mathcal{L}_i(z) &\to \mathcal{L}_j(z) \cup \{\bot\} \\ x &\mapsto \min_{\prec}\{y \in \mathcal{L}_j(z) \mid x \prec y\} \ . \end{aligned}$$

As in Sect. 3.2, we decompose the constraint $x \prec y$ into k linear constraints to build the linear problems $\mathcal{P}_{i,j}^k, 1 \leq i, j \leq p, 1 \leq k \leq n$ whose solutions are the following point in $\mathcal{L}_j$ whose first different coordinate is the k-th, considering that the current point is in $\mathcal{L}_i$. The general form of $\mathcal{P}_{i,j}^k$ is:

minimize x' subject to the constraints:

$$\begin{cases} Ez \leq f \\ A_i x + B_i y + C_i z \leq d_i \\ A_j x' + B_j y' + C_j z \leq d_j \\ x_1 = x'_1 \\ \vdots \\ x_{k-1} = x'_{k-1} \\ x_k < x'_k \end{cases}$$

We then combine the results of these problems to find the following point, $\mathrm{next}_i^k(x)$, in $\mathcal{U}(z)$ whose first different coordinate is the k-th, considering that the current point x is in a selected $\mathcal{L}_i(z)$: it is the minimum of the solutions of the $\mathcal{P}_{i,j}^k$ for all j. Such a minimum of QUASTs is a QUAST. We label each branch of this minimum QUAST by the indices j of the QUASTs it comes from, i.e. the statements which should be executed at the corresponding point. This branch labeling allows us to know which $\mathcal{L}_j(z)$ the following point belongs to.

Taking the minimum of the $\mathrm{next}_i^k(x)$ gives us $\mathrm{next}_i(x)$. These functions, next_i^k, are sufficient to compute function next in all cases. In fact, let us suppose that we know, for the current point x, the set $\mathcal{I}_x(z) = \{i \mid x \in \mathcal{L}_i(z)\}$. The following point is one of the $\mathrm{next}_i(x)$ for $i \in \mathcal{I}_x(z)$. Indeed, as the problems $\mathcal{P}_{i,j}^k$ differ only by their context for different $i \in \mathcal{I}_x(z)$, all their solutions are correct. So all the $\mathrm{next}_i(x)$ with $i \in \mathcal{I}_x(z)$ are the point following x in $\mathcal{U}(z)$. And, by reading the labels of the QUAST branches, we can now

identify $\mathcal{I}_{\mathrm{next}(x)}(z)$. By induction, we can iterate all the points of $\mathcal{U}(z)$ given the computation of all the $\mathcal{P}_{i,j}^k$.

We only lack the knowledge of $\mathcal{I}_{\mathrm{first}}(z)$. The solution is to decompose the linear problem $\min_{\prec}\{y \in \mathcal{U}(z)\}$ into $\min_{\prec}\{\min_{\prec}\{y \in \mathcal{L}_i(z)\} \mid i \in \mathbb{Z}_p^*\}$ and to label the branches as above.

Let us clarify this on a simple example :

Example 5.1 (Simple 2D example).
Let us consider two statements ① and ② whose iteration domains are $D_1 = D_2 = \{(a,b) \mid 1 \le a \le n, 1 \le b \le n\}$ and that are transformed by, respectively:

$$f_1 : (a,b) \mapsto (3a,b) \text{ and } f_2 : (a,b) \mapsto (3a+1,b).$$

k	i	next_i^k
2	1	**if** $b \le n-1$ **then** $a, b+1$ {①} **else** $\perp$
	2	**if** $b \le n-1$ **then** $a, b+1$ {②} **else** $\perp$
1	1	**if** $a \le 3n-3$ **then** $a+1, 1$ {②} **else** $\perp$
	2	**if** $a \le 3n-2$ **then** $a+2, 1$ {①} **else** $\perp$

Table 2. next_i^k.

Table 2 shows the next_i^k computed for this particular case. The symbols ① and ② label the leaves of next_i^k, indicating which statement has to be executed at the corresponding point.

$\min_{\prec} f_1(D_1)$	$\min_{\prec} f_2(D_2)$	start
$3,1$	$4,1$	$3,1$ {①}

Table 3. Starting point.

To determine the starting point, we have to solve the minima of $f_1(D_1)$ and $f_2(D_2)$ and their minimum. See Tab. 3 for these computations.

We have now computed all the information needed to scan the union $\mathcal{L}_1(n) \cup \mathcal{L}_2(n)$ lexicographically.

5.3. Code generation

Let us now explore the details of the iteration code generation. We have the same constraints as in Sect. 3.3: avoid as much as possible unneeded computations. The solution is also the same as before: we will try to avoid computing several times the same expressions.

Let us remark that the labels of the branches of the QUASTs are sets of statements (the $\mathcal{I}_x(z)$). So, to be able to do a complete optimization, we have to consider these sets and to generate a separate piece of code for each of them. Let us label these sets $\mathcal{I}^1, \mathcal{I}^2, \ldots, \mathcal{I}^S$. The abstract coding scheme is described below.

```
        x, s = first
        if x = ⊥ then goto endlabel
        goto label_s
label_1 statements of set I^1
        x, s = next_{i∈I^1}(x)
        if x = ⊥ then goto endlabel
        goto label_s
        ...
label_S statements of set I^S
        x, s = next_{i∈I^1}(x)
        if x = ⊥ then goto endlabel
        goto label_s
endlabel
```

To generate such a code while avoiding computing minima of QUASTs, the base functions we use are the next_i^k. We can now extract from the QUASTs all constant expressions with respect to their nesting level. Reusing the same structure as in Sect. 3.3, we would like to branch directly to the deepest nesting level, thus avoiding the computation of outer level constants.

We must be careful when doing this because some statements appear in several sets $\mathcal{I}^s$ and so we have to devise a mean to compute the constants at the right time. One solution is to use variables indicating whether these computations have to be done.

Let us clarify this: let $\mathrm{init}_{i,k} =$ `true` mean that constants for statement i at dimension k have to be computed. These variables are all initially `true`. They are set to `false` each time they are computed and to `true` whenever there is a change in the iteration dimension. We summarize this below with the coding scheme corresponding to the abstract code of a given set $\mathcal{I}^s$.

```
label_{s,1} foreach i ∈ I^s do
              if init_{i,1} then
                computation of level 1
                 constants for statement i
                init_{i,1} = false
              endif
            enddo
label_{s,2} foreach i ∈ I^s do
                if init_{i,2} then
                computation of level 2
                 constants for statement i
                init_{i,2} = false
                endif
```

```
        enddo
        ...
label_{s,d} foreach i ∈ J^s do
              if init_{i,d} then
            computation of level d
             constants for statement i
            init_{i,d} = false
              endif
        enddo
        computation of the statements
         of set J^s
        next_{i,d} ↣ goto label_{j,d}
        do j = 1, n
              init_{j,d} = true
        enddo
        next_{i,d-1} ↣ goto label_{j,d-1}
        do j = 1, n
              init_{j,d-1} = true
        enddo
        ...
        next_{i,1} ↣ goto label_{j,1}
        goto endlabel
```

Finally, using the same optimizations as in Sect. 3.3, we are now able to generate the iteration code for a union of LBLs. The generated code for our simple example is available in an extended version of this paper[2].

6. Conclusion

We have presented a general method for scanning linearly bounded lattices and unions of linearly bounded lattices. The power of this method comes from the fact that it accepts any number of parameters. Indeed, as all the underlying computations are done with PIP which is parameterized, one can for example parameterize the code generation by the number of the target processor and the size of the domain.

This method distinguishes itself from the other existing ones by producing low level code and not using Do-loops. We have however tried to keep the efficiency of the loop structure and the experimental results we have obtained show that we have achieved our goal. We should note that the efficiency of the produced code depends greatly on the optimizations done afterwards by the native compiler.

The codes we generate may look very complex. One however must keep three points in mind:

- Firstly, the generated code is to be compiled without human intervention. Hence, the only relevant figure of merit is execution time. From this point of view, our codes qualify.
- Next, we generate directly low level code. Low level code equivalent to loops such as in Sect. 4.2 would probably look as complicated as our own.
- Lastly, in many cases, the scanning code is inherently complex. The constraint that the scanning code must be simple and elegant should be taken care of when selecting code transformations. How to express this constraint and solve the associated optimization problem is unknown at present.

Further experimentation of the general case will be done with the inclusion of the prototype in a complete parallelizer.

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, Apr. 1991.

[2] P. Boulet and P. Feautrier. Scanning polyhedra without do-loops. Technical report, Laboratoire PR*i*SM, Université de Versailles-St Quentin en Yvelines, France, 1998. available at `http://www.lifl.fr/~boulet/publi/polyscanRR.ps.gz`.

[3] Z. Chamski. Scanning polyhedra with do loop sequences. In *Workshop on Parallel Algorithms '92*, Sofia, 1992.

[4] J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. Int. Conf. on Application in Parallel and Distributed Computing. IFIP WG 10.3*, pages 185–194. North Holland, Apr. 1994.

[5] J.-F. Collard, P. Feautrier, and T. Risset. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3):421–436, Sept. 1995.

[6] A. Darte. *Techniques de parallélisation automatique de nids de boucles*. PhD thesis, Ecole Normale Supérieure de Lyon, 1993.

[7] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opèrationnelle*, 22:243–268, Sept. 1988.

[8] P. Feautrier. Semantical analysis and mathematical programming. In M. C. et al., editor, *Parallel and distributed algorithms*, pages 309–320, North Holland, 1989. Elsevier Science Publishers B.V.

[9] F. Irigoin. *Partitionnement des boucles imbriquées, une technique d'optimisation pour les programmes scientifiques*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, June 1987.

[10] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *The 5th Symposium on Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, Feb. 1995.

[11] G.-R. Perrin and C. Reffay. Communication code generation in systems of affine recurrence equations. *Integration: the VLSI Journal*, 20:63–83, 1995.

[12] J. Xue. Automatic non-unimodular transformations of loop nests. *Parallel Computing*, 20(5):711–728, May 1994.

[13] J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22:1621–1645, Feb. 1997.

Annexe C

Static Tiling for Heterogeneous Computing Platforms

Tiling for Heterogeneous Computing Platforms*

Pierre Boulet[1], Jack Dongarra[2,3], Yves Robert[1,2] and Frédéric Vivien[1]

[1] LIP, Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France
[2] Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, USA
[3] Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
e-mail: [Pierre.Boulet, Yves.Robert, Frederic.Vivien]@ens-lyon.fr
e-mail: dongarra@cs.utk.edu

Abstract

In the framework of fully permutable loops, tiling has been extensively studied as a source-to-source program transformation. However, little work has been devoted to the mapping and scheduling of the tiles on physical processors. Moreover, targeting heterogeneous computing platforms has, to the best of our knowledge, never been considered. In this paper we extend tiling techniques to the context of limited computational resources with different-speed processors. In particular, we present efficient scheduling and mapping strategies that are asymptotically optimal. The practical usefulness of these strategies is fully demonstrated by MPI experiments on a heterogeneous network of workstations.

Key words: tiling, communication-computation overlap, mapping, limited resources, different-speed processors, heterogeneous networks

*This work was supported in part by the National Science Foundation Grant No. ASC-9005933; by the Defense Advanced Research Projects Agency under contract DAAH04-95-1-0077, administered by the Army Research Office; by the Office of Scientific Computing, U.S. Department of Energy, under Contract DE-AC05-84OR21400; by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615; by the CNRS–ENS Lyon–INRIA project *ReMaP*; and by the Eureka Project *EuroTOPS*. Yves Robert's work was conducted while he was on leave from Ecole Normale Supérieure de Lyon and partly supported by DRET/DGA under contract ERE 96-1104/A000/DRET/DS/SR.

1 Introduction

Tiling is a widely used technique to increase the granularity of computations and the locality of data references. This technique applies to sets of fully permutable loops [23, 15, 11]. The basic idea is to group elemental computation points into tiles that will be viewed as computational units (the loop nest must be permutable so that such a transformation is valid). The larger the tiles, the more efficient are the computations performed using state-of-the-art processors with pipelined arithmetic units and a multilevel memory hierarchy (this feature is illustrated by recasting numerical linear algebra algorithms in terms of blocked Level 3 BLAS kernels [12, 9]). Another advantage of tiling is the decrease in communication time (which is proportional to the surface of the tile) relative to the computation time (which is proportional to the volume of the tile). The price to pay for tiling may be an increased latency; for example, if there are data dependencies, the first processor must complete the whole execution of the first tile before another processor can start the execution of the second one. Tiling also presents load-imbalance problems: the larger the tile, the more difficult it is to distribute computations equally among the processors.

Tiling has been studied by several authors and in different contexts (see, for example, [14, 20, 22, 19, 4, 21, 6, 17, 1, 8, 16, 7, 13]). Rather than providing a detailed motivation for tiling, we refer the reader to the papers by Calland, Dongarra, and Robert [7] and by Högsted, Carter, and Ferrante [13], which provide a review of the existing literature. Briefly, most of the work amounts to partitioning the iteration space of a uniform loop nest into tiles whose shape and size are optimized according to some criterion (such as the communication-to-computation ratio). Once the tile shape and size are defined, the tiles must be distributed to physical processors and the final scheduling must be computed.

A natural way to allocate tiles to physical processors is to use a cyclic allocation of tiles to processors. Several authors [17, 13, 3] suggest allocating columns of tiles to processors in a purely scattered fashion (in HPF words, this is a `CYCLIC(1)` distribution of tile columns to processors). The intuitive motivation is that a cyclic distribution of tiles is quite natural for load-balancing computations. Once the distribution of tiles to processors is fixed, there are several possible schedulings; indeed, any wavefront execution that goes along a left-to-right diagonal is valid. Specifying a columnwise execution may lead to the simplest code generation.

When all processors have equal speed, it turns out that a pure cyclic columnwise allocation provides the best solution among all possible distributions of tiles to processors [7]—provided that the communication cost for a tile is not greater than the computation cost. Since the communication cost for a tile is proportional to its surface, while the computation cost is proportional to its volume,[1] this hypothesis will be satisfied if the tile is large enough.[2]

However, the recent development of heterogeneous computing platforms poses a new challenge: that of incorporating processor speed as a new parameter of the tiling problem. Intuitively, if the user wants to use a heterogeneous network of computers where, say, some processors are twice as fast as some other processors, we may want to assign twice as many tiles to the faster processors. A cyclic distribution is not likely to lead to an efficient implementation. Rather, we should use strategies that aim at load-balancing the work while not introducing idle time. The design of such strategies is the goal of this paper.

The rest of the paper is organized as follows. In Section 2 we formally state the problem of tiling

[1] For example, for two-dimensional tiles, the communication cost grows linearly with the tile size while the computation cost grows quadratically.

[2] Of course, we can imagine a theoretical situation in which the communication cost is so large that a sequential execution would lead to the best result.

for heterogeneous computing platforms. All our hypotheses are listed and discussed, and we give a theoretical way to solve the problem by casting it in terms of a linear programming problem. The cost of solving the linear problem turns out to be prohibitive in practice, so we restrict ourselves to columnwise allocations. Fortunately, there exist asymptotically optimal columnwise allocations, as shown in Section 3, where several heuristics are introduced and proved. In Section 4 we provide MPI experiments that demonstrate the practical usefulness of our columnwise heuristics on a network of workstations. Finally, we state some conclusions in Section 5.

2 Problem Statement

In this section, we formally state the scheduling and allocation problem that we want to solve. We provide a complete list of all our hypotheses and discuss each in turn.

2.1 Hypotheses

(H1) The computation domain (or iteration space) is a two-dimensional rectangle[3] of size $N_1 \times N_2$. Tiles are rectangular, and their edges are parallel to the axes (see Figure 1). All tiles have the same fixed size. Tiles are indexed as $T_{i,j}$, $0 \leq i < N_1$, $0 \leq j < N_2$.

(H2) Dependences between tiles are summarized by the vector pair

$$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}.$$

In other words, the computation of a tile cannot be started before both its left and upper neighbor tiles have been executed. Given a tile $T_{i,j}$, we call both tiles $T_{i+1,j}$ and $T_{i,j+1}$ its successors, whenever the indices make sense.

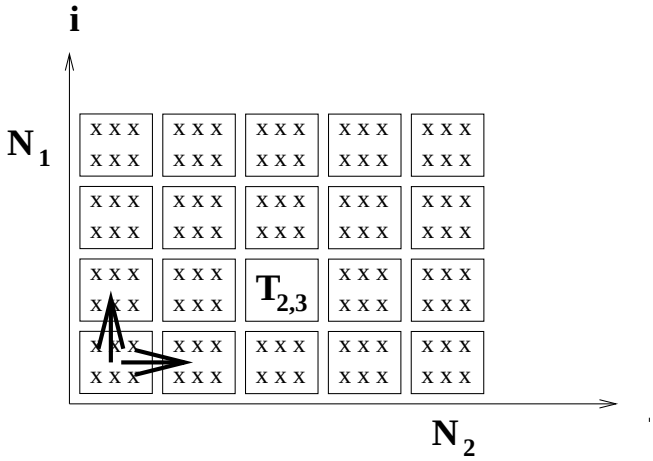


Figure 1: A tiled iteration space with horizontal and vertical dependencies.

[3]In fact, the dimension of the tiles may be greater than 2. Most of our heuristics use a columnwise allocation, which means that we partition a single dimension of the iteration space into chunks to be allocated to processors. The number of remaining dimensions is not important.

(H3) There are P available processors interconnected as a (virtual) ring.[4] Processors are numbered from 0 to $P-1$. Processors may have different speeds: let t_q the time needed by processor P_q to execute a tile, for $0 \leq q < P$. While we assume the computing resources are heterogeneous, we assume the communication network is homogeneous: if two adjacent tiles T and T' are not assigned to the same processor, we pay the same communication overhead T_{com}, whatever the processors that execute T and T'.

(H4) Tiles are assigned to processors by using a scheduling σ and an allocation function *proc* (both to be determined). Tile T is allocated to processor $\text{proc}(T)$, and its execution begins at time-step $\sigma(T)$. The constraints[5] induced by the dependencies are the following: for each tile T and each of its successors T', we have

$$\begin{cases} \sigma(T) + t_{\text{proc}(T)} \leq \sigma(T') & \text{if } \text{proc}(T) = \text{proc}(T') \\ \sigma(T) + t_{\text{proc}(T)} + T_{\text{com}} \leq \sigma(T') & \text{otherwise} \end{cases}$$

The makespan $MS(\sigma, \text{proc})$ of a schedule-allocation pair (σ, proc) is the total execution time required to execute all tiles. If execution of the first tile $T_{0,0}$ starts at time-step $t = 0$, the makespan is equal to the date at which the execution of the last tile is executed:

$$MS(\sigma, \text{proc}) = \sigma(T_{N_1,N_2}) + t_{\text{proc}(T_{N_1,N_2})}.$$

A schedule-allocation pair is said to be optimal if its makespan is the smallest possible over all (valid) solutions. Let T_{opt} denote the optimal execution time over all possible solutions.

2.2 Discussion

We survey our hypotheses and assess their motivations, as well as the limitations that they may induce.

Rectangular iteration space and tiles We note that the tiled iteration space is the outcome of previous program transformations, as explained in [14, 20, 22, 19, 4]. The first step in tiling amounts to determining the best shape and size of the tiles, assuming an infinite grid of virtual processors. Because this step will lead to tiles whose edges are parallel to extremal dependence vectors, we can perform a unimodular transformation and rewrite the original loop nest along the edge axes. The resulting domain may not be rectangular, but we can approximate it using the smallest bounding box (however, this approximation may impact the accuracy of our results).

Dependence vectors We assume that dependencies are summarized by the vector pair $\mathcal{V} = \{(1,0)^t, (0,1)^t\}$. Note that these are dependencies between tiles, not between elementary computations. Hence, having right- and top-neighbor dependencies is a very general situation if the tiles are large enough. Technically, since we deal with a set of fully permutable loops, all dependence vectors have nonnegative components only, so that $\mathcal{V}$ permits all other dependence vectors to be generated by transitivity. Note that having a dependence vector $(0,a)^t$ with $a \geq 2$ between tiles, *instead of* having vector $(0,1)^t$, would mean unusually long

[4] The actual underlying physical communication network is not important.

[5] There are other constraints to express (e.g., any processor can execute at most one tile at each time-step). See Section 2.3 for a complete formalization.

dependencies in the original loop nest, while having $(0, a)^t$ *in addition to* $(0, 1)^t$ as a dependence vector between tiles is simply redundant. In practical situations, we might have an additional diagonal dependence vector $(1, 1)^t$ between tiles, but the diagonal communication may be routed horizontally and then vertically, or the other way round, and even may be combined with any of the other two messages (because of vectors $(0, 1)^t$ and $(1, 0)^t$).

Computation-communication overlap Note that in our model, communications can be overlapped with the computations of other (independent) tiles. Assuming communication-computation overlap seems a reasonable hypothesis for current machines that have communication coprocessors and allow for asynchronous communications (posting instructions ahead, or using active messages). We can think of independent computations going along a thread while communication is initiated and performed by another thread [18]. An interesting approach has been proposed by Andonov and Rajopadhye [3]: they introduce the *tile period* P_t as the time elapsed between corresponding instructions of two successive tiles that are mapped to the *same* processor, while they define the *tile latency* L_t to be the time between corresponding instructions of two successive tiles that are mapped to *different* processors. The power of this approach is that the expressions for L_t and P_t can be modified to take into account several architectural models. A detailed architectural model is presented in [3], and several other models are explored in [2]. With our notation, $P_t = t_i$ and $L_t = t_i + T_{\text{com}}$ for processor P_i.

Homogeneous communication network We assume that the communication time for a tile T_{com} is independent of the two processors exchanging the message. This is a crude simplification because the network interfaces of heterogeneous systems are likely to exhibit very different latency characteristics. However, because communications can be overlapped with independent computations, they eventually have little impact on the performance, as soon as the granularity (the tile size) is chosen large enough. This theoretical observation has been verified during our MPI experiments (see Sectionsec:simul).

Finally, we briefly mention another possibility for introducing heterogeneity into the tiling model. We chose to have all tiles of same size and to allocate more tiles to the faster processors. Another possibility is to evenly distribute tiles to processors, but to let their size vary according to the speed of the processor they are allocated to. However, this strategy would severely complicate code generation. Also, allocating several neighboring fixed-size tiles to the same processor will have similar effects as allocating variable-size tiles, so our approach will cause no loss of generality.

2.3 ILP Formulation

We can describe the tiled iteration space as a task graph $G = (V, E)$, where vertices represent the tiles and edges represent dependencies between tiles. Computing an optimal schedule-allocation pair is a well-known task graph scheduling problem, which is NP-complete in the general case [10].

If we want to solve the problem as stated (hypotheses (H1) to (H4)), we can use an integer linear programming formulation. Several constraints must be satisfied by any valid schedule-allocation pair. In the following, T_{max} denotes an upper bound on the total execution time. For example, T_{max} can be the execution time when all the tiles are given to the fastest processor: $T_{max} = N_1 \times N_2 \times \min_{0 \leq i < P} t_i$.

We now translate these constraints into equations. In the following, let $i \in \{1, \ldots, N_1\}$ denote a row number, $j \in \{1, \ldots, N_2\}$ a column number, $q \in \{0, \ldots, P-1\}$ a processor number, and $t \in \{0, \ldots, T_{max}\}$ a time-step.

- **Number of executions**. Let $B_{i,j,q,t}$ be an integer variable indicating whether the execution of tile $T_{i,j}$ began at time-step t on processor q: if this is the case, then $B_{i,j,q,t} = 1$, and $B_{i,j,q,t} = 0$ otherwise. Each tile must be executed once, and thus starts at one and only one time-step. Therefore, the constraints are

$$\forall i,j,q,t, \quad B_{i,j,q,t} \geq 0 \text{ and } \forall i,j, \sum_{q=0}^{P-1} \sum_{t=0}^{T_{max}} B_{i,j,q,t} = 1.$$

- **Execution place and date**. Using $B_{i,j,q,t}$, we can compute the date $D_{i,j}$ at which tile (i,j) starts execution. We can also check which processor q processes tile (i,j). The 0/1 result is stored in $P_{i,j,q}$:

$$\forall i,j, \quad D_{i,j} = \sum_{p=0}^{P-1} \sum_{t=0}^{T_{max}} t \times B_{i,j,q,t} \quad \text{and} \quad \forall i,j,q, \quad P_{i,j,q} = \sum_{t=0}^{T_{max}} B_{i,j,q,t}.$$

- **Communications**. There must be a communication delay between the end of execution of tile $(i-1,j)$ (resp. $(i,j-1)$) and the beginning of execution of tile (i,j) if and only the two tiles are not executed by the same processor, that is, if and only if there exists q such that $P_{i,j,q} \neq P_{i-1,j,q}$ (resp. $P_{i,j,q} \neq P_{i,j-1,q}$). The boolean result is stored in $v_{i,j}$ (resp. $h_{i,j}$)): $v_{i,j} = 1$ if tiles $(i-1,j)$ and (i,j) are not executed by the same processor, and $v_{i,j} = 0$ otherwise. We have a similar definition for $h_{i,j}$) suing tiles $(i,j-1)$ and (i,j). The equations are:

$$\forall i \geq 2,j,q, \quad v_{i,j} \geq P_{i,j,q} - P_{i-1,j,q}, \quad v_{i,j} \geq P_{i-1,j,q} - P_{i,j,q}$$

$$\forall i,j \geq 2,q, \quad h_{i,j} \geq P_{i,j,q} - P_{i,j-1,q}, \quad v_{i,j} \geq P_{i,j-1,q} - P_{i,j,q}$$

  Note that if a communication delay is needed between the execution of tile $(i-1,j)$ and that of tile (i,j), then $v_{i,j}$ will impose one. If none is needed, $v_{i,j}$ may still be equal to 1, as long as this does not increase the total execution time.

- **Precedence constraints**. The execution of tile $(i-1,j)$ (resp. $(i,j-1)$) must be finished, and the data transferred, before the beginning of execution of tile (i,j):

$$\forall i \geq 2,j, \quad D_{i,j} \geq D_{i-1,j} + v_{i,j} T_{com} + \sum_{q=0}^{P-1} P_{i-1,j,q} t_q$$

$$\forall i,j \geq 2, \quad D_{i,j} \geq D_{i,j-1} + h_{i,j} T_{com} + \sum_{q=0}^{P-1} P_{i,j-1,q} t_q$$

- **Number of tiles executed at any time-step.** A processor executes (at most) one tile at a time. Therefore processor q can start executing at most one tile in any interval of time t_q (as t_q is the time to execute a tile by processor q):

$$\forall q, \quad t_q - 1 \leq t \leq T_{max}, \quad \sum_{t'=t-t_q+1}^{t} \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} B_{i,j,q,t'} \leq 1$$

$$
\begin{cases}
\min\left(D_{N_1,N_2} + \sum_q P_{N_1,N_2,q} t_q\right) & \\
\sum_{t'=t-t_q+1}^{t} \sum_{i,j} B_{i,j,q,t'} \leq 1 & 0 \leq q \leq P-1, t_q - 1 \leq t \leq T_{max} \\
D_{i,j} \geq D_{i-1,j} + v_{i,j} T_{com} + \sum_q P_{i-1,j,q} t_q & 2 \leq i \leq N_1, 1 \leq j \leq N_2 \\
D_{i,j} \geq D_{i,j-1} + h_{i,j} T_{com} + \sum_q P_{i,j-1,q} t_q & 1 \leq i \leq N_1, 2 \leq j \leq N_2 \\
v_{i,j} \geq P_{i,j,q} - P_{i-1,j,q} & 2 \leq i \leq N_1, 1 \leq j \leq N_2, 0 \leq q \leq P-1 \\
v_{i,j} \geq P_{i-1,j,q} - P_{i,j,q} & 2 \leq i \leq N_1, 1 \leq j \leq N_2, 0 \leq q \leq P-1 \\
h_{i,j} \geq P_{i,j,q} - P_{i,j-1,q} & 1 \leq i \leq N_1, 2 \leq j \leq N_2, 0 \leq q \leq P-1 \\
h_{i,j} \geq P_{i,j-1,q} - P_{i,j,q} & 1 \leq i \leq N_1, 2 \leq j \leq N_2, 0 \leq q \leq P-1 \\
P_{i,j,q} = \sum_t B_{i,j,q,t} & 1 \leq i \leq N_1, 1 \leq j \leq N_2, 0 \leq q \leq P-1 \\
D_{i,j} = \sum_q \sum_t t B_{i,j,q,t} & 1 \leq i \leq N_1, 1 \leq j \leq N_2 \\
\sum_q \sum_t B_{i,j,q,t} = 1 & 1 \leq i \leq N_1, 1 \leq j \leq N_2 \\
B_{i,j,q,t} \geq 0 & 1 \leq i \leq N_1, 1 \leq j \leq N_2, 0 \leq q \leq P-1, 0 \leq t \leq T_{max}
\end{cases}
$$

Figure 2: Integer linear program that optimally solves the schedule-allocation problem.

Now that we have expressed all our constraints in a linear way, we can write the whole linear programming system. We need only to add the objective function: the minimization of the time-step at which the execution of the last tile T_{N_1,N_2} is terminated. The final linear program is presented in Figure 2. Since an optimal rational solution of this problem is not always an integer solution, this program must be solved as an *integer* linear program.

The main drawback of the linear programming approach is its huge cost. The program shown Figure 2 contains more than $PN_1N_2T_{max}$ variables and inequalities. The cost of solving such a problem would be prohibitive for any practical application. Furthermore, even if we could solve the linear problem, we might not be pleased with the solution. We probably would prefer non-optimal but "regular" allocations of tiles to processors, such as columnwise or rowwise allocations. Fortunately, such allocations can lead to asymptotically optimal solutions, as shown in the next section.

3 Columnwise Allocation

Before introducing asymptotically optimal columnwise (or rowwise) allocations, we give a small example to show that columnwise allocations (or equivalently rowwise allocations) are not optimal.

3.1 Optimality and Columnwise Allocations

Consider a tiled iteration space with $N_2 = 2$ columns, and suppose we have $P = 2$ processors such that $t_1 = 5 \times t_0$: the first processor is five times faster than the second one. Suppose for the sake of simplicity that $T_{\text{com}} = 0$. If we use a columnwise allocation,

- either we allocate both columns to processor 0, and the makespan is $MS = 2N_1t_0$
- or we allocate one column to each processor, and the makespan is greater than N_1t_1 (a lower bound time for the slow processor to process its column)

The best solution is to have the fast processor execute all tiles. But if N_1 is large enough, we can do better by allocating a small fraction of the first column (the last tiles) to the slow processor, which will process them while the first processor is active executing the first tiles of the second column. For instance, if $N_1 = 6n$ and if we allocate the last n tiles of the first column to the slow processor (see Figure 3), the execution time becomes $MS = 11nt_0 = \frac{11}{6}N_1t_0$, which is better than the best columnwise allocation.[6]

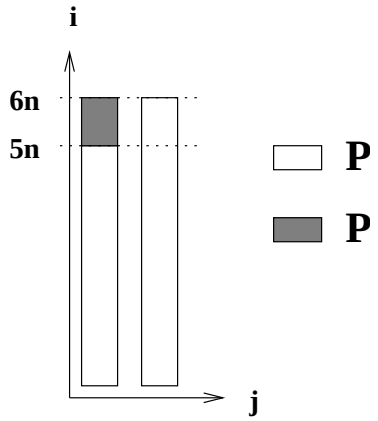


Figure 3: Allocating tiles for a two-column iteration space.

This small example shows that our target problem is intrinsically more complex than the instance with same-speed processors: as shown in [7], a columnwise allocation would be optimal for our two-column iteration space with two processors of equal speed.

3.2 Heuristic Allocation by Block of Columns

Throughout the rest of the paper we make the following additional hypothesis:

(H5) We impose the allocation to be columnwise:[7] all tiles $T_{i,j}$, $1 \leq i \leq N_1$, are allocated to the same processor.

We start with an easy lemma to bound the optimal execution time T_{opt}:

Lemma 1

$$T_{opt} \geq \frac{N_1 \times N_2}{\sum_{i=0}^{P-1} \frac{1}{t_i}}.$$

Proof Let x_i be the number of tiles allocated to processor i, $0 \leq i < P$. Obviously, $\sum_{i=0}^{P-1} x_i = N_1N_2$. Even if we take into account neither the communication delays nor the dependence constraints, the execution time T is greater than the computation time of each processor: $T \geq x_it_i$ for all $0 \leq i < P$. Rewriting this as $x_i \leq T/t_i$ and summing over i, we get $N_1N_2 = \sum_{i=0}^{P-1} x_i \leq (\sum_{i=0}^{P-1} \frac{1}{t_i})T$, hence the result. ■

The proof of Lemma 1 leads to the (intuitive) idea that tiles should be allocated to processors *in proportion to* their relative speeds, so as to balance the workload. Specifically, let

[6] This is not the best possible allocation, but it is superior to any columnwise allocation.

[7] Note that the problem is symmetric in rows and columns. We could study rowwise allocations as well.

$L = \text{lcm}(t_0, t_1, \ldots, t_{P-1})$, and consider an iteration space with L columns: if we allocate $\frac{L}{t_i}$ tile columns to processor i, all processors need the same number of time-steps to compute all their tiles: the workload is perfectly balanced. Of course, we must find a good schedule so that processors do not remain idle, waiting for other processors because of dependence constraints.

We introduce below a heuristic that allocates the tiles to processors by blocks of columns whose size is computed according to the previous discussion. This heuristic produces an asymptotically optimal allocation: the ratio of its makespan over the optimal execution time tends to 1 as the number of tiles (the domain size) increases.

In a columnwise allocation, all the tiles of a given column of the iteration space are allocated to the same processor. When contiguous columns are allocated to the same processor, they form a block. When a processor is assigned several blocks, the scheduling is the following:

1. Blocks are computed one after the other, in the order defined by the dependencies. The computation of the current block must be completed before the next block is started.

2. The tiles inside each block are computed in a rowwise order: if, say, 3 consecutive columns are assigned to a processor, it will execute the three tiles in the first row, then the three tiles in the second row, and so on. Note that (given 1.) this strategy is the best to minimize the latency (for another processor to start next block as soon as possible).

The following lemma shows that dependence constraints do not slow down the execution of two consecutive blocks (of adequate size) by two different-speed processors:

Lemma 2 *Let P_1 and P_2 be two processors that execute a tile in time t_1 and t_2, respectively. Assume that P_1 was allocated a block B_1 of c_1 contiguous columns and that P_2 was allocated the block B_2 consisting of the following c_2 columns. Let c_1 and c_2 satisfy the equality $c_1 t_1 = c_2 t_2$.*

Assume that P_1, starting at time-step s_1, is able to process B_1 without having to wait for any tile to be computed by some other processor. Then P_2 will be able to process B_2 without having to wait for any tile computed by P_1, if it starts at time $s_2 \geq s_1 + c_1 t_1 + T_{com}$.

Proof P_1 (resp. P_2) executes its block row by row. The execution time of a row is $c_1 t_1$ (resp. $c_2 t_2$). By hypothesis, it takes the same amount of time for P_1 to compute a row of B_1 as for P_2 to compute a row of B_2.

Since P_1 is able to process B_1 without having to wait for any tile to be computed by some other processor, it finishes computing the ith row of B_1 at time $s_1 + ic_1 t_1$.

P_2 cannot start processing the first tile of the ith row of B_2 before P_1 has computed the last tile of the ith row of B_1 and has sent that data to P_2, that is, at time-step $s_1 + ic_1 t_1 + T_{\text{com}}$.

Since P_2 starts processing the first row of B_2 at time s_2, where $s_2 \geq s_1 + c_1 t_1 + T_{\text{com}}$, it is not delayed by P_1. Later on, P_2 will process the first tile of the ith row of B_2 at time $s_2 + (i-1)c_2 t_2 = s_2 + (i-1)c_1 t_1 \geq s_1 + c_1 t_1 + T_{\text{com}} + (i-1)c_1 t_1 = s_1 + ic_1 t_1 + T_{\text{com}}$; hence P_2 will not be delayed by P_1. ■

We are ready to introduce our heuristic.

Heuristic

Let $P_0, \ldots, P_{P-1}$ be P processors that respectively execute a tile in time $t_0, \ldots, t_{P-1}$. We allocate column blocks to processors by chunks of $C = L \times \sum_{i=0}^{P-1} \frac{1}{t_i}$, where $L = \text{lcm}(t_0, t_1, \ldots, t_{P-1})$ columns. For the first chunk, we assign the block B_0 of the first L/t_0 columns to P_0, the block B_1 of the next

L/t_1 columns to P_1, and so on until P_{p-1} receives the last L/t_p columns of the chunk. We repeat the same scheme with the second chunk (columns $C+1$ to $2C$) first, and so on until all columns are allocated (note that the last chunk may be incomplete). As already said, processors will execute blocks one after the other, row by row within each block.

Lemma 3 *The difference between the execution time of the heuristic allocation by columns and the optimal execution time is bounded as*

$$T - T_{opt} \leq (P-1)T_{com} + (N_1 + P - 1)lcm(t_0, t_1, \ldots, t_{P-1}).$$

Proof Let $L = \text{lcm}(t_0, t_1, \ldots, t_{P-1})$. Lemma 2 ensures that, if processor P_i starts working at time-step $s_i = i(L + T_{\text{com}})$, it will not be delayed by other processors. By definition, each processor executes one block in time LN_1. The maximal number of blocks allocated to a processor is

$$n = \left\lceil \frac{N_2}{L \times \sum_{i=0}^{P-1} \frac{1}{t_i}} \right\rceil.$$

The total execution time, T, is equal to the date the last processor terminates execution. T can be bounded as follows:[8]

$$T \leq s_{P_1} + n \times LN_1.$$

On the other hand, T_{opt} is bounded below by Lemma 1. We derive

$$T - T_{opt} \leq (P-1)(L + T_{\text{com}}) + LN_1 \left\lceil \frac{N_2}{L \times \sum_{i=0}^{P-1} \frac{1}{t_i}} \right\rceil - \frac{N_1 \times N_2}{\sum_{i=0}^{P-1} \frac{1}{t_i}}.$$

Since $\lceil x \rceil \leq x + 1$ for any rational number x, we obtain the desired formula. ■

Proposition 1 *Our heuristic is asymptotically optimal: letting T be its makespan, and T_{opt} be the optimal execution time, we have*

$$\lim_{N_2 \to +\infty} \frac{T}{T_{opt}} = 1.$$

The two main advantages of our heuristic are (i) its regularity, which leads to an easy implementation; and (ii) its guarantee: it is theoretically proved to be close to the optimal. However, we will need to adapt it to deal with practical cases, because the number $C = L \times \sum_{i=0}^{P-1} \frac{1}{t_i}$ of columns in a chunk may be too large.

4 Practical Heuristics

In the preceding section, we described a heuristic that allocates blocks of columns to processors in a cyclic fashion. The size of the blocks is related to the relative speed of the processors. However, a straightforward application of our heuristic would lead to difficulties, as shown next in Section 4.1. Furthermore, the execution time variables t_i are not known accurately in practice. We explain how to modify the heuristic (computing different block sizes) in Section 4.2.

[8] Processor P_{P-1} is not necessarily the last one, because the last chunk may be incomplete.

4.1 Processor Speed

To expose the potential difficulties of the heuristic, we conducted experiments on a heterogeneous network of eight Sun workstations. To compute the relative speed of each workstation, we used a program that runs the same piece of computation that will be used later in the tiling program. Results are reported in Table 1.

Name	nala	bluegrass	dancer	donner	vixen	rudolph	zazu	simba
Description	Ultra 2	SS 20	SS 5	SS 5	SS 5	SS 10	SS1 4/60	SS1 4/60
Execution time t_i	11	26	33	33	38	40	528	530

Table 1: Measured computation times showing relative processor speeds.

To use our heuristic, we must allocate chunks of size $C = L \sum_{i=0}^{7} \frac{1}{t_i}$ columns, where $L = \text{lcm}(t_0, t_1, \ldots, t_7) = 34,560,240$. We compute that $C = 8,469,789$ columns, which would require a very large problem size indeed. Needless to say, such a large chunk is not feasible in practice. Also, our measurements for the processor speeds may not be accurate,[9] and a slight change may dramatically impact the value of C. Hence, we must devise another method to compute the sizes of the blocks allocated to each processor (see Section 4.2). In Section 4.3, we present simulation results and discuss the practical validity of our modified heuristics.

4.2 Modified Heuristic

Our goal is to choose the "best" block sizes allocated to each processor while bounding the total size of a chunk. We first define the cost of a block allocation and then describe an algorithm to compute the best possible allocation, given an upper bound for the chunk.

4.2.1 Cost Function

As before, we consider heuristics that allocate tiles to processors by blocks of columns, repeating each chunk in a cyclic fashion. Consider a heuristic defined by $\mathcal{C} = (c_0, \ldots, c_{P-1})$, where c_i is the number of columns in each block allocated to processor P_i:

Definition 1 *The cost of a block size allocation $\mathcal{C}$ is the maximum of the block computation times ($c_i t_i$) divided by the total number of columns computed in each chunk:*

$$cost(\mathcal{C}) = \frac{\max_{0 \le i \le P-1} c_i t_i}{\sum_{0 \le i \le P-1} c_i}$$

Considering the steady state of the computation, all processors work in parallel inside their block, so that the computation time of a whole chunk is the maximum of the computation times of the processors. During this time, $s = \sum_{0 \le i \le P-1} c_i$ columns are computed. Hence, the average time to compute a single column is given by our cost function. When the number of columns is much larger than the size of the chunk, the total computation time can well be approximated by $cost(\mathcal{C}) \times N_2$, the product of the average time to compute a column by the total number of columns.

[9] The 8 workstations were not dedicated to our experiments. Even though we were running these experiments during the night, some other users' processes might have been running. Also, we have averaged and rounded the results, so the error margin roughly lies between 5% and 10%.

4.2.2 Optimal Block Size Allocations

As noted before, our cost function correctly models reality when the number of columns in each chunk is much smaller than the total number of columns of the domain. We now describe an algorithm that returns the best (with respect to the cost function) block size allocation given a bound s on the number of columns in each chunk.

We build a function that, given a best allocation with a chunk size equal to $n-1$, computes a best allocation with a chunk size equal to n. Once we have this function, we start with an initial chunk size $n=0$, compute a best allocation for each increasing value of n up to $n=s$, and select the best allocation encountered so far.

First we characterize the best allocations for a given chunk size s:

Lemma 4 *Let $\mathcal{C}=(c_0,\ldots,c_{P-1})$ be an allocation, and let $s=\sum_{0\leq i\leq P-1} c_i$ be the chunk size. Let $m=\max_{1\leq i\leq p} c_i t_i$ denote the maximum computation time inside a chunk. If $\mathcal{C}$ verifies*

$$\forall i, 0\leq i\leq P-1,\ t_i c_i \leq m \leq t_i(c_i+1), \tag{1}$$

then it is optimal for the chunk size s.

Proof Take an allocation verifying the above condition 1. Suppose that it is not optimal. Then there exists a better allocation $\mathcal{C}'=(c'_0,\ldots,c'_{P-1})$ with $\sum_{0\leq i\leq P-1} c'_i = s$, such that

$$m' = \max_{0\leq i\leq P-1} c'_i t_i < m.$$

By definition of m, there exists i_0 such that $m=c_{i_0}t_{i_0}$. We can then successively derive

$$\begin{aligned}
c_{i_0}t_{i_0} = m > m' \geq c'_{i_0}t_{i_0} & \\
c_{i_0} > c'_{i_0} & \\
\exists i_1, c_{i_1} < c'_{i_1} \quad & \left(\text{because } \sum_{0\leq i\leq P-1} c_i = s = \sum_{0\leq i\leq P-1} c'_i\right) \\
c_{i_1}+1 \leq c'_{i_1} & \\
t_{i_1}(c_{i_1}+1) \leq t_{i_1}c'_{i_1} & \\
m \leq m' \quad & \text{(by definition of } m \text{ and } m')
\end{aligned}$$

which contradicts the non-optimality of the original allocation. ■

There remains to build allocations satisfying Condition (1). The following algorithm suffices:

- For the chunk size $s=0$, take the optimal allocation $(0,0,\ldots,0)$.
- To derive an allocation $\mathcal{C}'$ verifying equation (1) with chunk size s from an allocation $\mathcal{C}$ verifying (1) with chunk size $s-1$, add 1 to a well-chosen c_j, one that verifies

$$t_j(c_j+1) = \min_{0\leq i\leq P-1} t_i(c_i+1). \tag{2}$$

  In other words, let $c'_i=c_i$ for $0\leq i\leq P-1, i\neq j$, and $c'_j=c_j+1$.

Lemma 5 *This algorithm is correct.*

Proof We have to prove that allocation $\mathcal{C}'$, given by the algorithm, verifies Equation (1).

Since allocation $\mathcal{C}$ verifies equation (1), we have $t_i c_i \leq m \leq t_j(c_j+1)$. By definition of j from Equation (2), we have

$$m' = \max_{0\leq i\leq P-1} t_i c'_i = \max\left(t_j(c_j+1), \max_{1\leq i\leq q, i\neq j} t_i c_i\right) = t_j c'_j.$$

We then have $t_j c'_j \leq m' \leq t_j(c'_j+1)$ and

$$\begin{aligned}\forall i \neq j, 1 \leq i \leq q,\\ \mathbf{t_i c'_i} = t_i c_i \leq m \leq \mathbf{m'} \leq t_j c'_j = \min_{0\leq i\leq P-1} t_i(c_i+1) \leq t_i(c_i+1) = \mathbf{t_i(c'_i+1)},\end{aligned}$$

so the resulting allocation does verify Equation (1). ■

To summarize, we have built an algorithm to compute "good" block sizes for the heuristic allocation by blocks of columns. One selects an upper bound on the chunk size, and our algorithm returns the best block sizes, according to our *cost* function, with respect to this bound.

The complexity of this algorithm is $O(Ps)$, where P is the number of processors and s, the upper bound on the chunk size. Indeed, the algorithm consists of s steps where one computes a minimum over the processors. This low complexity allows us to perform the computation of the best allocation at runtime.

A Small Example. To understand how the algorithm works, we present a small example with $P = 3$, $t_0 = 3$, $t_1 = 5$, and $t_2 = 8$. In Table 2, we report the best allocations found by the algorithm up to $s = 7$. The entry "Selected j" denotes the value of j that is chosen to build the next allocation. Note that the cost of the allocations is not a decreasing function of s. If we allow chunks of size not greater than 7, the best solution is obtained with the chunk $(3, 2, 1)$ of size 6.

Chunk Size	c_0	c_1	c_2	Cost	Selected j
0	0	0	0		0
1	1	0	0	3	1
2	1	1	0	2.5	0
3	2	1	0	2	2
4	2	1	1	2	0
5	3	1	1	1.8	1
6	3	2	1	1.67	0
7	4	2	1	1.71	

Table 2: Running the algorithm with 3 processors: $t_0 = 3$, $t_1 = 5$, and $t_2 = 8$.

Finally, we point out that our modified heuristic "converges" to the original asymptotically optimal heuristic. For a chunk of size $C = L \times \sum_{i=0}^{P-1} \frac{1}{t_i}$, where $L = \text{lcm}(t_0, t_1, \ldots, t_{P-1})$ columns, we obtain the optimal cost

$$cost_{opt} = \frac{L}{C} = \left(\sum_{0\leq i\leq P-1} \frac{1}{t_i}\right)^{-1},$$

which is the inverse of the harmonic mean of the execution times divided by the number of processors.

4.2.3 Chunk size choice

Choosing a chunk size is not easy. There are several constraints: it should be large enough to have a good cost but small enough to be insensitive to the domain boundaries overhead. Between these two bounds, no clear choice appears. Actually, the experiments do not help us here as can be seen in figure 6.

4.2.4 Remark on the Multidimensional Approximation Problem

Indeed, our algorithm solves the multidimensional approximation problem where one wants to approximate some real numbers with rationals sharing the same denominator. Many algorithms exist to solve this problem (see [5], for example), but these algorithms focus on finding a "best approximation" with respect to the real numbers where we only want "good" small numbers approximations. Most (at least all the ones we tried) of these algorithms miss the approximations we want. They are too fast to be controled by a bound on the chunk size!

4.3 MPI Experiments

We report several experiments on the network of workstations presented in Section 4.1. After comments on the experiments, we focus on cyclic and block-cyclic allocations and then on our modified heuristics.

4.3.1 General Remarks

We study different columnwise allocations on the heterogeneous network of workstations presented in Section 4.1. Our simulation program is written in C using the MPI library for communication. It is not an actual tiling program, but it simulates such behavior: we have not inserted the code required to deal with the boundaries of the computation domain. Actually, our code only simulates the communications generated by a tiling, it does fake computations (hence, no data allocation). The tiling is assumed given. Our aim is not to find the "best" tiling. The tile domain has 100 rows and a number of columns varying from 200 to 1000 by steps of 100. An array of doubles of size the square root of the tile area is communicated for each communication (we assume here that the computation volume is proportional to the tile area while the communication volume is proportional to its square root).

The actual communication network is a coax type Ethernet network. It can be considered as a bus, not as a point-to-point connection ring; hence our model for communication is not fully correct. However, this configuration has little impact on the results, which correspond well to the theoretical conditions.

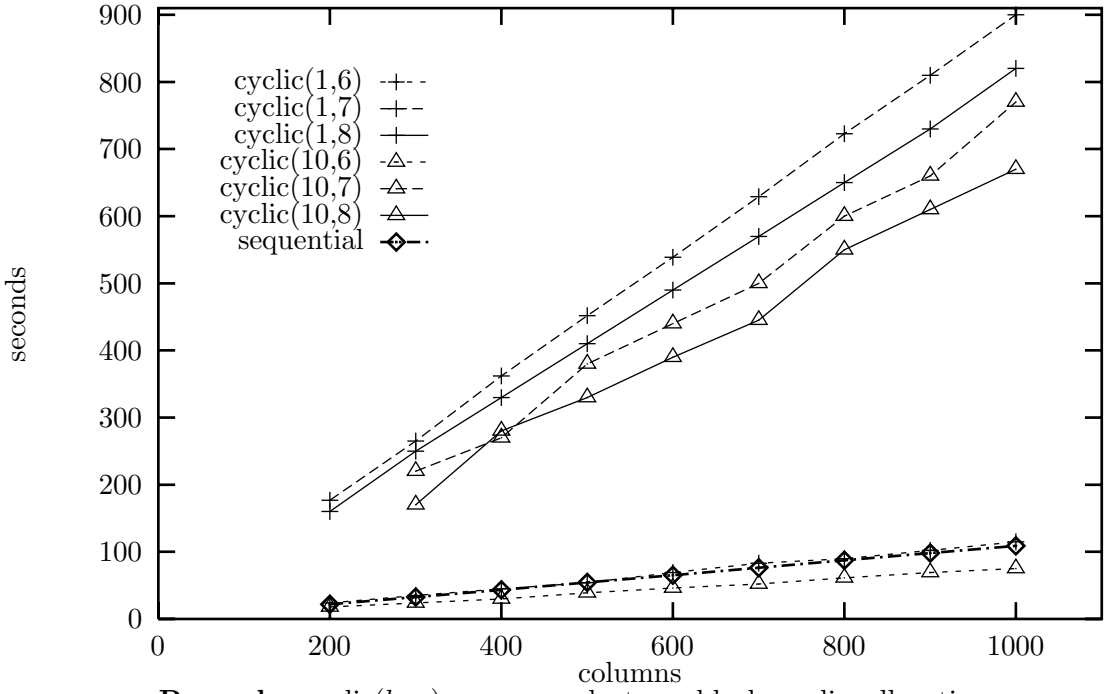
As already pointed out, the workstations we use are multiple-user workstations. Although our simulations were made at times when the workstations were not supposed to be used by anybody else, the load may vary. The timings reported in the figures are the average of several measures from which aberrant data have been suppressed.

In Figures 4 and 6, we show for reference the sequential time as measured on the fastest machine, namely, "nala".

4.3.2 Cyclic Allocations

We have experimented with cyclic allocations on the 6 fastest machines, on the 7 fastest machines, and on all 8 machines. Because cyclic allocation is optimal when all processors have the same

speed, this will be a reference for other simulations. We have also tested a block cyclic allocation with block size equal to 10, in order to see whether the reduced amount of communication helps. Figure 4 presents the results[10] for these 6 allocations (3 purely cyclic allocations using 6, 7, and 8 machines, and 3 block-cyclic allocations).



Remark cyclic(b,m) corresponds to a block cyclic allocation with block size b, using the m fastest machines of Table 1.

Figure 4: Experimenting with cyclic and block-cyclic allocations.

We comment on the results of Figure 4 as follows:

- With the same number of machines, a block size of 10 is better than a block size of 1 (pure cyclic).
- With the same block size, adding a single slow machine is disastrous, and adding the second one only slightly improve the disastrous performances.
- Overall, only the block cyclic allocation with block size 10 and using the 6 fastest machines gives some speedup over the sequential execution.

We conclude that cyclic allocations are not efficient when the computing speeds of the available machines are very different. For the sake of completeness, we show in Figure 5 the execution times obtained for the same domain (100 rows and 1000 columns) and the 6 fastest machines, for block cyclic allocations with different block sizes. We see that the block-size as a small impact on the performances, which corresponds well to the theory: all cyclic allocations have the same cost.

[10]Some results are not available for 200 columns because the chunk size is too large.

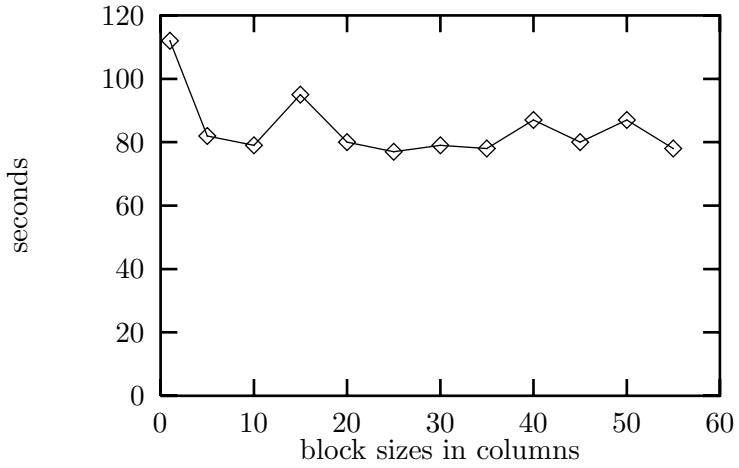


Figure 5: Cyclic allocations with different block sizes.

4.3.3 Using our modified heuristic

Let us now consider our heuristics. In Table 3, we show the block sizes computed by the algorithm described in Section 4.2 for different upper bounds of the chunk size. The best allocation computed with bound u is denoted as $\mathcal{C}_u$.

	nala	bluegrass	dancer	donner	vixen	rudolph	zazu	simba	*cost*	chunk
$\mathcal{C}_{25}$	7	3	2	2	2	2	0	0	4.44	18
$\mathcal{C}_{50}$	15	6	5	5	4	4	0	0	4.23	39
$\mathcal{C}_{100}$	33	14	11	11	9	9	0	0	4.18	87
$\mathcal{C}_{150}$	52	22	17	17	15	14	1	1	4.12	139

Table 3: Block sizes for different chunk size bounds.

The time needed to compute these allocations is completely negligible with respect to the computation times (a few milliseconds versus several seconds).

Figure 6 presents the results for these allocations. Here are some comments:

- Each of the allocations computed by our heuristic is superior to the best block-cyclic allocation.
- The more precise the allocation, the better the results.
- For 1000 columns and allocation $\mathcal{C}_{150}$, we obtain a speedup of 2.2 (and 2.1 for allocation $\mathcal{C}_{50}$), which is very satisfying (see below).

The optimal cost for our workstation network is $cost_{opt} = \frac{L}{C} = \frac{34{,}560{,}240}{8{,}469{,}789} = 4.08$. Note that $cost(\mathcal{C}_{150}) = 4.12$ is very close to the optimal cost. The peak theoretical speedup is equal to $\frac{\min_i t_i}{cost_{opt}} = 2.7$. For 1000 columns, we obtain a speedup equal to 2.2 for $\mathcal{C}_{150}$. This is satisfying considering that we have here only 7 chunks, so that side effects still play an important role. Note also that the peak theoretical speedup has been computed by neglecting all the dependencies in the computation and all the communications overhead. Hence, obtaining a twofold speedup with 8 machines of very different speeds is not a bad result at all!

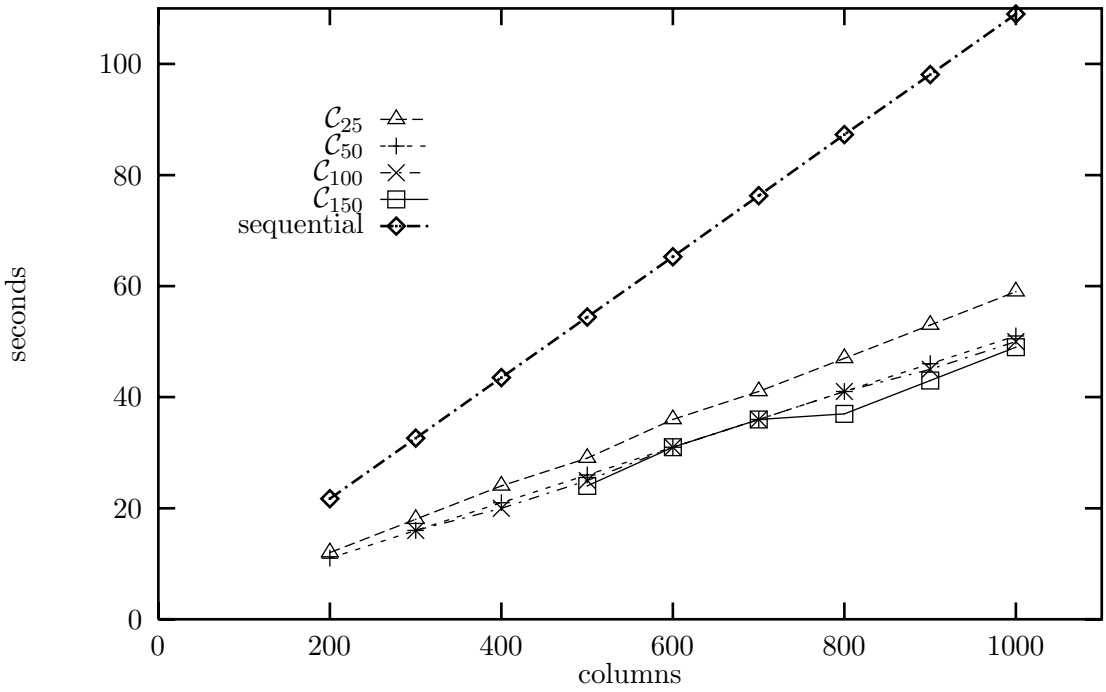


Figure 6: Experimenting with our modified heuristics.

5 Conclusion

In this paper, we have extended tiling techniques to deal with heterogeneous computing platforms. Such platforms are likely to play an important role in the near future. We have introduced an asymptotically optimal columnwise allocation of tiles to processors. We have modified this heuristic to allocate column chunks of reasonable size, and we have reported successful experiments on a network of workstations. The practical significance of the modified heuristics should be emphasized: processor speeds may be inaccurately known, but allocating small but well-balanced chunks turns out to be quite successful.

Heterogeneous platforms are ubiquitous in computer science departments and companies. The development of our new tiling techniques allows for the efficient use of older computational resources *in addition to* newer available systems.

References

[1] A. Agarwal, D.A. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans. Parallel Distributed Systems*, 6(9):943–962, 1995.

[2] Rumen Andonov, Hafid Bourzoufi, and Sanjay Rajopadhye. Two-dimensional orthogonal tiling: from theory to practice. In *International Conference on High Performance Computing (HiPC)*, pages 225–231, Trivandrum, India, 1996. IEEE Computer Society Press.

[3] Rumen Andonov and Sanjay Rajopadhye. Optimal tiling of two-dimensional uniform recurrences. *Journal of Parallel and Distributed Computing*, to appear. Available as Technical Report LIMAV-RR 97-1, `http://www.univ-valenciennes.fr/limav/andonov`.

[4] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (pen)-ultimate tiling? *Integration, the VLSI Journal*, 17:33–51, 1994.

[5] A. J. Brentjes. *Multi-dimensional continued fraction algorithms*. Mathematisch Centrum, Amsterdam, 1981.

[6] Pierre-Yves Calland and Tanguy Risset. Precise tiling for uniform loop nests. In P. Cappello et al., editors, *Application Specific Array Processors ASAP 95*, pages 330–337. IEEE Computer Society Press, 1995.

[7] P.Y. Calland, J. Dongarra, and Y. Robert. Tiling with limited resources. In L. Thiele, J. Fortes, K. Vissers, V. Taylor, T. Noll, and J. Teich, editors, *Application Specific Systems, Achitectures, and Processors, ASAP'97*, pages 229–238. IEEE Computer Society Press, 1997.

[8] Y-S. Chen, S-D. Wang, and C-M. Wang. Tiling nested loops into maximal rectangular blocks. *Journal of Parallel and Distributed Computing*, 35(2):108–120, 1996.

[9] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. *Computer Physics Communications*, 97:1–15, 1996. (also LAPACK Working Note #95).

[10] Ph. Chretienne. Task scheduling over distributed memory machines. In M. Cosnard, P. Quinton, M. Raynal, and Y. Robert, editors, *Parallel and Distributed Algorithms*, pages 165–176. North Holland, 1989.

[11] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 1997. Special issue, to appear. Also available as Tech. Rep. LIP, ENS-Lyon, RR96-34.

[12] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, 1995.

[13] K. Högstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Principles of Programming Languages*, pages 160–173. ACM Press, 1997. Extended version available as Technical Report UCSD-CS96-489.

[14] François Irigoin and Rémy Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.

[15] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, January 1997.

[16] Naraig Manjikian and Tarek S. Abdelrahman. Scheduling of wavefront parallelism on scalable shared memory multiprocessor. In *Proceedings of the International Conference on Parallel Processing ICPP 96*. CRC Press, 1996.

[17] H. Ohta, Y. Saito, M. Kainaga, and H. Ono. Optimal tile size adjustment in compiling general DOACROSS loop nests. In *1995 International Conference on Supercomputing*, pages 270–279. ACM Press, 1995.

[18] Peter Pacheco. *Parallel programming with MPI.* Morgan Kaufmann, 1997.

[19] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.

[20] Robert Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical Report 90-38, The University of Tennessee, Knoxville, TN, August 1990.

[21] S. Sharma, C.-H. Huang, and P. Sadayappan. On data dependence analysis for compiling programs on distributed-memory machines. *ACM Sigplan Notices*, 28(1), January 1993. Extended Abstract.

[22] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44. ACM Press, 1991.

[23] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4):452–471, October 1991.

Annexe D

High Level Parallelization of a 3D Electromagnetic Simulation Code with Irregular Communication Patterns

High Level Parallelization of a 3D Electromagnetic Simulation Code With Irregular Communication Patterns

Emmanuel Cagniot[1]*, Thomas Brandes[2], Jean-Luc Dekeyser[1], Francis Piriou[3], Pierre Boulet[1] and Stéphance Clénet[3]

[1] Laboratoire d'Informatique Fondamentale de Lille (LIFL)
U.S.T.L. Cité Scientifique, F-59655 Villeneuve d'Ascq Cedex, France
[2] Institute for Algorithms and Scientific Computing (SCAI)
German National Research Center for Information Technology (GMD)
Schloß Birlinghoven, D-53754 St. Augustin, Germany
[3] Laboratoire d'Electrotechnique et d'Electronique de Puissance (L2EP)
U.S.T.L. Cité Scientifique, F-59655 Villeneuve d'Ascq Cedex, France

Abstract. 3D simulation in electrical engineering is based on recent research work (Whitney's elements, auto-gauged formulations, discretization of the source terms) and it results in complex and irregular codes. Generally, explicit message passing is used to parallelize this kind of applications requiring tedious and error prone low level coding of complex communication schedules to deal with irregularity. In this paper, we focus on a high level approach using the data-parallel language High Performance Fortran. It allows both an easier maintenance and a higher software productivity for electrical engineers. Though HPF was initially conceived for regular applications, it can be successfully used for irregular applications when using an unstructured communication library that deals with indirect data accesses.

Topics: cellular automata and physics.

1 Introduction

[1] Electrical engineering consists of designing electrical devices like iron core coils (example 1) or permanent magnet machines (example 2) (see Fig. 1). As prototypes can be expensive, numerical simulation is a good solution to reduce development costs. It allows to predict device performance from physical design information. Accurate simulations require 3D models, inducing high storage capacity and CPU power needs. As computation times can be very important, parallel computers are well suited for these models.

* Corresponding author, e-mail: `cagniot@lifl.fr`, Tel: +33-0320-434730, Fax: +33-0320-436566

[1] Candidate to the best student paper award

3D Electromagnetic problem modeling is based on Maxwell's equations. Generally, the resolution of these partial differential equations requires numerical methods. They transform the differential equations into an algebraic equation system whose solution gives an approximation of the exact solution. The space discretization and the time discretization can be done respectively by the finite element method (FEM) and by the finite difference method (FDM). The finite elements used are Whitney's elements (nodes, edges, facets and volumes) [1]: they allow to keep the properties of continuity of the field at the discrete level. In function of the studied problem, the equation system can be linear or non-linear.

As for many other engineering applications, FEM codes use irregular data structures such as sparse matrices where data access is done via indirect addressing. Therefore, finding data distributions that provide both high data locality and good load balancing is difficult. Generally, parallel versions of these codes use explicit message passing.

In this paper, we focus on a data-parallel approach with High Performance Fortran (HPF). Three reasons can justify this choice. First, a high level programming language is more convenient than explicit message passing for electrical engineers. It allows both an easier maintenance and a higher software productivity. Second, libraries for optimizing unstructured communications in codes with indirect addressing exist [4]. The basic idea is that these codes reuse several times the same communication patterns. Therefore, it is possible to compute these patterns one time and to reuse them if possible. Third, the programmer can further optimize its code by mixing special manual data placements and simple HPF distributions to provide both high data locality and good load balancing.

Section 2 presents the main features of our electromagnetic code. Section 3 presents the HPF parallelization of the magnetostatic part of this code. Section 4 presents the unstructured communication library we used to efficiently parallelize the preconditioned conjugate gradient method. Section 5 presents the results we obtained on a SGI Origin and an IBM SP2. Conclusion and future works are given in Section 6.

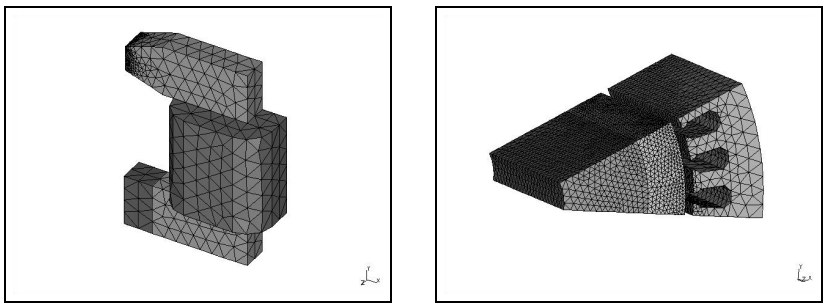


Fig. 1. An iron core coil and a permanent magnet machine.

2 The Code

The L2EP at the University of Lille has developed a 3D FORTRAN 77 code for modeling magnetostatic (time-independent) and magnetodynamic (time-dependent) problems [5]. The magnetostatic part uses formulations in terms of scalar or vector potentials where the unknowns of the problem are respectively the nodes and the edges of the grid. The magnetodynamic part uses hybrid formulations that mix scalar and vector potentials.

The great particularity of this code is the computation of the source terms using the *tree* technique. Generally, in the electromagnetic domain, gauges are required to obtain an unique solution. They can be added into the partial differential equations but this solution involves additional computations during the discretization step. Another solution results from a recent research work [7]. A formulation is said compatible when all its entities have been discretized onto Whitney's elements. This work has shown that problems are auto-gauged when both a compatible formulation and an iterative solver are used. Therefore, to obtain a compatible formulation, source terms must be discretized using the *tree* technique, actually a graph algorithm.

As the meshing tool only produces nodes and elements, the edges and the facets must be explicitly computed. The time discretization is done with Euler's implicit algorithm. The FEM discretization of a non-linear problem results in an iterative loop of resolutions of linear equation systems. Two non-linear methods are used: Newton-Raphson's method and the fixed-point method. Their utilization is formulation-dependent. The linear equation systems are either symmetric positive definite or symmetric semi-positive definite. Therefore, the preconditioned conjugate gradient method (PCG) is used. The preconditioner results from an incomplete factorization of Crout.

The overall structure of this code is as follows:

```
1. define the media, the inductors, the magnets, the boundary
   conditions, etc.
2. read the input file and compute the edges and the facets.
3. compute the source terms.
4. check the boundary conditions and number the unknowns.
5. time loop:
   5.1. create the Compressed Spare Row (CSR) representation of the
        equation system.
   5.2. non-linear loop:
        5.2.1. assembly loop:
                 - compute and store the contribution of all the
                   elements in the equation system.
        5.2.2. compute the preconditioning matrix.
        5.2.3. solving loop:
                 - iterate preconditioned conjugate gradient.
```

This structure shows that computation times can be very important when we use time-dependent formulations in the non-linear case.

The matrices for the equation systems are represented by the Compressed Sparse Row (CSR) format as shown in Fig. 2. As they are symmetric, the upper triangular part is not explicitly availabe to save memory.

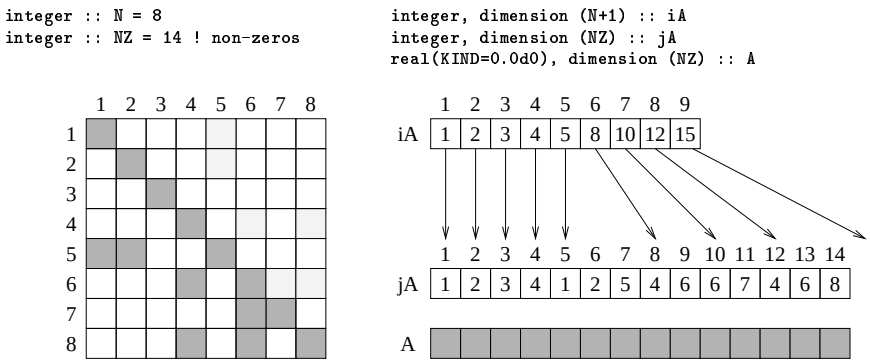


Fig. 2. Compress sparse row format of a matrix.

3 HPF Parallelization of the Magnetostatic Part

For software engineering reasons, the magnetostatic part of the code has been ported to Fortran 90 and it has been optimized. The Fortran 90 version makes extensive use of modules (one is devoted to the data structures), derived data types and array operations. It consists of about 9000 lines of code. This new version has been converted to HPF considering only the parallelization of the assembling and the solver that take about 80% of the whole execution time in the linear case. All other parts of the code remained serial.

In a first step, all the data structures including the whole equation system have been replicated on all the processors. Each of them has to perform the same computations as the others until the CSR structure of the equation system is created. Some small code changes were necessary to reduce the number of broadcasts implied by the serial I/O operations.

The assembling loop is a parallel loop over all the elements, each element contributing some entries to the equation system (see Fig. 3). Though one unknown can belong to more than one element, and so two elements might add their contribution at the same position, the addition is assumed to be an associative and commutative operation. The order in which the loop iterations are executed does not change the final result (except for possible round-off errors). Therefore it is safe to use the `INDEPENDENT` directive of HPF together with the `REDUCTION` clause for the arrays containing the values of `A` (matrix) and `B` (right hand side). A one-dimensional block-distributed template, whose size is given by

the number of elements, has been used to specify the work distribution of this loop via the `ON HOME` clause.

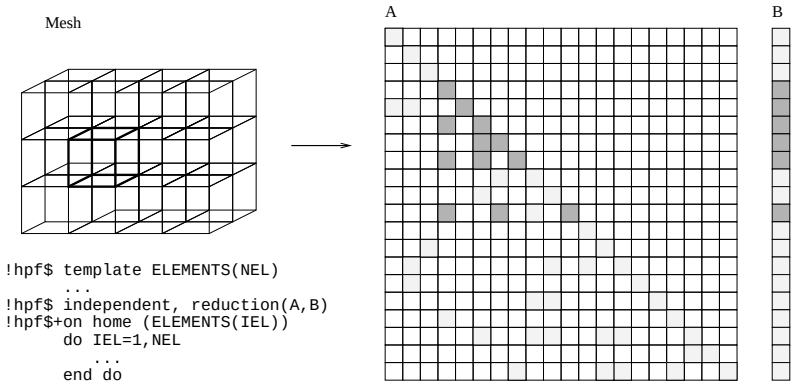


Fig. 3. Assembling the contributions for the equation system.

In a second step, the equation system and the preconditioning matrix have been general block distributed in such a way that all the information of one row $A(i,:)$ resides on the same processor as vector element $z(i)$. By the `RESIDENT` directive, the HPF compiler gets the information that avoids unnecessary checks and synchronizations for accesses to these matrices.

The algorithm for the preconditioned conjugate gradient method is dominated by the matrix-vector multiplications and by the forward/backward substitutions required during the preconditioning steps.

Due to the dependences in the computations, the incomplete factorization of Crout before the CG iterations and the forward/backward substitution per iteration are not parallel at all. The factorization has been replaced with an incomplete block factorization of Crout that takes only the local blocks into account forgetting the coupling elements. By this way, the factorization and the forward/backward substitution do not require any communication. As the corresponding preconditioning matrix becomes less accurate, the number of iterations increases with the number of blocks (processors). But the overhead of more iterations is less than the extra work and communication needed otherwise. HPF provides `LOCAL` extrinsic routines where every processor sees only the local part of the data. This concept is well suited to restrict the factorization and the forward/backward substitution to the local blocks where local dependences in one block are respected and global dependences between the blocks are ignored.

The matrix-vector multiplication uses halos [3] (see next section) that provide an image of the non-local part of a vector on each processor, and the communication schedule needed for the related updating operations. The computation

time for the halo and its communication schedule is amortized over the number of iterations.

For running our HPF code on parallel machines we used the Adaptor HPF compilation system of GMD [2]. Only Adaptor supported the features needed for the application (`ON` clause, general block distributions, `RESIDENT` directive, `REDUCTION` directive, `LOCAL` routines, halos and reuse of communication schedules). By means of a source-to-source transformation, Adaptor translates the data parallel HPF program to an equivalent SPMD program (single program, multiple data) in Fortran where this Fortran program is compiled with a native Fortran compiler. The generated SPMD program contains a lot of calls to the Adaptor specific HPF runtime system, called DALIB (distributed array library) that implements also the functionality needed for halos (see Figure 4).

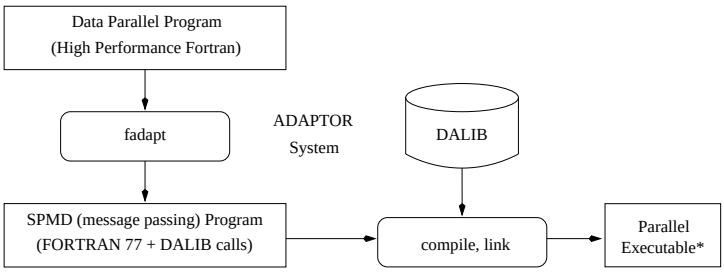


Fig. 4. Schematic view of Adaptor HPF compilation.

4 Unstructured Communication Library Approach

The Adaptor HPF compilation system provides a library that supports shadow edges (ghost points) for unstructured applications using indirect addressing, also called halos [4]. A halo provides additionally allocated local memory to keep on one processor also non-local values of the data that is accessed by the processor and a communication schedule that reflects the communication pattern between the processors to update these non-local copies. As the size of the halo and the communication schedule depend on the values of the indirection array, they can only be computed at runtime.

The use of halos (overlapping, shadow points) is common manual practice in message passing programs for the parallelization of unstructured scientific applications. But the calculation of halo sizes and communication schedules is tedious and error prone and requires a lot of additional coding. Up to now, commercial HPF compilers do not support halos. The idea of halos has already been followed within the HPF+ project and implemented in the Vienna Fortran Compiler [3] where the use of halos is supported by additional language features instead of a library.

Fig. 5 shows the use of halos for the matrix-vector multiplication $B = A \times X$. The vectors X and B are block distributed among the available processors. The matrix A is distributed in such a way that one row $A(i,:)$ is owned by the same processor that owns $B(i)$. For the matrix-vector multiplication, every processor needs all elements $X(j)$ for the non-zero entries $A(i,j)$ owned by it. Though most of these values might be available, there remain some non-local accesses. The halo will contain these non-local values after an update operation using the corresponding halo schedule.

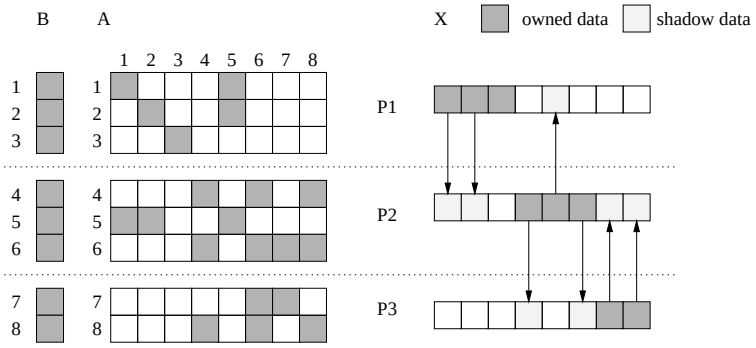


Fig. 5. Distribution of matrix with halo nodes for the vector.

As mentioned before, the upper triangular of `A` part is not explicitly available. For $j > i$ the elements $A(i,j)$ must be accessed via $A(j,i)$. To avoid the communication of matrix elements the values of $A(i,j) * X(j)$ are not computed by the owner of $B(i)$ but by the owner of $X(j)$ as here $A(j,i)$ is local. Therefore we have an additional communication step to reduce the non-local contributions of B. But for this unstructured reduction we can use the same halo structure for the vector B.

For the calculation of the halo, we had to provide the halo array, which is the indirectly addressed array, and the halo indexes that are used for the indirect addressing in the distributed dimension. In our case, the halo arrays are the vectors X and B, and the halo indexes are given by the integer array `jA` containg the column indices used in the CSR format. Beside the insertion of some subroutine calls for the HPF halo library, we had only to insert some HPF directives for the parallelization of the loop implementing the matrix-vector multiplication.

The calculation of the halo structure is rather expensive. But we can use the same halo structure for the update of the non-local copies of vector X and for the reduction of the non-local contributions of vector B. Furthermore, this halo can be reused for all iterations of the iteration loop in the solver as the structure of the matrix and therefore the halo indexes do not change.

5 Results

Table 1 shows the characteristics of the problems of Fig. 1 in the case of a vector potential formulation. These results have been obtained for the test cases of Table 1 on a SGI Origin and on an IBM SP2 in the linear case (in the magnetodynamic case, each column would represent results associated with one time increment). Their quality has been measured by two ways: graphically with the dumped files and numerically with the computed magnetic energies. All the computed solutions are the same.

	example 1	example 2
nodes	8059	25730
edges	51356	169942
facets	84620	284669
elements	41322	140456
unknowns	49480	162939
non-zero entries	412255	1382596

Table 1. Test cases of the code

Table 2 presents results for example 1 on the SGI Origin with up to four processors. Both, the assembling and the solver scale well for a small number of processors.

	NP = 1	NP = 2	NP = 3	NP = 4
assembling	12.87 s	6.54 s	4.63 s	3.61 s
solver	30.47 s	18.89 s	9.74 s	8.80 s
iterations	225	260	213	237

Table 2. Results for example 1, SGI Origin

Table 3 presents results for example 1 on the IBM SP2 with up to 16 processors. In the assembling loop the computational work for one element is so high that the parallelization still gives good speed-ups for more processors even if the reduction overhead increases with the number of processors. The scalability of the solver is limited as the data distribution has not been optimized for data locality yet. On the other hand, the number of solver iterations does not increase dramatically with the number of processors and the higher inaccuracy.

Comparison between Table 2 and Table 3 shows that for the two parallel machines the numbers of iterations are different for a same number of processors. This can be explained by the application of different aggressive optimizations of the native Fortran compilers (optimization level 3) that may generate (even slightly) different results.

	NP = 1	NP = 2	NP = 4	NP = 8	NP = 16
assembling	32.70 s	16.77 s	8.98 s	5.27 s	3.47 s
solver	48.45 s	36.99 s	23.51 s	14.45 s	10.59 s
iterations	224	247	257	247	258

Table 3. Results for example 1, IBM SP2

	NP = 1	NP = 2	NP = 3	NP = 4
assembling	43.77 s	22.71 s	16.03 s	12.95 s
solver	625.06 s	160.84 s	116.63 s	93.25 s
iterations	1174	464	509	505
time/iter	532 ms	347 ms	229 ms	165 ms

Table 4. Results for example 2, SGI Origin

Table 4 presents results for example 2 on the SGI Origin. The assembling and one iteration of the solver scale well. Regarding the number of solver iterations, the results are surprising. For its explanation we must interest ourselves to the numbering of the unknowns which plays an important role in the conditioning of the equation system. In our case, the meshing tool sorts its elements by volumes, every volume corresponding to a magnetic permeability (air, iron, etc.). To avoid large jumps of coefficients in the matrix, the unknowns are numbered by scanning the list of elements. Therefore, to every volume of the grid corresponds a homogeneous block of the matrix. The conditioning of this matrix is directly linked to the uniformity of the magnetic permeability of these different blocks. When this uniformity is poor, block preconditioners resulting from domain decomposition methods can be used. By preconditioning each block independently, they allow to bypass the problem. In our case, the incomplete block factorization of Crout has led to the same result. For its verification we have re-sorted the elements of the grid by merging volumes with the same magnetic permeability. As a result, we have divided the number of PCG iterations by two in the Fortran 90 program.

6 Conclusions and Future Work

This HPF version achieves acceptable speed-ups for smaller number of processors. According to the few number of HPF directive added in the Fortran 90 code, it is a very cheap solution that allows both an easier maintenance and a higher software productivity for electrical engineers, compared to an explicit message passing version. This was its main objective. Results obtained for real electrical engineering problems show that HPF can deal efficiently with irregular codes when using an irregular communication library.

In order to improve data-locality and to reduce memory consumption, the next HPF version will use a conjugate gradient method where the preconditioner will be based on domain decomposition using a Schur complement method [6].

In the final step of this work we will add the magnetodynamic formulations.

References

1. A.Bossavit. A rational for edge elements in 3d fields computations. In *IEEE Trans. Mag.*, volume 24, pages 74–79, january 1988.
2. ADAPTOR. High Performance Fortran Compilation System. WWW documentation, Institute for Algorithms and Scientific Computing (SCAI, GMD), 1999. `http://www.gmd.de/SCAI/lab/adaptor`.
3. S. Benkner. Optimizing Irregular HPF Applications Using Halos. In Rolim, J. et al., editor, *Parallel and Distributed Processing, Proceedings of IPPS/SPDP Workshops*, volume 1586 of *Lecture Notes in Computer Science*, pages 1015–1024, San Juan, Puerto Rico, USA, April 1999. Springer.
4. T. Brandes. HPF Library, Language and Compiler Support for Shadow Edges in Data Parallel Irregular Computations. In *CPC 2000, 8th International Workshop on Compilers for Parallel Computers, Aussois, France, January 4-7*, pages 21–34, January 2000.
5. Y.Le Menach, S.Clénet, and F.Piriou. Determination and utilization of the source field in 3d magnetostatic problems. In *IEEE Trans. Mag.*, volume 34, pages 2509–2512, 1998.
6. Barry F. Smith, Petter E. Bjørstad, and William Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 1996.
7. Z.Ren. Influence of R.H.S. on the convergence behaviour of curl-curl equation. In *IEEE Trans. Mag.*, volume 32, pages 655–658, 1996.

Annexe E

Towards Distributed Process Networks with CORBA

Towards Distributed Process Networks with CORBA

Abdelkader Amar, Pierre Boulet and Jean-Luc Dekeyser
Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille
Cité Scientifique, Bat. M3, 59655 Villeneuve d'Ascq cedex France
{Abdelkader.Amar,Pierre.Boulet,Jean-Luc.Dekeyser}@lifl.fr

May 15, 2002

Abstract

Process networks is a widely used model to describe highly concurrent applications. We present here a distributed implementation of a slightly restricted process network model realized using the CORBA middleware. This implementation allows the non computer science specialist to easily program heterogeneous meta-applications based on an assembly of components communicating through FIFO queues.

1 Introduction

Many parallel applications, specially in the signal processing domain can be modeled with Kahn process networks [5, 6]. In this model, processes communicate via unbounded first-in first-out (FIFO) queues exclusively. This model has a dataflow flavor and can express a high degree of concurrency which makes it particularly well suited to model intensive signal processing applications or complex scientific applications. This model makes no assumption on the computation load of the different processes and thus is heterogeneous by nature.

Distributed architectures provide an attractive alternative to supercomputers in terms of computation power and cost to execute such complex and computation intensive applications. The two main weak points of these architectures are their communication capabilities (relatively high latency) and often the heterogeneity of their hardware.

We present in this paper a distributed implementation of a subcase of the process network model on heterogeneous distributed hardware. The different processing power of the connected computers is a good support for the different computation needs of the networked processes. We have chosen to use the Common Object Request Broker Architecture (CORBA) [10]

middleware to handle the communications for its interoperability properties. Indeed, each process of the process network can be written in a different language and run on a different hardware, provided that these are supported by the chosen Object Request Broker (ORB).

The concept of dynamicity (migration and replacement of components) seems essential to us because of our application domain, namely scientific computing and more specifically intensive digital signal processing. Indeed, we target long running distributed applications. Actually these applications may run indefinitely, taking an infinite data stream as input. Allowing to improve the code or the hardware while the application is running is an important goal to us.

The rest of this paper is organized as follows. In section 2, we present the model we use and motivate our approach. Section 3 gives a description of the basic building block of our distributed process networks, namely distributed FIFO queues. The components (implementing the distributed processes) are described in section 4 and section 5 explains how to build a dynamic component graph. We then detail an application in section 6 and we outline our conclusions and plans for the future work in section 7.

2 Model and Implementation Choices

2.1 Process Networks

The process network model has been proposed by Kahn and MacQueen [5, 6] to easily express concurrent applications. Processes communicate only through unidirectional FIFO queues. A process is blocked when it attempts to read from an empty queue. A process can be seen as a mapping from its input streams to its output streams. The number of tokens produced and their values are completely determined by the definition of the network and do not depend on the scheduling of the processes. Thus the process network model is called determinate.

The choice of a scheduling of a process network only determines if the computation terminates and the sizes of the FIFO queues. Some networks do not allow a bounded execution. Parks [11] studies these scheduling problems in depth. He compares three classes of dynamic scheduling: data-driven, demand-driven or a combination of both with respect to two requirements:

1. Complete execution (the application should execute completely, in particular if the program is non-terminating, it should execute forever).

2. Bounded execution (only a bounded number of tokens should accumulate on any of the queues).

These two properties are shown undecidable by Buck [3] on boolean dataflow graph which are a special case of process networks. Thus they are also

undecidable for the general case of process networks. Data-driven schedules respect the first requirement, but not always the second one. Demand-driven schedules may cause artificial deadlocks. A combination of the two is proposed by Parks [11] to allow a complete, unbounded execution of process networks when possible.

Several implementations of process networks are used for different purposes: for heterogeneous modeling with PtolemyII [8], for signal processing application modeling with YAPI [4] and for metacomputing in the domain of Geographical Information Systems with Jade/PAGIS [12, 13]. Only the Jade/PAGIS implementation is distributed. The PtolemyII and YAPI implementations use threads to represent the different processes. Furthermore, none of these implementations allow the coupling of processes written in different languages.

2.2 Restrictions on the Process Network Model

One of our goals is to hide the complexity of building distributed applications to the programmer, typically a non computer science specialist. The user should just have to write his domain specific processing functions and their prototypes and a code generator should take these descriptions to produce distributed code, effectively hiding all the details of communication and synchronization, thus achieving a high level of transparency.

To reach this goal, we make some restrictions on the processes in our implementation. These restrictions simplify the scheduling of the process network while retaining the expressing power we need for our applications. In our implementation processes are functional, meaning that they work on the following schema:

1. optional initialization phase where the process can write to its output queues (to allow cyclic process networks),
2. infinite loop:
   (a) read the inputs form the input queues,
   (b) compute the outputs,
   (c) store the results in the output queues.

When a process reads from an input FIFO queue, it can read (*get*) several tokens at a time and remove (*take*) another number of tokens. This is a common extension to the process network model that can be expressed easily by the original model.

These restrictions allow us to easily represent complex applications based on an assembly of components. This coarse grain view of the application is better suited to a performant execution on a network of computers than a finer grain view which generates too much communications. This does not

forbid the component to be parallel and to execute on a parallel computer. Basically, this model is well suited to model computation intensive meta-applications.

2.3 Distribution

The distribution is done in a different way than in Jade [12]. To avoid a central point of failure and unnecessary copies, the communications are handled by the components themselves through half FIFO queues. These queues implement the blocking read needed by the process network model but, as they are bounded, the write may block also if the queue is full. This can create deadlocks[1], but as the execution is fully distributed, deadlock detection is difficult. We propose that the user selects the maximum size of the FIFO queues at creation time as he has an idea of the expected behavior of his application and of the resource usage he is ready to pay to run this application. These queues run asynchronously with the process computation so as to mask the network overhead and allow the vectorization of the communications. See section 3 for more details on the implementation of the distributed FIFO queues and the hybrid data-driven, demand-driven communication protocol.

The fact that the FIFO queues are bounded also allows for an incremental development where computation can start even if the application is not complete. When all the output queues are full, the computation is suspended and can restart as soon as a consuming component is attached to the not-yet-connected output queues. More details about the dynamicity of the component graph is given in section 5.

2.4 Distributed Heterogeneous Components with CORBA

To deal with the heterogeneity of the connected components of the application, we have chosen to use CORBA [10]. The interoperability features of CORBA allow us to build distributed components which communicate through distributed FIFO queues without concern for their implementation language. All the user has to do is to provide an IDL[2] interface for its computing functions. The rest of the code generation is done by an automatic code generator we have developed [1] and the CORBA tools (IDL compiler, libraries, etc).

Other component based architectures are currently being defined, such as the CORBA Component Model (CCM) [9], the Common Component Architecture (CCA) [2] and the Parallel Application Workspace(PAWS) [7].

[1] Artificial deadlocks can only appear in cyclic graphs which are uncommon at the coarse grain programming level we propose.

[2] IDL is the Interface Definition Language used to define the exported interfaces of the CORBA objects.

But none of them hides the network transparency by using FIFO queues, the components communicate directly with each other without transparent communication latency hiding.

3 Distributed FIFO Queue

We describe in this section our implementation of a distributed FIFO queue. We focus here on efficient communications and transparency of the distribution. The manipulation of these FIFO queues is done by the usual `get`, `put` and `take` methods to respectively remove, insert and read tokens (or data elements) from the queue. To achieve this transparency, we have split the queue in two parts which communicate over the network. These two half-queues manage both the storage of the data and their communication.

3.1 Interface

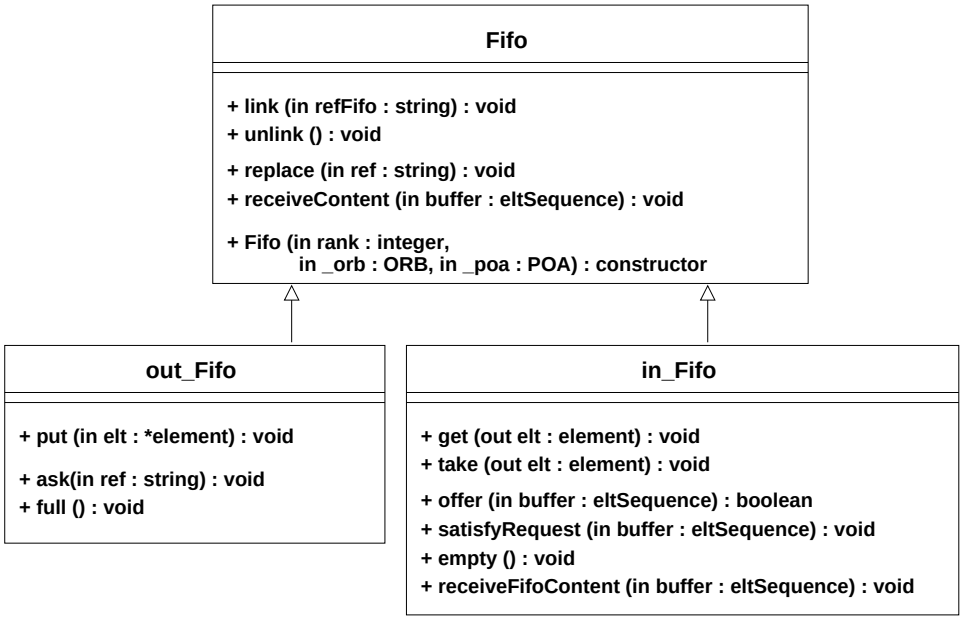


Figure 1: Class Diagram for the distributed FIFO queue

The FIFO queue class diagram is shown in figure 1. This class has the `get`, `put` and `take` methods used to interact with the queue, hiding the distribution.

The other methods are used to manage the interactions between the two half-queues or to initialize the queue. The `link` and `unlink` methods permit to create or remove (in the replacement case) a link between two half-queues.

The `replace` and `receiveContent` methods are used to replace a half-queue by another: the half-queue to be replaced receives a `replace` call with the reference of the new one, it then sends its content by a `receiveContent` call on this last one.

From the initial FIFO class, we derive two classes which are the output FIFO queue (first half or producer) and the input FIFO queue (second half or consumer). The output half-queue contains results of data processing and implements a method to receive token requests (`ask`) from the linked input half-queue. On the other hand, the linked input half-queue contains data to be processed and implements methods to receive (`satisfyRequest` or `offer`) data from the corresponding output half-queue.

3.2 Communication Protocol

The communication protocol is based on the exchange of data between the half-queues. A communication can be triggered by any of the two half-queues in a hybrid data-driven, demand-driven protocol. This protocol is completely distributed, no central authority directs the communication. Each FIFO queue handles its data transmission independently of the rest of the process network.

To manage the communications, two thresholds on the number of elements of the FIFO queue have been defined:

- a maximal threshold (for the producer half-queues) which indicates that offering a part of its tokens is necessary to avoid overloading,
- a minimal threshold (for the consumer half-queues) which indicates that it is necessary to ask the linked producer half-queue for some tokens.

When the length of the output half-queue exceeds the maximal threshold, it sends an `offer` request to the linked input half-queue. If this one reaches its maximum capacity, sending `offer` requests is momentarily suspended and the input half-queue responds by sending a `full` request to signal that it is full and can't receive data. The initiating output half-queue will then cease to `offer` data until it receives an `ask` request. Of course, if the capacity of the input half-queue was not reached, this one doesn't respond, and the output half-queue can continue to send `offer`s.

In the other case, when a minimal threshold is detected, the input half-queue is alerted. It sends an `ask` request to the linked output half-queue. This one makes a read operation in its data (the size of read data is calculated according to some criteria taking into account the size of the output half-queue) and responds with a `satisfyRequest` method invocation. If the output half-queue does not have enough data, it responds with `empty`, and the input half-queue will not ask for anything until it receives an `offer` request. This avoids numerous requests without answers.

Obviously a good choice of these thresholds is important. As this choice depends on the application, we will not discuss it here and leave an in-depth study of these parameters for future work.

4 Distributed Processes as Components

We describe here the internal structure of a component, embedding the user provided computation function and the half-queues connecting this process to others in the process network.

4.1 Component Structure

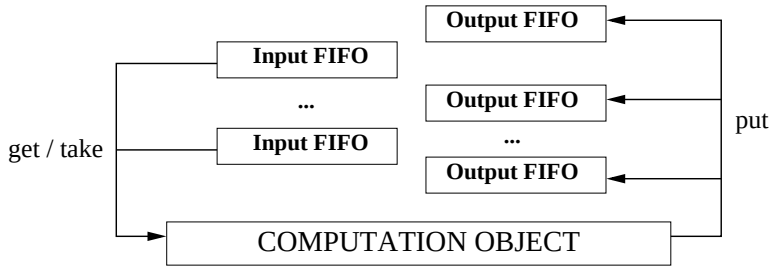


Figure 2: Component Structure

In our model, each process is a component which encapsulates different objects. In each component, there are three object types as shown in figure 2: a computation object to carry out the processing of the input FIFO queue elements and one or more input and output half-FIFO-queue objects. Each of these objects is a thread inside the component process and thus they run concurrently. The class diagram corresponding to this structure is given in figure 4.

The computation object handles the FIFO queues by calls to the `get`, `take` and `put` methods which respectively indicate the consumption, read and production operations.

The Computation class diagram is shown in figure 3. This class implements the *computation_Interface* that defines the methods used to suspend or resume the computation, and a method to receive the internal state of another component. The CORBA type `any` that can hold the value of any IDL type is used to transfer the internal state. The actual representation of this state is the responsibility of the programmer.

4.2 Asynchronism

The computation in the calculation object and the communication in the management object are executed concurrently, each object being a thread.

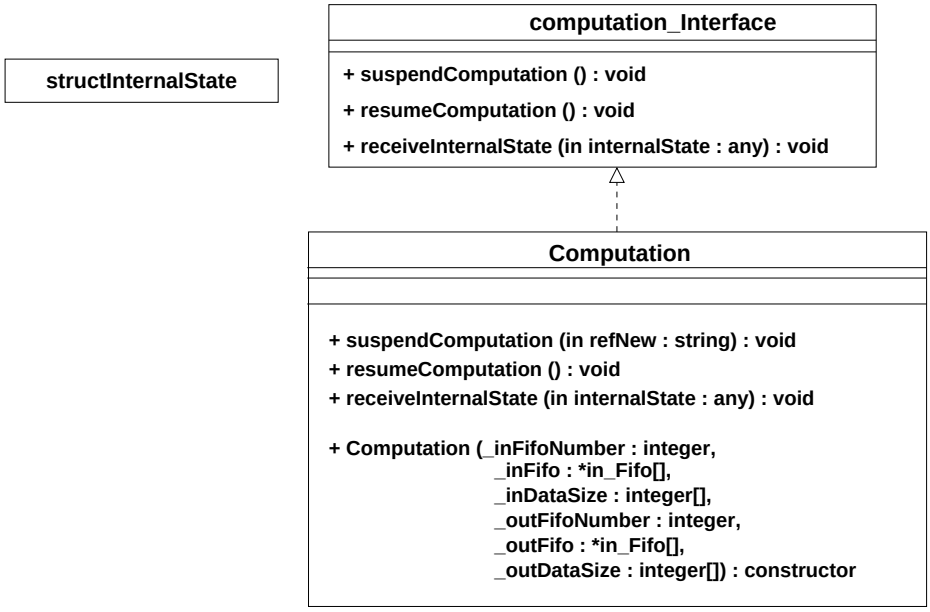


Figure 3: Class Diagram for the Computation object

However, to preserve the consistency of the data, we use critical sections to lock the concurrent accesses to the FIFO queue elements. To make the accesses to the FIFO queues more flexible, the critical section concerns the accesses to one element. We have also taken into account the performance aspect by eliminating the data copying between the different objects.

The computation inside the component and the communications with the other connected FIFO queues are done asynchronously. To do that, the requests are asynchronous and asking for data or sending data is not blocking for the computation.

5 Dynamic Component Graph

As said before, a process network is represented by a component graph. This graph is dynamic. This dynamicity is seen under three aspects:

- The first is the application's incremental development. It allows the construction of the application by adding available components interactively.
- The second is to be able to replace, during the execution, a component by another (for example to change the calculation function or to use a more performant or specialized component)[3].

[3]we focus on long running applications, so this feature is important

- The third is to migrate interactively a component from a computer to another for various reasons such as hardware upgrading or load balancing.

5.1 Interactive Console

To control the components, a console has been developed. It consists of a program which controls the component connection and execution by the use of a simple language. The presence of a manager program is contrary to the peer-to-peer character of component systems. However, the console is minimal and serves only two roles, collaboration control and component replacement. All the communications between the components are done without involving the console through the distributed FIFO queues presented in section 3. One of our future objectives is to add automatic capabilities to this console so that it adapt the mapping of the components in function of the load of the participating computers for example.

The component links that form the process network are made interactively by this console to which all the components must be connected just after their launching. The use of a console allows more flexibility in the connection choice and a dynamic control of the components.

The Process Network class diagram is shown in figure 4. In this diagram, we see mainly methods and data structures that are visible by the CORBA ORB.

The FIFO queue implements the `basic_Management` interface. Additionally, `out_Fifo` and `in_Fifo` implement respectively the `out_Management` and `in_Management` interfaces. The Console implements the `console_Management` interface which contains the method used by the components to bind the console at the beginning of their execution. In this method, the console retrieves the component name, FIFO queue informations and computation object reference. FIFO queue informations are stored into a sequence of structures containing the FIFO queue references and types.

5.2 Console Command Language

This console allows the user to interactively build and control its graph with the following commands:

- *list*: list the active components on the system;
- *init* `id`: launch the fabric object on a component;
- *link* `id1 id2`: link two FIFO queue between them;
- *suspend* `id`: suspend the computation on a component;
- *replace* `id1 id2`: replace a component by another;

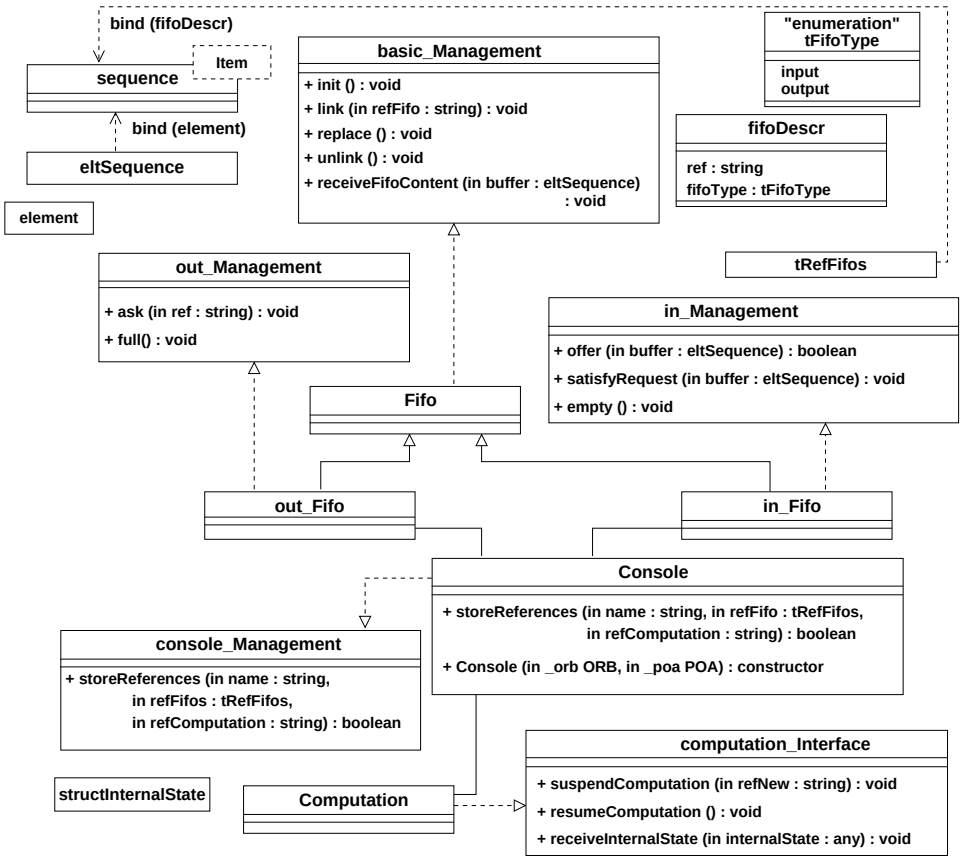


Figure 4: Class Diagram for the Components

- *show*: to see all the links between the FIFO queue;
- *length*: to get the number of tokens in all the FIFO queues;
- *fifolen* `id`: to get the number of tokens in given FIFO queue;
- *state*: to get informations about the system;
- *check*: to verify the integrity of all the components.

5.3 Replacement of a Component

The component replacement is done interactively from the console. The replacement scenario is the following:

1. The FIFO queues connected to the component that will be replaced are prevented to send or to request anything. This avoids the loss of data or the waiting of a response that will never come.
2. The console then asks the component that must be replaced to suspend the computation by invoking the `suspendComputation` method in the computation object. When this component receive this request and suspend really the computation, it sends its internal state to the new component computation object.
3. After that, FIFO queues of the component to be replaced receive the order from the console to transfer their contents to the new component FIFO queues.
4. The other components of the graph are connected to the new one, by relinking the different FIFO queues and finally computation is launched.

6 Signal Processing Application

We illustrate our model on a representative example of signal processing.

6.1 Application Description

The application (see figure 5) consists of providing frequencies and location correlations (so called *beams*) from a continuous stream of data. It is based on elementary signal transformations: FFT (Fast Fourier Transformation) and discrete integration.

- The *Hydrophones*, an ($h = 512 \times T = \infty$) array, `HYDRO`, is the input of the application. It delivers a continuous stream of data from a set of 512 captors.

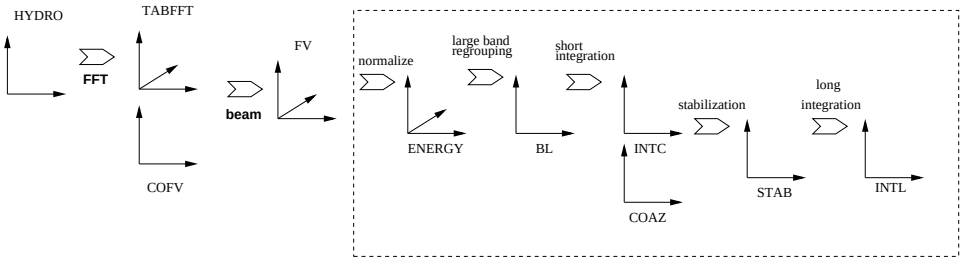


Figure 5: Application Description

- The first task computes FFT for each captor and period of 512 units of time. It produces `TABFFT`, a three dimensional array ($512 \times 256 \times \infty$).
- The second task computes a beam for each period, frequency and set of captors. It outputs *Beams*, an ($t = 128 \times T = \infty \times f = 200$) array, `FV`.
- The beams are then treated successively by different functions, to finally extract characteristic frequencies. The input and output array sizes of the different functions are:

Function	Input array	Output Array
Normalization	$(128 \times \infty \times 200)$	$(128 \times \infty \times 200)$
Large band regrouping	$(128 \times \infty \times 200)$	$(128 \times \infty)$
Short integration	$(128 \times \infty)$	$(128 \times \infty)$
Stabilization	$(128 \times \infty)$	$(128 \times \infty)$
Long integration	$(128 \times \infty)$	$(128 \times \infty)$

6.2 Process Network Specification

From the application description, it is easy to define the corresponding process network. Due to the high speed of the five last tasks compared to the two first ones, we group them in just one process. We obtain three processes in a linear pipe-line. The first task computes an ($h = 512$) array, `HYDRO` and so on. The time dimension (with infinite size) becomes the successive tokens produced as input of the first process.

The sequential program used for performance analysis breaks the infinite dimension too, but only one process executes the complete application.

6.3 Performance Measures

For the performance measures, we have used three networked workstations. All the input tokens are almost available at the beginning of the run. The

acquisition of the hydrophone values is considered as instantaneous and the pipe-line works in a saturation state. We have done several measures (see table 1): the sequential program running on a 1 GHz Pentium III computer, the three component programs on two 1 GHz Pentium III computers and one 266 MHz Pentium II computer linked by a 10 Mb/s Ethernet bus. All the times measured here are wall clock times and it should be noted that the workstations were lightly used by other processes.

# of tokens	Sequential P3	CORBA P3	CORBA cluster	1st Task P3	2nd Task P3	3rd task P2
5000	51 s	52 s	32 s	22 s	26 s	10 s
10000	100 s	104 s	62 s	41 s	48 s	20 s
15000	154 s	168 s	97 s	56 s	68 s	29 s
20000	203 s	214 s	123 s	75 s	91 s	38 s
25000	246 s	271 s	148 s	94 s	115 s	48 s
30000	309 s	322 s	173 s	118 s	131 s	57 s
35000	362 s	382 s	198 s	141 s	156 s	65 s
40000	408 s	430 s	230 s	166 s	180 s	74 s
45000	459 s	496 s	263 s	184 s	205 s	83 s
50000	516 s	570 s	290 s	210 s	236 s	93 s
2 500 000	26120 s	N.A.	15980 s	11270 s	12590 s	4570 s
2 750 000	27580 s	N.A.	17730 s	12910 s	13140 s	5120 s
3 000 000	29050 s	N.A.	19096 s	13550 s	13800 s	5580 s

Table 1: Application performances

Working under saturation state, the ratios between the sequential and the CORBA solutions stay similar. It means that the queue management time is proportional to the size of the queue. The overhead cost due to the CORBA layers without communications is less than 10% (on the same monoprocessor P3).

The speed-up allows to increase the frequency of acquisition in the same proportion, see figure 6 that shows the relative times of the tasks with respect to the total execution time.

Dynamic redistribution and FIFO queue length evolution. Table 2 and figure 7 show the evolution of the FIFO queue length between the two first components. During the execution under saturation state, the second component which runs on a Pentium II 266 MHz PC, has been replaced by another one which runs on a faster machine (Pentium III 1 GHz PC). We note that the FIFO queue length has been decreased immediately after the replacement. A bigger acquisition frequency should be supported on this new cluster of PCs.

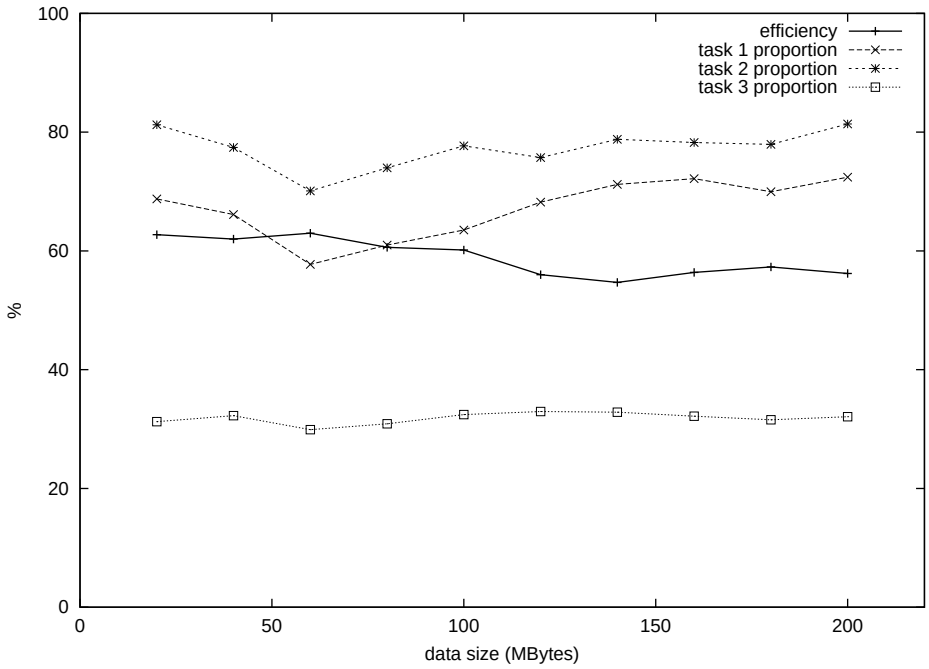


Figure 6: Cluster execution efficiency - load unbalancing of the 3 tasks

Data size (Mbytes)	P3-P2-P2	P3-P3-P2
20	80 s	47 s (replacement after 30 s)
40	166 s	90 s (replacement after 60 s)
60	258 s	158 s (replacement after 90 s)
72	302 s	188 s (replacement after 120 s)

Table 2: Dynamic redistribution: the second task migrates from a P2 to a P3 during the run

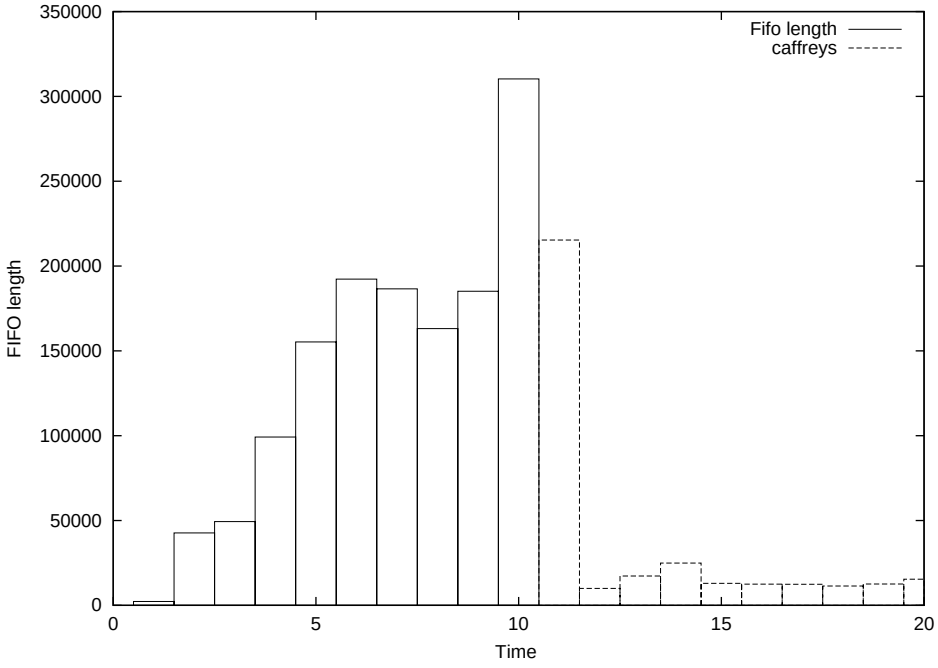


Figure 7: Fifo queue length evolution during the redistribution

7 Conclusion

The process networks are specially suited to the development of intensive signal processing applications or scientific computation. Those are built on the concept of FIFO queues which convey tokens between processes. This execution model becomes a preferential target of a specification model dedicated to applications of this type for network architectures such as PC clusters.

We have proposed a distributed execution of this model on the top of the CORBA middleware. It guarantees interoperability between the languages and the dynamicity not only of the application specification but also of its placement on a network. The results obtained are promising, they show that the distributed implementation of a signal processing application makes possible to satisfy higher frequencies of acquisition according to a higher speed-up.

The mapping and the control of the scheduling of the processes is carried out explicitly via a console interface. That makes feasible to start an application respecting the process network model before this one is not completely specified. It also allows the migration of processes between two machines, for example to ensure a better load balance. Finally it authorizes the substitution of one process by a more efficient one. This can be done

during the execution of the application, without untimely shutdown. Thereafter, it would be appropriate, by the means of informations received during the execution to set up an automatic or semi-automatic adaptive system which carries out these transformations during the execution. A decision tool would then be coupled with our interface.

For applications where the processing time of the various processes strongly varies, the improvement of the performances and thus of the acquisition frequencies in the case of the signal processing, could be obtained by association of several processes to the same input queue.

Finally experiments under a stabilized acquisition mechanism should make possible to build performance cost function of a placement by taking into account the times of acquisition, the sizes of the input data as well as the processing times of each process.

References

[1] Abdelkader Amar, Pierre Boulet, and Jean-Luc Dekeyser. Assembling dynamic components for metacomputing using CORBA. In *Parallel Computing 2001*, Naples, Italy, September 2001.

[2] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, , and Brent Smolinski. Toward a Common Component Architecture for high-performance scientific computing. In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, 1999.

[3] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California at Berkeley, 1993.

[4] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI: Application modeling for signal processing systems. In *37th Design Automation Conference*, Los Angeles, CA, June 2000.

[5] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., August 1974.

[6] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*, pages 993–998. North-Holland, 1977. Proc.IFIP Congress.

[7] Kate Keahey, Pat Fasel, and Sue Mniszewski. PAWS: Collective interactions and data transfers. In *High Performance Distributed Computing (HPDC-10)*, August 2001.

[8] Edward A. Lee. *Overview of the Ptolemy Project.* University of California, Berkeley, March 2001.

[9] Raphaël Marvie, Philippe Merle, and Jean-Marc Geib. Towards a Dynamic CORBA Component Platform. In *Proceedings of the 2nd International Symposium on Distributed Object Applications (DOA'2000)*, Antwerp, Belgium, September 2000.

[10] Object Management Group, Inc., editor. *Common Object Request Broker Architecture (CORBA), Version 2.6.* `http://www.omg.org/technology/documents/formal/corba_iiop.htm`, December 2001.

[11] Thomas M. Parks. *Bounded Scheduling of Process Networks.* PhD thesis, University of California at Berkeley, 1995.

[12] Darren Webb, Andrew Wendelborn, and Kevin Maciunas. Process networks as a high-level notation for metacomputing. In *Workshop on Java for Parallel and Distributed Computing (IPPS)*, Puerto Rico, April 1999.

[13] Darren Webb, Andrew Wendelborn, and Julien Vayssière. A study of computational reconfiguration in a process network. In *IDEA7*, Victor Harbour, South Australia, February 2000.

Annexe F

Visual Data-Parallel Programming for Signal Processing Applications

Visual Data-parallel Programming for Signal Processing Applications

Pierre Boulet Jean-Luc Dekeyser Jean-Luc Levaire Philippe Marquet
Julien Soula
Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, France

Alain Demeure
Thomson Marconi Sonar, Sophia-Antipolis, France

Abstract

Matrix manipulation programs are easily developed using a visual language. For signal processing, a graph of tasks operates on arrays. Each task iterates the same code on different patterns tilling these arrays. In this case visual specifications of dependencies between the pattern elements are enough to define an application. From the Array-OL language developed by Thomson Marconi Sonar, we propose a graphical environment, Gaspard, dedicated to the data-parallel paradigm. Only elementary SPMD tasks are textual. A full environment has been implemented; it includes a graphical editor, a code transformer and a code generator for SMP computers.

1 Introduction

Signal processing dedicated to detection systems refers to multidimensional arrays. As in digital sound processing, a first dimension allows to sample the signal in chronological order. A second dimension generally represents the different sensors, the temporal sampling is applied on each of them. During the signal processing, others dimensions may appear. For example during the FFT implementation a new dimension represents the frequency. The temporal reference is modified and matches the sampling of the different FFT execution ages.

In order to conform to the needs for specification, standardization and efficiency of the multidimensional signal processing, Thomson Marconi Sonar has developed a Signal processing oriented language: Array-OL (Array Oriented Language) [5]. Taking into account that matrix manipulation programs can be more easily constructed with a visual language than with a textual language [7], Array-OL relies on a graphical formalism in which the signal processing appears as a graph of tasks. Each task is performed on multidimensional arrays. The language compilation targets as much Unix workstations, mainly to tune applications, as dedicated parallel processors designed to be embedded.

Using the Array-OL programming model, we propose a visual specification tool that allows to design an application program by collecting graphical elementary software components. From this assembly, the compilation phase takes place to generate C++ code (with/without Pthread calls). Gaspard also enables to update the code through modifications of the visual specification.

After a brief introduction to Array-OL in agreement with the different levels of the development environment, the programming model will be explained. Subsequently the whole specification environment will be presented. Lastly the successive visual steps to built a signal processing application will be described and illustrated with screen snapshots.

2 Array-OL Programming Environment

From the Array-OL language, TMS developed a whole toolbox dedicated to signal processing applications. Four different levels have been established:

- In order to ease the Array-OL program production and to free the programmer from the syntactic constraints usually associated to programming languages, TMS proposes a visual programming interface built on the Ptolemy environment [8]. It allows to graphically specify (visual programming with examples) an Array-OL application through a graph of elementary tasks. By means of the comprehenser, it automatically generates the corresponding source code in the Array-OL syntax.

  Gaspard inherits in straight line from this tool. It takes up the main characteristics adding to the easy access to visual programming by offering reusable software components. It is fully written with public domain tools, freeing the user from all the constraints due to Ptolemy usages.

- The application domain intended for covers filtering computations on regular data flux. (Fast Fourier transformation, discrete transformation...). Array-OL is an array oriented language particularly well suited for signal processing.

  Gaspard allows to extend the application area by the creation of new user-defined specialized software component libraries.

- Once the application is written, for validation, it is often desirable to simulate and evaluate the application on a workstation. TMS has developed for this purpose an Array-OL to C++ compiler.

  The runtime libraries of this compiler are also used in Gaspard. Moreover the code generation for SMP multiprocessors is included into the Gaspard compilation stage using the Pthread library.

- The objective was to efficiently execute applications in a real world situation, so TMS has built a parallel processor dedicated to Array-OL. Code generation for this processor needs to call micro-code.

  Gaspard was defined in such a way that it could integrate this code generation for this specialized processor, even if the current version does not support it.

3 Array-OL Language

Array-OL proposes a two level approach [5, 4]. A first global level defines the task scheduling in the form of dependencies between tasks and arrays. A second local level details the elementary actions the tasks realize on array elements.

Global Model. The global model looks like the well-known dataflow model: the application is composed of task nodes, while edges define dependencies between tasks. Each edge carries an array. Nevertheless, in the dataflow model, the graph edges carry a continuous token flow and all the tasks are running in parallel; in the Array-OL model, an edge carries an unique array (which may be of infinite size) and each task is triggered only once. A graph node (task) execution produces its output arrays from its input arrays. The task specification and the details of the array element usage are hidden at this specification level.

Array: a Data Structure for Signal Processing. Signal processing applications are organized around a regular and potentially infinite stream of data. Array-OL captures this stream in arrays with a possible infinite dimension.

Some spatial dimensions of arrays used in signal processing correspond to sensors. Such sensors may be organized in circle. Consequently, Array-OL array dimensions wrap around to form a toroid.

Local Model. The local model details the task specification. It defines the access to the data in

the arrays and the computations to be done on those data. The whole task execution is divided in small identical computations called *Elementary Transformations* (ET). An ET operates on subsets of the arrays called *patterns*. Output patterns (patterns in the output arrays) are produced by applying the ET on the patterns of the input arrays. So, a task always consists of an iterator constructor; whose iterations are independent.

ARRAY-OL being restricted to the specification of signal processing applications, the shape of the patterns, the array tiling by the patterns, and the task code are not general purpose. Ad hoc specifications are proposed.

Fitting: Pattern Definition. Patterns are multidimensional arrays. Equidistant elements in a pattern are equidistant in the array.

A pattern may be defined by an origin in the array and a set of vectors (fitting vectors; one vector being associated to each dimension of the pattern). The other points of the pattern are defined in the array by a shift of the origin along the fitting vectors as much as required by the pattern size (Figure 1(a)).

Paving: Tiling of an Array with its Patterns. Two equidistant output patterns are produced by two equidistant input patterns.

The array paving with patterns is given by a first pattern in each array and a set of paving vectors. The other patterns are defined by a shift of the initial pattern along the paving vectors as much as needed to cover the master array (Figure 1(b)). By definition, two patterns of an output array may not overlap.

ET Library or Hierarchical Definition. For each paving iteration, a task extracts the input patterns from the input arrays and applies an ET on these patterns to produce output patterns. These patterns are then stored in the output arrays.

A library of predefined ETs is available for usual signal processing operations (FFT, integration...). An ET takes arrays as input and returns arrays; it may be parameterized, for example by the size of the arrays.

A hierarchical extension of ARRAY-OL allows the programmer to define its own ET in ARRAY-OL. Input/output patterns of the first level are considered as arrays on the sublevel. A new ARRAY-OL global level defines tasks that manipulate these arrays. This hierarchical construction may be applied as many times as necessary.

4 GASPARD Specification Environment

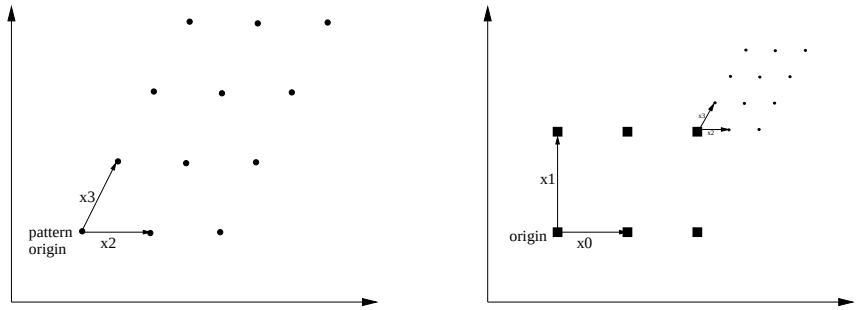
The GASPARD visual specification environment is built upon the characteristics inherited from the global model of ARRAY-OL. This global model is represented by a visual description of a software component. The metaphor we use is the design of an electronic device (on a printed circuit). The arrays are pieces of information flowing on the wires, these wires interconnect slots into which other software components are plugged. The plugging process specifies the paving and fitting that indicate how the plugged component is fed with arrays. The patterns produced by the slot through this plugging process become arrays of the plugged component.

Thus each component can be specified independently of the components it references. Such components are inherently reusable. A component becomes intanciated only when it is plugged into a slot. Thus, component libraries can be built to help the design of specific application categories.

The components only accept arrays as inputs or outputs. And the array-component links express the data dependencies used by the compiler to build an execution scheme. Let us now precise the metaphor.

Component. A *component* corresponds to an encapsulated action that acts on all or some of the arrays. There are two types of components:

- A *primary component* is not specified by other components. It is described by a (possibly multithreaded) C++ function that takes arrays as inputs and produces arrays. These input and output parameters are the *links* of the compo-



(a) (b)

Figure 1: Fitting and paving: (a) A set of paving vectors and an origin define a pattern in an array. (b) From an origin pattern in an array, the subsequent patterns in are defined by a set of paving vectors.

nent. The function has to be pure to allow a SPMD execution.

- A *compound component* consists of several primary or compound components linked together by arrays.

Link. Some arrays of a compound component are particularized: the *input/output links*:

- An *input link* is an array that is neither produced inside the component, neither associated to a file, nor initialized by constants.
- An *output link* is an array that is neither consumed inside the component, nor associated to a file.

These links define the component's interface; they are later associated to arrays when the component is plugged into a slot. The components are regular: the size of the links must be static[1]. The links and the component are "welded" together and so are indissociable.

Component Completeness. The *completeness* of a component is obtained by the connection of all its links to arrays of a higher level in the hierarchy of the application components. This completeness is done by the mean of the slots. Two types of completeness are defined:

- *Direct completeness*: the connection is done between an array of the higher level component and an input/output link of the plugged component. The two arrays must be conformant: same shape, same size. The reception slot is called a *direct slot*. The instance of the plugged component is then executed once, sequentially.
- *Iterative completeness*: the links of the plugged component are related to the arrays of the higher level one through a paving/fitting iterative operator. The links of the plugged component are thus connected to the patterns defined by such an *iterative slot*. One of the output links is defined as the master of the iteration. It specifies the iteration domain by the complete tiling of the array of the higher level component it is associated with. The plugged component is instantiated in a SPMD way. The execution order is not specified.

A compound component is said complete if all its components are complete. The two kinds of completeness (direct or iterative) are applicable to the

[1] A dynamic approach has been introduced in [1].

two kinds of components (primary or compound). A compound is said *executable* if it is complete and has no input or output link. It is then called an *application*.

The data are multidimensional arrays of elementary types: integers, floats or complexes. At most one dimension of an array may be of infinite size. It allows to consider time as a dimension of the arrays upon which paving and fitting iterations are also available. All the dimensions of the arrays are toroidal, allowing paving and fitting without any edge effect. During the compilation, transformations by hierarchical level insertion are applied on the application allowing the use of this infinite dimension notion on finite memories [2].

5 Example: Computing Energy of Hydrophone Signals

We describe here the successive steps to specify an application in the GASPARD environment. This application takes in input `Hydro`, a two dimensional array (512 × infinite) (Figure 2). The first dimension corresponds to the signals of the 512 sensors, the second infinite dimension represents time. The application first computes a Fast Fourier Transform (FFT) on each signal in a SPMD manner, producing `TabFft`, a three dimensional array (512 × 256 × infinite). The second dimension now represents the frequencies, the infinite one is still the time. Then it creates the different tracks in the `Fv` array, whose norms will represent the energies of the signals. Both computations are also SPMD.

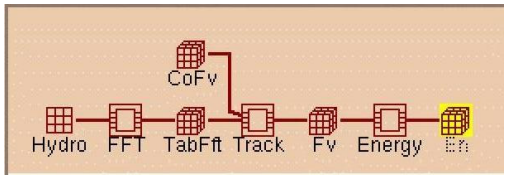
The specification of this application is built interactively using the GASPARD environment. The first step is to create `Energy`, a new project corresponding to the application, and `Main`, its toplevel component (Figure 2(a)). The second step is to declare arrays and slots in this component, and to link them using the mouse to specify the flow of the data (Figures 2(b) and 2(c)). At the same time, array characteristics may be specified by opening them (Figure 2(d)).

At that time, the `Main` component is not yet executable as its three slots are not plugged with other components. The user can then create new components in his application and plug them into the slots, or use existing components provided by other applications or libraries. In our example, we plug into the `FFT` slot, using drag and drop, the `FFTDbl` primary component provided by another application. Each slot, once plugged, should be edited to connect its input and output arrays with the input and output links of the plugged component (Figure 3(a)). Finally, for each connection, the paving and fitting vectors have to be specified to ensure the completeness of the application (Figure 3(b)). Actually, the GASPARD environment does not impose any order in the specification of an application. We propose here a top-down approach, but any other order is possible. The only point is that an application is fully specified when its top-level component is complete.

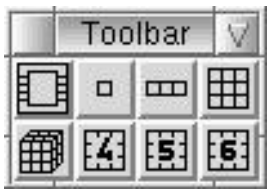
Once complete, an application may be compiled and executed. During this phase, the management of the infinite dimensions is ensured by computing an explicit recurrence value for each infinite dimension. These values are deduced from the paving and fitting vectors associated to the infinite arrays. Actually, introducing a recurrence amounts to add a level in the component hierarchy. A sequence of slots linked to infinite arrays may be replaced with a compound component representing the same sequence, but linked to finite arrays. The recurrence values computed above will determine the sizes of those arrays. This technique can be seen as a hierarchical programming methodology, leaving all infinite arrays at the top level (Figure 4). It has also been integrated in the GASPARD environment as a transformation tool, aimed to tune applications to specific architectures [2]. Sizes of arrays have indeed a direct performance impact on different architectures.

The target language of the compiler currently is the C++ language. The compiler has been extended to also generate multithreaded C++ code, using the Pthread library. The transformation tools, the compilers and the graphical interface all use the same internal representation, an abstract syntax tree, which is managed via a unique server. Thus, the effects of transformation tools are immediately visible through the graphical interface. The architecture of the GASPARD environment is presented in Figure 5 [3].

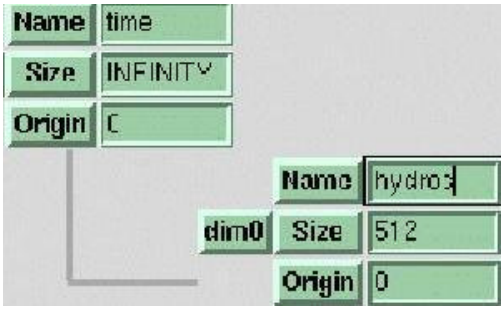
(a) Creating a new project and some components



(b) Declaring and linking arrays and slots...

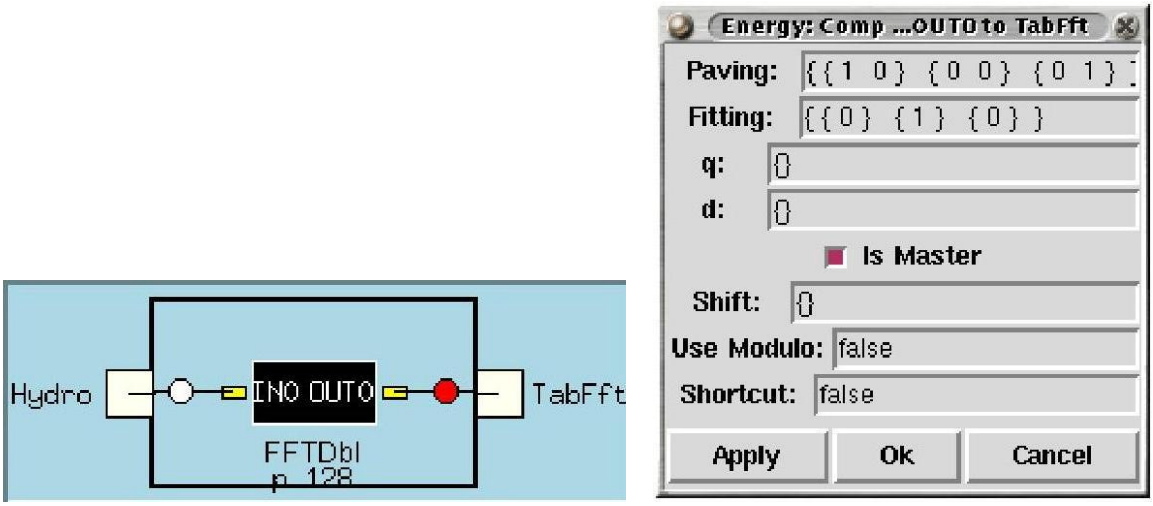


(c) ...using drag and drop from this toolbar



(d) Specifying array characteristics

Figure 2: Declaration of arrays and slots for the Energy application.



(a) Connecting arrays and links

(b) Specifying paving and fitting

Figure 3: Plugging a component into a slot.

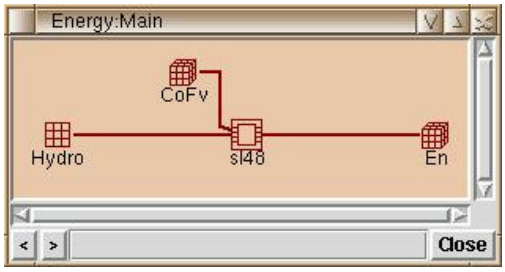
6 Conclusion and Future Works

We have presented a specification environment built on the data-parallel paradigm (local model) inside a dataflow graph (global model): Gaspard. This one takes up the signal processing algorithms developed in Array-OL. It widens the field of application by offering new elementary software components. These components are defined by the programmers themselves. The visual hierarchical aspect allows to reuse components and to create specialized component libraries. No directive concerning the execution model is expressed; the compiler is able to plan specific mapping strategies according to the dependence specifications and the target architecture. Right now, the compiler is intended for Unix workstations and SMP multiprocessors through calls to the Pthread library. Visual programming associated to textual programming (code of the elementary components) discharges the programmer from all the syntactic constraints due to a parallel language, without constraining him to visually specify all the code of the application (at the instruction level) [6]. The printed circuit metaphor retains scientist's habits and some experiments show the easiness of the elaboration of new applications using Gaspard.

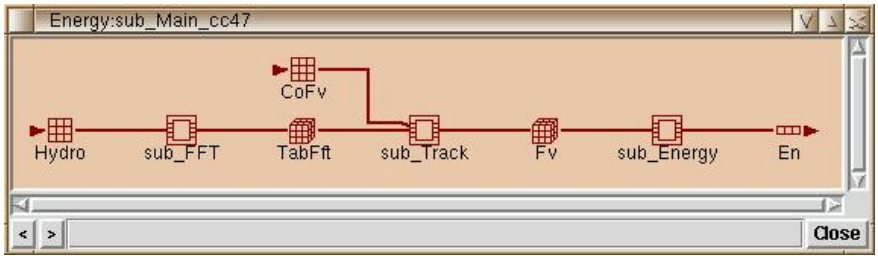
Clearly, Gaspard only puts up with regular SPMD executions for which the size of the data and the dependencies between the array elements are statically defined. An extension towards a collection based model was experimented and will be integrated in the next versions of Gaspard [1].

Gaspard remains an editing tools although the visual interface is well suited to assist the programmers at runtime. In this way, we plan to make Gaspard interactive. It should facilitate the spying of the execution of an application as and when the progress of its specification. This approach seems to us perfectly suitable for a SPMD metacomputing system implemented on a cluster of SMP computers. In this case, the array flow of Gaspard corresponds to the message exchanges between the computer nodes of the network.

(a) The new project



(b) The toplevel component



(c) The subtask plugged in the toplevel component

Figure 4: Hierarchization of the `Main` component of the initial `Energy` application (Figure 2(b)).
(b) The top level component computing infinite arrays now consists of a sole slot (`sl48`) that directly produces the `En` infinite array from the `Hydro` infinite array.
(c) Each component, each array, and each slot of this new component is automatically produced from the specifications of the initial project.

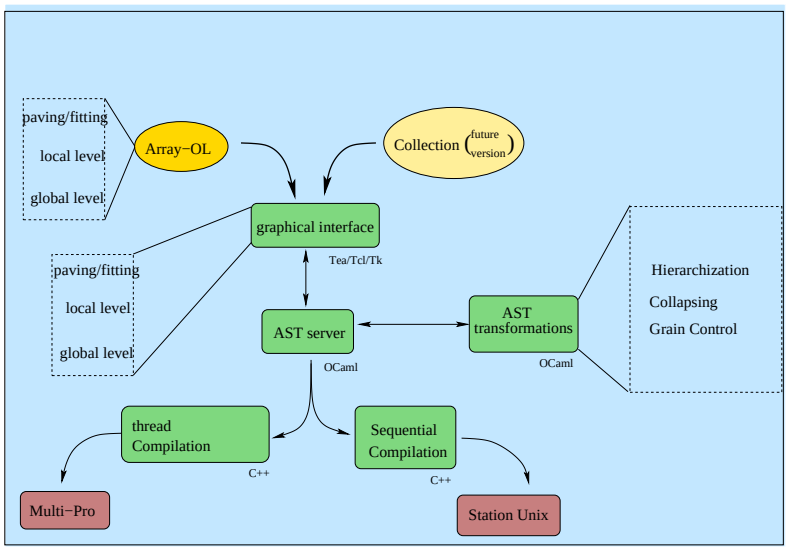


Figure 5: Architecture of the GASPARD Environment

References

[1] Pierre Boulet, Jean-Luc Dekeyser, Alain Demeure, Florent Devin, and Philippe Marquet. Une approche à la SQL du traitement de données intensif dans Gaspard. In *RenPar'11, Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, Rennes, France, June 1999.

[2] Jean-Luc Dekeyser, Alain Demeure, Philippe Marquet, and Julien Soula. Array-OL compilation by code transformation. Research Report 99-15, LIFL, Université de Lille, France, December 1999.

[3] Jean-Luc Dekeyser, Philippe Marquet, and Julien Soula. Video kills the radio stars. In *Supercomputing'99 (poster session)*, Portland, OR, November 1999. (`http://www.lifl.fr/west/gaspard/`).

[4] Alain Demeure and Yannick Del Gallo. An array approach for signal processing design. In *Sophia-Antipolis conference on Micro-Electronics (SAME 98)*, France, October 1998.

[5] Alain Demeure, Anne Lafage, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude Dufourd, and Jean-Louis Marro. Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Gretsi*, Juan-Les-Pins, France, September 1995.

[6] Martin Erwig and Bernd Meyer. Heterogeneous visual languages - integrating visual and textual programming. In *VL'95, IEEE Symposium on Visual Languages*, pages 318–325, Darmstadt, Germany, 1995.

[7] Rajeev Pandey and Margaret Burnett. Is it easier to write matrix manipulation programs visually or textually? An empirical study. In *IEEE Symposium on Visual Languages*, pages 344–351, Bergen, Norway, August 1993.

[8] Ptolemy. An overview of the Ptolemy project. Technical report, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, March 1994.