Collège doctoral

N° attribué par la bibliothèque

T H È S E

pour obtenir le grade de
Docteur de l'École des Mines de Paris
spécialité *Informatique Temps Réel, Robotique, Automatique*

présentée et soutenue publiquement
par
M. Nicolas MUSEUX
le 20 décembre 2001

AIDE AU PLACEMENT D'APPLICATIONS DE TRAITEMENT DU SIGNAL SUR MACHINES PARALLÈLES MULTI-SPMD

RENCONTRE DE LA PARALLÉLISATION AUTOMATIQUE ET DE LA PROGRAMMATION PAR CONTRAINTES

Thèse effectuée en collaboration avec
THALES Research & Technology FRANCE
(ex Laboratoire Central de Recherches de THALES)
Domaine de Corbeville 91404 Orsay

Thèse dirigée par François IRIGOIN

Jury

M.	Paul	FEAUTRIER	Président	INRIA/UVSQ-PRiSM
M.	Denis	BARTHOU	Rapporteur	UVSQ-PRiSM
M.	Patrice	BOIZUMAULT	Rapporteur	Univ. Caen-GREYC
M.	Eric	LENORMAND	Rapporteur Industriel	THALES
Mme.	Corinne	ANCOURT	Examinatrice	EMP-CRI
M.	Michel	BARRETEAU	Examinateur	THALES
M.	François	IRIGOIN	Examinateur	EMP-CRI
M.	Michel	MINOUX	Examinateur	Univ. P et M Curie-LIP6

À Marlène, Charlotte et mes parents.

Remerciements

Si l'on compare un sujet de thèse à une forêt vierge, le défrichage nécessaire pour la traverser est semé d'embûches. Mais pendant ce parcours difficile on fait des rencontres permettant d'avancer encore et toujours. Ces rencontres sont différentes les unes des autres. Certaines améliorent le quotidien, d'autres sont plus professionnelles. Mais toutes contribuent d'une façon ou d'une autre, directement ou indirectement, au travail résultant de ces trois années de thèse. Aussi, si une page de remerciements semble obligatoire dans un tel mémoire, elle y a sa place légitime. Celle-ci tout particulièrement.

Je commencerai par remercier très vivement Juliette Mattioli et François Irigoin pour la confiance qu'ils m'ont accordée l'un et l'autre, leur temps précieux qu'ils m'ont consacré, leurs conseils toujours bienvenus ainsi que leur soutien de tous les instants.

Je remercie Patrice Boizumault et Denis Barthou d'avoir accepté de rapporter cette thèse. Je remercie également Paul Feautrier et Michel Minoux qui m'ont fait l'honneur de participer à ce jury.

J'exprime toute ma reconnaissance à Corinne Ancourt et Michel Barreteau pour leur patience, leur écoute, leur conseils et les différentes discussions fructueuses que nous avons pu avoir.

Sans Éric Lenormand et Denis Aulagnier, avec qui nous avons eu de grandes discussions autour de l'application et l'architecture radar exploitées, cette thèse n'aurait pu se rapprocher autant d'un contexte opérationnel.

Au travers de Robert Mahl et Yves Rouchaleau, j'aimerai remercier toute l'équipe du Centre de Recherche en Informatique de l'École des Mines de Paris, qui a su m'accueillir chaleureusement (avec une pensée particulière pour Youcef et Nga - Tenez bon, c'est la dernière ligne droite.).

Un grand merci à Jean Jourdan et à l'ensemble des personnes constituant le Department of Advance Software de Thales Research & Technology, présentes et passées, parmi lesquels Pierre, Simon, Youssef, William, Fabien, Agnès, Christophe, Ludivine, Rodrigo... Chacun m'a fait profiter de son expérience que j'ai essayé de faire fructifier dans mon travail et pendant les pauses café.

Mais il y a également tout ceux qui m'ont soutenu, dans l'ombre, durant ces trois années. Tout d'abord mes parents (mes plus fidèles supporters depuis le début de mon histoire), pour qui j'estime qu'un simple merci ne suffit pas, mais à qui j'exprime la plus profonde des reconnaissances pour avoir toujours été là (surtout entre le 26 décembre 1999 et mai 2000 ;-D). Je voudrais, ensuite, rendre hommage à Marlène qui a fait preuve de patience et d'indulgence envers moi, et qui a supporté mes différents sauts d'humeur. Je n'oublie pas non plus Charlotte, François, le petit Rémi (c'est Tonton !), Arnaud, Christel, Carine, Vincent, Hervé, Lucie, Fred, Laurent, Nadège, ma belle-mère (qui sait comment me gâter : un morceau de gâteau sec et un verre de pineau), et tout ceux que je n'ai pas cités ici.

Je terminerai cette page en remerciant la première personne, en dehors de ma famille, qui m'ait fait confiance. Cette personne était mon professeur d'histoire/géographie alors que je n'étais qu'en $4^e/3^e$. Madame Farrache, en serais-je arrivé là sans la confiance dont vous m'avez gratifié à l'époque ?

Sommaire

Aide au placement d'applications de traitement du signal sur machines parallèles multi-spmd

Chapitre 1

Introduction

Face aux énormes puissances de calcul requises par les applications de traitement du signal et à la complexité des architectures qui doivent les absorber, la conception conjointe logiciel-matériel devient incontournable, réduisant ainsi le coût de réalisation et la durée du cycle de développement. En effet, l'évolution incessante de la technologie silicium permet de concevoir de véritables systèmes intégrant différents types de traitements sur une même puce : les *Systems On a Chip* (ou SOCs). La complexité de leur réalisation et les coûts financiers qu'ils induisent nécessitent une conception conjointe logiciel-matériel de plus en plus élaborée en vue d'un prototypage rapide. Les environnements existants offrent souvent des fonctionnalités limitées à leurs applications. Cette situation s'applique au problème du placement[1], sur des architectures multi-processeurs, d'applications temps réel embarquées. En effet, puisque la distribution des calculs et des données sur la machine et leur séquencement influencent les performances de ces applications (au travers du nombre de ponts de communications ou de l'utilisation des zones d'allocation mémoire induits). Un outil d'aide au placement qui prend en compte les caractéristiques conjointes de l'algorithme et de l'architecture cible permet d'accélérer la recherche de la meilleure adéquation et ainsi de réduire le cycle de développement.

Le placement optimal de nids de boucles présentant un parallélisme potentiel sur une architecture parallèle est un problème reconnu NP-complet [44]. La recherche réalisée par la communauté de la parallélisation automatique est principalement centrée sur la conception de nouveaux algorithmes et techniques pour la résolution séparée de chacune des fonctions nécessaires au placement d'applications sur machines parallèles [27]. Certains s'intéressent à la résolution conjointe de quelques unes de ces fonctions [22]. Peu de travaux ont été consacrés au traitement simultané et de façon concurrente des fonctions, cependant usuelles dans cette communauté [48][40][45][21][52], que sont le partitionnement, l'alignement, la distribution et l'ordonnancement.

De manière générale, le placement est la recherche d'une adéquation entre un algorithme et une machine sur laquelle doit s'exécuter le programme sous-jacent à l'algorithme.

Nous montrons dans cette introduction avec quels moyens nous avons outillé la fonction placement, au travers des applications de traitement de signal systématique (TSS) de Thales. Nous donnons ensuite une définition du placement d'applications sur machines pa-

[1] Distribution de calculs et de données dans l'espace et le temps.

rallèles. Nous expliquons, de façon macrocoscopique, notre méthodologie de placement ainsi que ce qu'est la Programmation Par Contraintes et quels sont ses mécanismes.

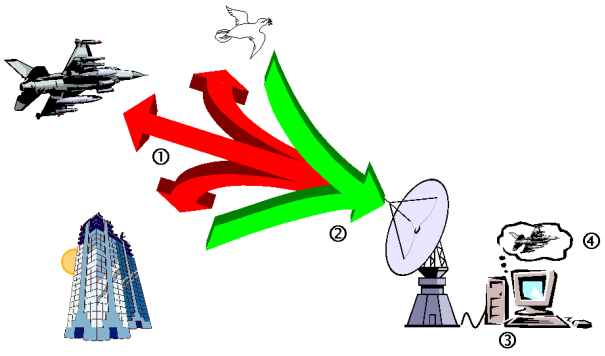
1 Le placement dans THALES

Dans le cadre de cette thèse, les algorithmes à placer pour THALES sont des applications de TSS. Les machines cibles sont des architectures multi-processeurs à mémoire distribuée. La raison d'effectuer des recherches dans ce domaine est simple : les puissances de calcul toujours croissantes donnent de nouvelles possibilités aux différents algorithmes de calculs scientifiques. Cependant le programmeur voulant profiter au maximum de ces avancées technologiques doit faire preuve d'agilité intellectuelle. La recherche d'une adéquation parfaite entre application et architecture cible est loin d'être aisée lorsqu'il s'agit de systèmes embarqués, exigeant généralement des temps de réponse limités, et contraints par les ressources de la machine. De ce fait la nécessité d'un outil d'aide au développement dès la conception des algorithmes apparaît comme incontournable. L'aboutissement de ces recherches permettront également d'amortir les coûts de développement lors d'évolutions (architecturales ou applicatives) et accroît la capitalisation du travail effectué. Enfin, le maintien d'un système est d'autant plus facile qu'il ne dépend pas uniquement du programmeur.

1.1 Caractérisation d'une application TSS

Les applications étudiées sont celles du TSS[2] (fig. 1.1, [16]). Elles sont composées de différentes tâches faisant chacune appel à une fonction de filtrage bas niveau (un calcul de puissance de crête ou une transformation de Fourier par exemple) agissant sur un signal réfléchi échantillonné. Nous appellerons ces fonctions de filtrage des tâches élémentaires (TE).

FIG. 1.1 – Principe de fonctionnement d'un radar



Un radar émet une onde (phase 1) réfléchie par différents objets (ici un avion de chasse, une colombe et un immeuble). Elle revient alors au radar (phase 2). Son échantillonnage est transmis à un calculateur chargé d'éliminer le «bruit» introduit dans le signal initial afin de détecter une cible dans l'espace et/ou le temps (phase 3). La phase 4 est la reconnaissance de la cible et la prise de décision quant à l'action adéquate à déclencher. Les applications étudiées sont celles constituant la phase 3.

Le signal échantillonné est stocké dans un tableau multidimensionnel où chaque dimension correspond à une sémantique physique (temps, capteur, fréquence d'émission entre autres) permettant ainsi d'associer ce signal reçu au signal d'origine émis. Une tâche est un nid de boucles parfait permettant de parcourir ce tableau pour en extraire les données (ou des paquets de données, appelés *motifs*) utiles pour l'exécution de la TE qu'il contient.

[2]Nous parlerons indifféremment d'*algorithme* ou d'*application* TSS pour désigner ce traitement.

Définition 1.1-1 : Nid de boucles parfait
Un nid de boucles parfait est une suite d'instructions d'affectation englobée par au moins une boucle.

Cette TE écrit à son tour dans un tableau multidimensionnel qui servira de source d'entrée à la tâche suivante. On définit ainsi le parcours des données dans l'application. Sa représentation en graphe est appelée graphe flots de données (GFD) ou graphe des précédences. Il est orienté et sans circuit. Par abus de langage nous parlerons de graphe acyclique.

Les tableaux dans lesquels sont stockées les données sont en assignation unique (une unique écriture pour chaque cellule de ces tableaux). De ce fait, un tableau n'est initialisé que par une et une seule tâche qui n'utilise pas les données qu'elle produit. Bernstein a donné trois conditions à vérifier pour s'assurer de l'absence de dépendance entre deux instructions x et y ([11]) :

1. y ne lit aucune variable modifiée par x ;
2. y ne modifie aucune variable lue par x ;
3. x et y ne modifient pas les mêmes variables.

En supposant que l'instruction x soit exécutée avant y, ces conditions permettent de catégoriser les dépendances :

Flow dependence : (Dépendance de flot) y lit au moins une variable modifiée par x ;

Output dependence : (Dépendance d'écriture) y modifie au moins une variable modifiée par x ;

Input dependence : (Dépendance de lecture) y lit au moins une variable lue par x ;

Anti-dependence : (Anti-dépendance) y modifie au moins une variable lue par x ;

Pour en revenir à nos nids de boucles, il n'y a pas de dépendances d'écriture ni dans un nid de boucles, ni entre deux nids de boucles. Les seules dépendances existant entre tâches sont les dépendances de flots et les dépendances de lecture.

Comment ces nids de boucles sont-ils structurés ? Pour simplifier, faisons abstraction du domaine applicatif. Soit l'exemple suivant représentant une tâche :

```
for i1 = 0, N1
  for i2 = 0, N2
   \
    for in = 0, Nn
       CALCUL(IN, OUT)
    end for
   /
  end for
end for
```

Nid de boucles principal.

```
procedure  CALCUL(IN:array, OUT:array)
  S = 0
  for m = 0, M
    S = S + IN[f2(i1,...,in,m)]
  end for
  OUT[f1(i1,...,in)] = S
end procedure
```

Appel procédural.

Avant d'aller plus loin, donnons quelques définitions.

Définition 1.1-2 : Vecteur d'itérations

Le *vecteur d'itérations* d'un nid de boucles parfait est le vecteur dont les composantes sont les itérateurs des boucles constituant ce nid. La première composante est l'itérateur de la boucle la plus externe. La dernière composante est l'itérateur de la boucle la plus interne. Une instance de ce vecteur correspond à une itération du nid de boucles. L'ordre défini sur cet ensemble d'instances est l'ordre lexicographique.
Si l'on reprend le petit programme exemple, $i = (i_1, i_2, \ldots, i_n)^t$ est le vecteur d'itérations du nid de boucles principal.

Définition 1.1-3 : Domaine d'itérations d'une boucle

Le *domaine d'itérations* d'une boucle est l'ensemble des valeurs que peut prendre son itérateur.
Le domaine d'itérations de la boucle i_1 est $[0, N_1]$.

Définition 1.1-4 : Espace d'itérations d'un nid de boucles parfait

L'*espace d'itérations* d'un nid de boucles parfait est le produit cartésien des domaines d'itérations des boucles qui le composent.
L'espace d'itération de i est $[0, N_1] \times [0, N_2] \times \ldots \times [0, N_n]$.

Définition 1.1-5 : Vecteur d'indexations d'un tableau

Le *vecteur d'indexation* (ou vecteur d'accès) est l'image du vecteur d'itérations par une application affine de ses composantes. Cette application est propre à un tableau. Le *vecteur d'indexation* donne l'accès à une donnée ou à un ensemble de données, stockées dans ce tableau.

Nous pouvons constater dans notre exemple la présence de deux niveaux de nid de boucles. Le premier, englobant l'appel procédural `CALCUL`, sera qualifié d'*externe*[3]. Le second, contenu dans l'appel procédural `CALCUL`, sera qualifié d'*interne*.

Le nid de boucles externe est parfait et totalement parallélisable. C'est-à-dire que l'ordre de parcours de son espace d'itérations n'a pas d'importance. De plus, les applications affines associées aux tableaux (que ce soit pour des accès en lecture ou en écriture) sont représentables par des matrices diagonales[4] définies positives. Il y a bijection entre l'ensemble des instances du vecteur d'itérations externe et les ensembles des instances des vecteurs d'indexations externes des tableaux. Nous parlerons de *pavage* et noterons sa matrice Ω_{pav} (*pav* pour *paving*) pour qualifier le parcours des tableaux par ce nid de boucles.

Le nid de boucles interne est associé à un tableau. Il définit le plus petit ensemble de données d'un tableau que l'on utilise par itération de l'appel procédural. Nous appellerons cet ensemble *motif*. Est associée à cet ensemble une fonction d'accès, dont la matrice est également diagonale à une permutation près et définie positive. Elle ne donne accès qu'à une et une seule donnée du motif à la fois. Cette fonction est appelée *ajustage* de part son rôle d'ajuster le pavage du nid de boucles externe pour obtenir l'élément atomique d'un tableau : une donnée. Sa matrice sera notée Ω_{fit} (*fit* pour *fitting*).

L'association du pavage et de l'ajustage caractérise la fonction d'accès globale des données d'un tableau pour une tâche. Elle sera notée Ω, et $\Omega = (\Omega_{pav} \quad \Omega_{fit})$.

Les procédures de traitement du signal sont séquentielles et optimisées pour fonctionner en tant que telles. Ce sont les appels à ces procédures (*i.e.*, le nid de boucles externe) que nous

[3] Dans toute la suite, lorsque nous parlerons de nid de boucles, le terme *externe* sera implicite.

[4] à une permutation près, permettant d'associer la sémantique physique portée par l'itérateur à la dimension du tableau correspondant à cette sémantique.

allons chercher à paralléliser. Si les nids de boucles externes sont totalement parallélisables, où est la difficulté ? Elle tient en la résolution globale du placement faisant intervenir des problèmes de granularités différentes à forte combinatoire.

1.2 Qu'est-ce que le placement ?

La problématique du placement pour THALES consiste à utiliser au mieux les ressources d'une machine parallèle pour exécuter une application de TSS. Les algorithmes de filtrage composant ces applications sont très simples (en terme de complexité algorithmique) mais doivent traiter énormément de données. Nous parlerons de traitement de données intensif. Le parallélisme recherché est un parallélisme de données. C'est-à-dire que résoudre ce problème consiste à trouver une bonne distribution des données sur les processeurs.

Comme cela a été présenté dans la section 1.1, rappelons qu'une application TSS est caractérisée par un ensemble de nids de boucles totalement parallélisables. Chacun de ces nids trouve en son cœur un appel procédural à une bibliothèque de fonctions élémentaires optimisées de traitement du signal. Ces fonctions manipulent les tableaux multi-dimensionnels dans lesquels sont stockées les données provenant d'un radar ou d'un sonar. Comme les accès à ces données se font via les itérateurs de boucles, définir une distribution des données consiste à définir un partitionnement de cet espace d'itérations.

Ce partitionnement consistera à «découper» les boucles de façon à n'exécuter sur chaque processeur que des sous-ensembles de l'espace d'itérations initial.

Soit le petit programme de la figure 1.2. Son espace d'itérations est défini par le seul intervalle d'itération de `i`, c'est-à-dire [0, 15]. Un partitionnement possible, en supposant que l'on désire utiliser 4 processeurs, pourrait être de construire 4 intervalles à partir de l'intervalle initial, et de les distribuer sur chaque processeur. Ainsi, un processeur aurait la charge d'exécuter les itérations 0 à 3, un deuxième 4 à 7, un troisième 8 à 11 et le dernier 12 à 15.

FIG. 1.2 – Exemple d'un nid de boucles à une boucle.

```
for i = 0, 15
   T1[i] = T0[i] * T0[i]
end for
```

La détermination de ce partitionnement serait triviale s'il ne s'agissait pas d'algorithmes de filtrage agissant sur des données provenant d'un radar ou d'un sonar. En effet, un radar (ou un sonar) fonctionne en continu et fourni des données continuellement. Dès lors, il n'est pas possible d'attendre que le premier nid de boucles ait fini son traitement pour entamer le deuxième, et ainsi de suite, puisque ce premier traitement n'aboutit jamais. De ce fait, le découpage de l'espace d'itérations de chaque nid de boucles va permettre d'exécuter des blocs de chacun sans attendre qu'ils aient tous achevé leur tâche.

Une fois ceci établi apparaît alors deux autres problèmes, le premier est qu'il faut impérativement respecter les dépendances entre nids de boucles. C'est-à-dire que lorsque l'on

va utiliser une partie d'un tableau, il est nécessaire qu'elle ait été définie au préalable. Le deuxième problème est qu'il va falloir choisir un ordre d'exécution des différents blocs des différents nids de boucles pour que ces dépendances soient respectées.

FIG. 1.3 – Le placement dans son expression la plus simple.

À ce stade, nous pouvons dire que le problème général du placement se limite à trouver un ordonnancement des différentes partitions des nids de boucles respectant les dépendances existant entre ces nids (illustré par la figure 1.3). Il reste cependant deux aspects à prendre en compte pour obtenir un placement satisfaisant :

1. l'aspect architectural ;
2. l'aspect opérationnel.

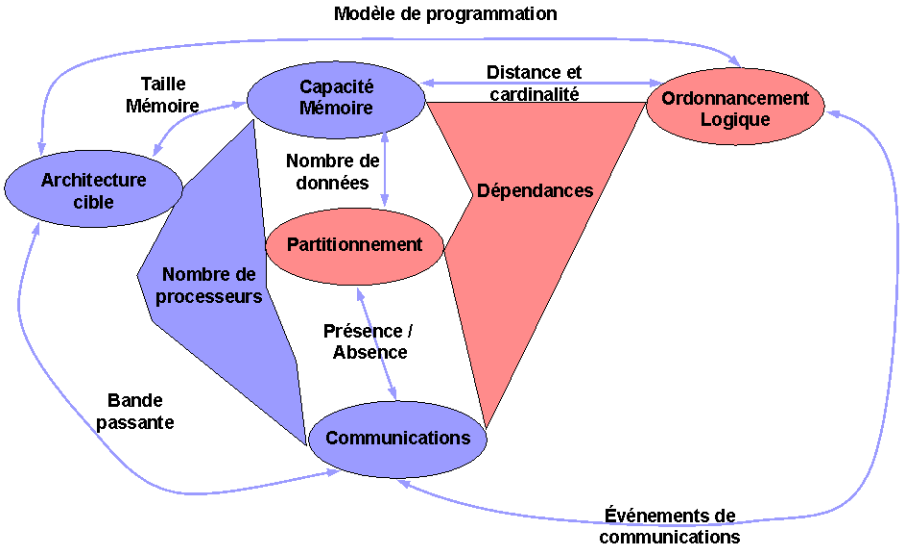
Le premier est important du fait de la définition du placement qui *consiste à trouver l'adéquation entre l'application et la machine cible*. C'est là que l'on définit la machine, en termes de nombre de processeurs, de types et tailles mémoires, distribuées ou partagées, d'interconnexion des processeurs, de débit du réseau de communication, de modèle de programmation (selon la classification de Flynn SIMD, SPMD, MIMD et par extension M-SPMD, MPMD)[5].

Cette description de la machine cible va définir un ensemble de contraintes qu'il va falloir respecter. Par exemple, le nombre de processeurs utilisés pour la parallélisation ne peut excéder la quantité de processeurs disponibles. De même pour la mémoire ou les communications. Ainsi, la nature de notre problématique du placement est quelque peu modifiée. Nous passons d'un problème général de placement d'applications, à un problème de placement d'applications *sous contraintes de ressources* où les ressources sont celles de la machine. Ces informations architecturales sont directement liées au partitionnement puisque le nombre de processeurs influence le nombre de partitions et la capacité mémoire influence la taille des partitions. Le mode de programmation influence quant à lui la façon dont le partitionnement va être défini. Par ailleurs, les échanges de données entre processeurs vont dépendre de la façon dont les calculs (*i.e.*, les données) sont répartis. Les interactions entre le placement (*partitionnement + dépendances + ordonnancement*) et la machine sont illustrées par la figure 1.4 (sur ce schéma, "nombre de processeurs" et "Dépendances" sont des hyper-liens).

Le deuxième aspect élargit la notion de «ressource» évoquée ci-dessus en ajoutant d'autres types de contraintes à respecter, à savoir principalement des contraintes de respect du temps-réel. L'algorithme TSS a pour unique objet la détection d'une éventuelle cible (fig. 1.1). Il est donc évident que le temps de réaction entre le moment où le signal arrive au radar et le moment où un opérateur décide de réagir doit être très rapide. Un radar moins efficace que l'œil humain est inutile. Par conséquent, la durée des calculs et des communications, ainsi que le temps nécessaire à une donnée pour traverser le GFD (*i.e.*, la latence) sont des points

[5]Dans le contexte étudié, *i.e.*, le traitement de données intensif, nous serons plus attachés aux modèles de programmation SPMD, SIMD et M-SPMD, qu'aux deux autres relevant plus de la parallélisation de tâches.

FIG. 1.4 – Prise en compte de la dimension architecturale dans la problématique du placement.



cruciaux pour garantir des temps de réponse. De plus, les données d'entrées arrivent à une certaine cadence qu'il faut également prendre en considération pour assurer un traitement de ces données dans des temps raisonnables (pour éviter entre autres choses que les buffers d'entrées ne se remplissent plus vite qu'ils ne se vident).

Tout ceci nous permet de donner une définition du placement :

Définition 1.1-6 : Le placement dans THALES

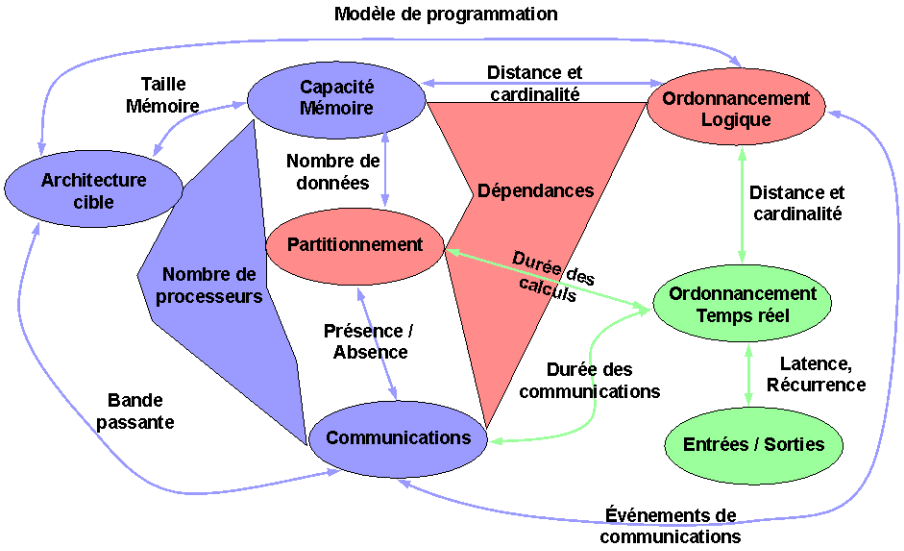
> Le placement d'une application TSS sur une machine parallèle est la recherche d'un partitionnement des nids de boucles composant cette application et d'un ordonnancement de ces partitions respectant : leurs dépendances, les contraintes architecturales de la machine cible et les contraintes opérationnelles d'une application temps-réel.

La figure 1.5 est un résumé schématique de la définition donnée ci-dessus. Notre approche va consister à associer un modèle formel à chacune des bulles de ce schéma (indépendamment les uns des autres), les arcs exprimant les liens entre ces modèles. Nous obtenons ainsi un ensemble de variables et de contraintes sur ces variables. Un modèle sera constitué de variables et de contraintes qui lui sont propres. Le partage de variables, ou des contraintes faisant intervenir des variables de modèles différents, constituent les liens. De ce fait, c'est tout naturellement que nous nous sommes tournés vers la Programmation Par Contraintes (PPC) comme méthode de résolution. Nous avons nommé PLC^2 la combinaison de ces deux méthodes pour la résolution du placement[6].

Du fait que nous rendons tous les paramètres, caractérisant la fonction placement que

[6] PLC^2 n'est pas un acronyme, mais un jeu de mots impliquant la technologie initialement utilisée *PLC* et les trois paramètres principaux du placement p, l et c (que nous définissons dans la section 3 p. 43). Ce nom a été choisi lors de la thèse de Christophe Guettier [32], à laquelle nous faisons souvent référence dans ce document.

FIG. 1.5 – La problématique du placement dans son ensemble.

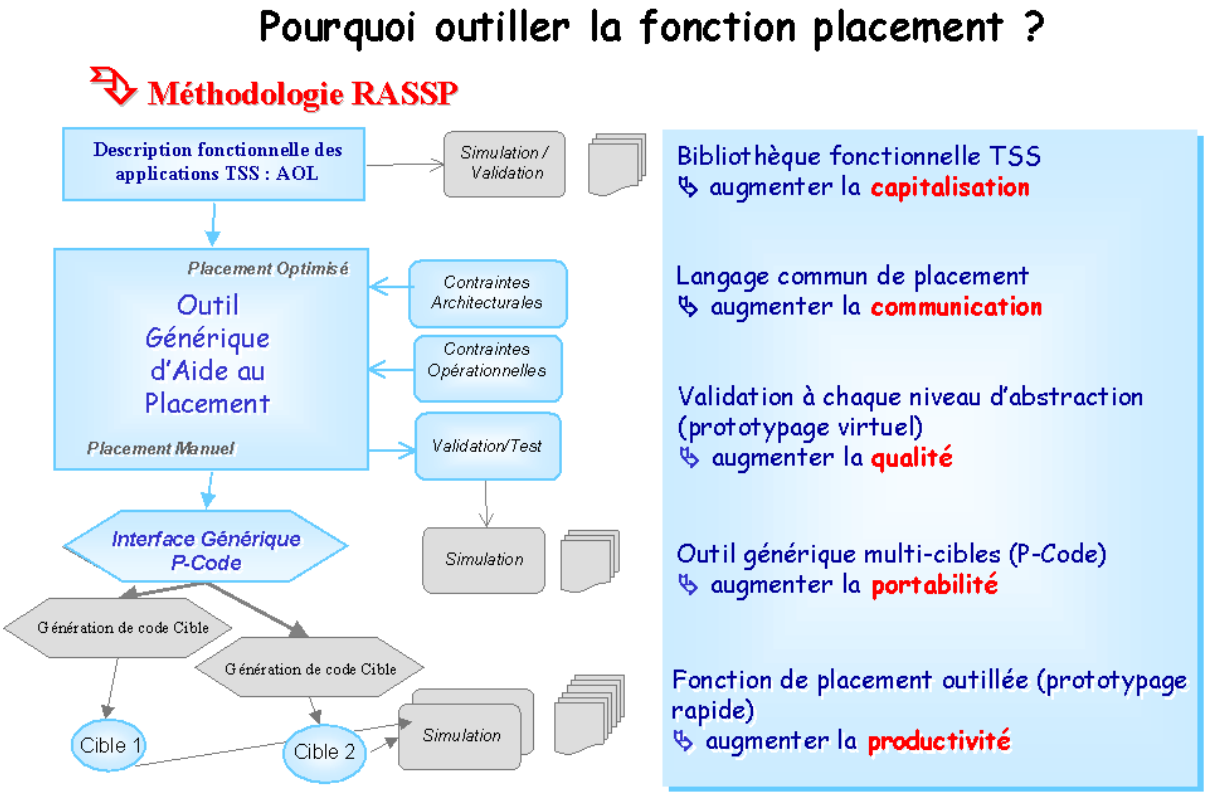


nous venons de définir, simultanément variables, la résolution globale de cette fonction est très fortement combinatoire. En plus de cette combinatoire, chaque modèle peut comporter un niveau de granularité différent et complexe à formaliser. L'approche multi-modèles et la PPC vont nous permettre de réduire cette complexité cumulée.

2 PLC^2 : une approche multi-modèles

PLC^2 est une méthodologie issue de travaux conjoints THALES et ÉCOLE DES MINES DE PARIS [32]. Elle s'inscrit dans un schéma identique à celui issu du programme RASSP, illustrée par la figure 1.6 (Chap. 2, sec. 1). Elle repose sur une approche multi-modèles et utilise la Programmation Par Contraintes comme méthode de résolution. Cette méthodologie est supportée par l'outil APOTRES développé par TRT-France.

FIG. 1.6 – Le programme RASSP au service du placement



2.1 Originalité de cette approche

Le problème étudié est celui de l'optimisation globale du placement automatique d'applications de TSS sur machines parallèles. Cette optimisation globale porte sur l'ensemble de la chaîne de traitement du signal, du placement des données, et de l'ordonnancement des calculs et des communications en prenant en compte les contraintes architecturales et opérationnelles. Pour résoudre ce problème complexe, une approche originale a été choisie en lieu et place d'une approche algorithmique : la modélisation concurrente.

La modélisation concurrente est vue comme un ensemble de modèles, un pour chaque

constituant qu'il soit fonctionnel ou physique. **Un modèle doit être vu sémantiquement comme l'ensemble des spécifications du comportement du constituant qu'il modélise**.

Les propriétés du paradigme relationnel ont des conséquences immédiates sur les propriétés des composants logiciels. Ceci est essentiellement dû au fait que les modèles constituent eux-mêmes la réalisation.

Les propriétés du paradigme relationnel sont :

Description formelle :

Les relations représentent une description formelle du comportement du constituant. En effet, pour chacune des sous-fonctions du placement, une modélisation mathématique a été spécifiée formellement. Dans la plupart des cas, elle est issue des travaux de la communauté parallélisation mais a été adaptée non seulement au cadre applicatif qu'est le traitement du signal mais étendue au contexte de la modélisation concurrente.

Adirectionalité :

La relation permet d'abandonner le paradigme fonctionnel basé sur la distinction des entrées/sorties. Une relation assure une réversibilité totale des arguments. Ceci, permet de ne faire la distinction qu'à l'exécution, en fonction de la nature des arguments (connus et inconnus). Concrètement, une relation entre le dimensionnement de la machine et le débit permet, selon l'information que l'on a, de déterminer le débit (si l'on a les ressources de la machine) ou les ressources de la machine (si l'on a le débit).

Compositionalité :

La composition des modèles relationnels est tout simplement la conjonction logique des relations qui constituent le modèle. Ceci implique une sémantique simple de la compositionalité. L'ensemble des solutions d'un modèle composite est tout simplement l'intersection des solutions des modèles.

Elles contribuent à la généricité du programme. Les propriétés du logiciel induites sont alors :

Un domaine d'utilisation élargi : Un modèle peut être utilisé dans plusieurs contextes en fonction du but à réaliser.

Une interchangeabilité accrue : Un modèle peut être modifié ou complètement redéfini par la donnée d'une nouvelle spécification sans avoir à intervenir sur les autres modèles.

Une compositionalité intrinsèque : Le modèle d'un système est construit à partir du modèle de ses composants.

Une maintenabilité simple : La maintenabilité reste locale à chaque modèle.

Une extensibilité aisée : Étendre un système revient à le composer avec l'extension souhaitée.

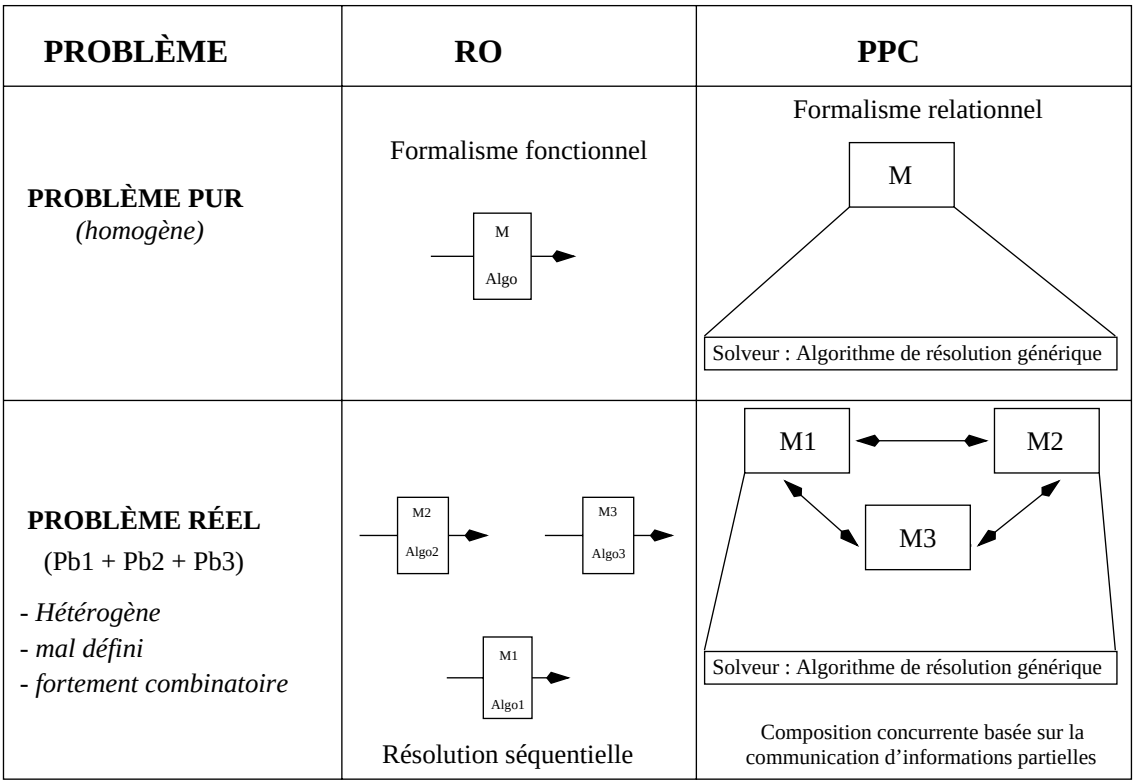
La Programmation Par Contraintes vient alors naturellement comme moyen de résolution pour une approche multi-modèles.

2.2 La Programmation Par Contraintes comme système de résolution

La résolution du placement d'applications industrielles ne se cantonne pas à une problématique bien définie, mais intègre la combinaison de plusieurs sous-problèmes. En effet,

nous avons souvent à résoudre des problèmes d'optimisation combinatoire sur des problèmes multi-composants, multi-fonctions dans lesquels les contraintes sont très hétérogènes et où il faut considérer les différents éléments à différents niveaux de granularité. La PPC offre des solutions permettant **la coexistence de modèles se recouvrant partiellement, se coordonnant et se décomposant** (fig. 1.7).

FIG. 1.7 – La PPC versus la RO



La recherche opérationnelle (RO) et l'intelligence artificielle (IA) ont pour préoccupation commune, la résolution de problèmes à forte combinatoire. Cependant, la RO semble plus adaptée à l'étude de problèmes spécifiques, souvent simplifiés par rapport aux problèmes réels et pour lesquels il s'agit de trouver l'algorithme aussi efficace que possible. La PPC semble plus adaptée à résoudre des problèmes hétérogènes par le biais d'une formalisation «fine» des différents modèles et de leurs relations.

Une alternative pour les problèmes multi-composants, multi-fonctions :

La spécification de modèles pour chaque composant et pour chaque fonction qui se combinent ensuite lors de la résolution d'un but, permet d'apporter une alternative à la résolution de problèmes de systèmes. De plus, les modèles sont souvent très hétérogènes. Certains peuvent s'exprimer exclusivement à l'aide de contraintes linéaires, d'autres peuvent nécessiter des contraintes symboliques ou booléennes.

La PPC permet, indépendamment de l'hétérogénéité des contraintes, par de simples interactions locales, de garantir une coordination globale du système.

Une alternative pour les problèmes d'optimisation combinatoire :

La résolution des problèmes hautement combinatoires se résume rarement à une seule formalisation mathématique. Souvent, pour ne pas dire toujours, des formalisations

complémentaires sont nécessaires. Un exemple pourrait être les formalisations redondantes qui tirent avantage des propriétés des solutions partielles, un autre exemple serait la prise en compte des symétries du problème, etc.

Le problème qui se pose alors est d'utiliser en même temps toutes ces informations. Dans le contexte d'approches plus classiques, comme la RO ou la programmation en nombres entiers, cette étape s'avère toujours très délicate. En effet, les programmes écrits dans un langage impératif nécessitent un temps de développement non négligeable et sont souvent difficiles à étendre et à modifier.

La PPC apparaît pour ce type de problème non seulement comme **un bon outil d'expérimentation** mais permet, au moment de la résolution, l'**utilisation concurrente de tous les modèles redondants**.

Une alternative pour un raisonnement à plusieurs niveaux de granularité :

L'efficacité de la réalisation finale dépend de trois choses. Tout d'abord, elle dépend crucialement de l'efficacité du système de contraintes utilisé, du contrôle que l'on a sur la recherche d'une solution, mais aussi, considérablement, de la granularité considérée dans la modélisation. L'expérience montre qu'il est indispensable de considérer différents niveaux de granularité et de pouvoir raisonner sur les différents niveaux. Ici encore, la PPC offre une alternative en permettant la description, la coexistence et la coordination de modèles à différents niveaux de granularité.

Comparée aux méthodes classiques utilisées dans la résolution de problèmes complexes, l'approche Programmation Par Contraintes offre des avantages clairs incluant une plus grande **flexibilité**, **plus d'interactions** avec un utilisateur, sans pour autant sacrifier **l'efficacité**. De plus, cette technique offre des solutions plus génériques et plus robustes qui ont un impact important sur le **coût d'évolution et de mise à niveau** du logiciel.

Toutefois, il est à noter que plus le problème est complexe, comme c'est le cas pour le placement d'applications sur machine parallèle, plus une collaboration entre spécialistes du domaine applicatif et spécialistes de la technique PPC est nécessaire. En effet, le succès d'une telle approche réside dans le degré de modélisation des différents composants du problème et de leur spécialisation pour qu'ils soient compatibles avec le cadre arithmétique de la PPC.

3 La Programmation Par Contraintes

La PPC est directement issue de la Programmation Logique avec Contraintes (PLC). *D'où vient la PLC ?*, *Quels sont ses fondements formels ?*, *Comment en arrive-t-on à la PPC ?*, *Comment fonctionne un moteur de contraintes ?* sont les questions auxquelles nous allons tenter de répondre brièvement dans cette section. L'annexe A agrémente cette section. Il s'agit d'un article présentant d'avantage l'interaction qu'il peut y avoir entre la technologie de Programmation Par Contraintes et le monde industriel.

3.1 La programmation logique avec contraintes

La programmation logique avec contraintes (PLC) constitue une «technologie parapluie» qui recueille plusieurs concepts issus de grands domaines tels que la logique (théories du premier ordre, décidabilité ou indécidabilité), l'interprétation abstraite (opérateurs de Galois), les systèmes formels (calcul des prédicats du premier ordre, arithmétique formelle encore appelée algèbre de Peano), de la recherche opérationnelle (RO), de l'intelligence artificielle (IA), et de la programmation logique ou programmation mathématique.

La logique des prédicats, déjà utilisée comme langage de spécifications, a tout d'abord donné naissance à la programmation logique qui utilise le concept de déduction comme paradigme de calculabilité.

La PLC repose alors sur le principe général de remplacer l'opérateur d'unification sur les termes de la programmation logique par la satisfaction de contraintes sur ces domaines dans une ou plusieurs structures mathématiques fixées mais adéquates. Seules les structures décidables sont prises en compte : l'arithmétique linéaire entière ou réelle, les algèbres d'arbres, par exemple. C'est un concept de programmation dans lequel un problème posé est modélisé par un ensemble de variables mathématiques et de relations définies par des contraintes et des symboles de prédicats définis par les expressions du langage.

De manière plus formelle, l'axiomatique d'un langage PLC est définie par un ensemble V de variables, un ensemble S_F dénombrable de symboles de constantes et de fonctions, un ensemble S_C dénombrable de symboles de prédicats supposé contenir *true* et $=$. Une contrainte atomique est alors une proposition atomique de ce langage. Le langage des contraintes est alors défini comme la fermeture par conjonction et par quantification existentielle d'un ensemble de formules du premier ordre contenant les contraintes atomiques.

Les langages de PLC sont alors paramétrés par au moins une structure mathématique qui fixe l'interprétation (en logique on parlera plutôt de réalisation) du langage de contraintes. Cette structure $\mathcal{S}$ est entièrement caractérisée par le quadruplet $\mathcal{S} = (\mathfrak{D}, \mathcal{E}, \mathcal{O}, \mathcal{R})$ et par la fonction s où $\mathfrak{D}$ est un ensemble appelé domaine[7], $\mathcal{E}$ est un sous-ensemble de $\mathfrak{D}$ d'éléments associés à chaque symbole de constantes de S_F (i.e. d'arité 0), l'ensemble $\mathcal{O}$ est l'ensemble des opérateurs sur $\mathfrak{D}$ associés à chaque symbole de fonctions de S_F en respectant les arités $\mathfrak{D}^n \rightarrow \mathfrak{D}$, l'ensemble $\mathcal{R}$ de relations sur $\mathfrak{D}$ associés à chaque symbole de prédicat de S_C en respectant les arités $\mathfrak{D}^n \rightarrow \{0,1\}$ et la fonction s : V $\rightarrow \mathfrak{D}$ qui s'étend aux termes par morphisme. De plus, on suppose que dans $\mathcal{S}$, le problème de satisfiabilité des contraintes est décidable[8]. Cette structure $\mathcal{S}$ peut être représentée par une théorie $\mathcal{T}$ du 1er ordre défini sur

[7] On parle de domaine par abus de langage vis à vis des domaines des valeurs, mais $\mathfrak{D}$ n'est pas nécessairement un domaine au sens topologique du terme (i.e. convexe connexe).

[8] Il existe un algorithme qui permet de décider si pour toute contrainte c du langage on a $\mathcal{S} \models \exists s(c)$ ou

l'alphabet S_F et S_C et vérifiant :

1. $\mathcal{S} \models \mathcal{T}$ (correction)
2. pour toute contrainte c, soit $\mathcal{T} \models \exists c$ soit $\mathcal{T} \models \neg\exists c$ (complétude pour la satisfaction de contraintes).

Sous ces hypothèses, pour toute contrainte c, on a

$$\mathcal{S} \models \exists s(c) \Leftrightarrow \mathcal{T} \vdash \exists c$$

Chaque structure $\mathcal{S}$ définit alors un langage de PLC distinct.

Signalons les plus classiques :
- CLP($\mathbb{R}$) où $\mathcal{S}$ est l'arithmétique réelle ($\mathbb{R}$,0,1,+,*, =, <)
- CLP($\mathbb{N}$) où $\mathcal{S}$ est restreint à l'arithmétique de Presburger définie par ($\mathbb{N}$,0,1,+, =)
- CLP(FD) sur les domaines finis, i.e. l'ensemble des intervalles bornés d'entiers muni des prédicats de l'arithmétique usuelle ainsi que d'autres prédicats définissant des contraintes dites symboliques comme element(I,[x_1,...,x_n],V) qui est *true* si x_I=V (avec I et V inconnues).

Dans toute la suite, nous travaillerons sur les domaines finis et donc nous utiliserons CLP(FD).

La nuance séparant la PLC de la PPC repose sur le langage de programmation utilisé pour représenter le formalisme décrit ici. La PLC utilise des langages de programmation logique (Prolog) [25] alors que la PPC utilise des langages plus impératifs (C++ ou Claire) [37][47]. Le solveur PPC utilisé pour réaliser cette thèse est celui développé par Thales : Eclair.

3.2 De la théorie à la pratique

Un problème en Programmation Par Contraintes se présente sous la forme d'un ensemble de relations (contraintes) entre variables ayant chacune un domaine[9]. Ce dernier est l'ensemble des valeurs possibles qu'une variable peut prendre. Une solution est une instanciation cohérente (*i.e.*, qui respecte toutes les contraintes) de ces variables. De ce fait, la solution est construite petit à petit, puisqu'à chaque fois que l'on va attribuer une valeur à une variable, la cohérence de l'ensemble est vérifiée. Si bien que si l'on représente l'espace des instances possibles par un arbre (nœud = variable, arc = valeur), alors les branches allant de la racine jusqu'aux feuilles de cet arbre (correspondant à des instanciations de toutes les variables) sont les solutions du problème.

Cette méthode constructive repose sur deux mécanismes :
- un mécanisme de raisonnement déductif : la propagation des contraintes ;
- un mécanisme de raisonnement hypothétique : l'énumération des valeurs.

Le premier, appelé TELL, a pour rôle le maintien de la cohérence partielle du système. Il a pour but d'éliminer les valeurs des domaines des variables pour lesquelles il est sûr qu'elles ne peuvent appartenir à une solution.

La Programmation Par Contraintes, par opposition au monde des CSP[10] qui a une vue plus «algorithmie généraliste», a ceci de particulier qu'elle n'utilise pas un algorithme de propagation généraliste (type arc-cohérence comme AC{4,5,6,7} [53][24][12][13] par exemple),

$\mathcal{S} \models \neg\exists s(c)$

[9] Ce domaine est représenté par un ensemble de valeurs ou par un intervalle.

[10] Constraints Satisfaction Problems

mais un algorithme propre à chaque contrainte. La dimension langage de la Programmation Par Contraintes permet de profiter de la sémantique de la contrainte pour déduire plus d'information. Cette information se traduit par une réduction des domaines des variables.

Le second mécanisme, appelé ASK, fait des hypothèses sur les valeurs des variables. Il construit l'arbre de recherche, à l'aide d'heuristiques pour ordonner les variables et leurs valeurs[11]. Après chaque décision il fait appel au TELL pour valider ou réfuter cette décision.

Ces deux mécanismes sont donc intimement liés et s'exécutent jusqu'à ce que toutes les variables soient instanciées à des valeurs respectant toutes les contraintes.

Voici un algorithme simple permettant d'illustrer ce fonctionnement :

Propagation initiale À chaque nouvelle contrainte posée, on vérifie sa cohérence avec les autres. Cette étape a pour effet de réduire les domaines des variables. S'il arrive que le domaine d'une variable soit réduit à l'ensemble vide, alors le problème n'a pas de solution. Une fois toutes les contraintes posées, si le système n'est pas déclaré incohérent, et qu'il reste malgré tout des variables non instanciées, on passe à l'étape suivante : la recherche de solution (appelé également *énumération*).

Sélection d'une variable On choisit une variable v dont le domaine $\mathfrak{D}_v$ n'est pas vide. On fixe un point de choix : pour la variable sélectionnée, on choisit une valeur **val** de son domaine ; on enregistre le fait qu'un choix a été fait et qu'il existe d'autres possibilités ; on sauvegarde les domaines de toutes les variables avant d'essayer de fixer v à la valeur choisie.

Sélection d'une valeur pour la variable v sélectionnée

1. Fixer v à **val** ;
2. Propager l'information à toutes les contraintes où v apparaît ;
3. Si le domaine d'une autre variable se réduit à l'ensemble vide c'est que la valeur **val** choisie n'est pas bonne. Il faut revenir en arrière, au point de choix, restaurer les domaines des variables, enlever **val** de $\mathfrak{D}_v$, choisir une autre valeur et reprendre à l'étape 1.
4. Si aucun domaine n'est réduit à l'ensemble vide, appliquer l'algorithme récursivement à partir de l'étape `Sélection d'une variable`.

Si toutes les valeurs du domaine de v conduisent à un échec, revenir au point de choix précédent et appliquer l'étape 3 décrite ci-dessus à la variable v' correspondant à ce point de choix. Dans le cas où il n'y a plus de point de choix alors il n'y a pas de solutions.

Ce mécanisme permet de trouver une solution à un problème. Cependant, il arrive que la difficulté, ou plus simplement l'intérêt, ne soit pas de trouver une solution mais une solution optimale par rapport à un critère donné, éventuellement plusieurs s'ils sont ordonnés par préférence et en appliquant un ordre lexicographique pour comparer deux solutions[12].

Généralement, l'algorithme utilisé pour l'optimisation est le Branch & Bound. Selon le compromis choisi entre rapidité et qualité, il est possible de rendre cet algorithme incom-

[11] Cette construction peut être dynamique selon les heuristiques choisies. Par exemple, si l'on utilise l'heuristique min_domain (qui choisit les variables de plus petit domaine en premier), une variable peut avoir un domaine de cardinal plus petit qu'une autre après une propagation, alors que c'était l'inverse avant cette propagation.

[12] Les différentes possibilités de collaboration de techniques d'aide à la décision multi-critères [29] et la Programmation Par Contraintes sont actuellement étudiées à THALES.

plet en utilisant différentes stratégies de façon à obtenir rapidement une solution voisine de l'optimum, *i.e.*, satisfaisante à partir d'un certain seuil (en utilisant l'algorithme LDS par exemple, [36]).

3.3 Un exemple

Considérons trois variables X, Y et Z définies respectivement sur les domaines $[1, 40]$, $[-20, 10]$ et $[-100, 81]$. Le problème à résoudre est modélisé par les deux contraintes suivantes : $X < Y$ et $X \times Y = Z$. Le mécanisme de propagation décrit ici s'effectue sur des domaines définis par leur bornes (et non pas en extension), ce qui n'autorise pas les «trous». On parlera alors de *cohérence aux bornes*.

Trouver une solution à ce problème signifie trouver une valeur pour chacune des variables X, Y et Z telle que les contraintes sont respectées. La première étape va donc consister à poser ces contraintes. L'ordre dans lequel elles le sont a tout au plus un impact sur la durée des propagations initiales. Changer cet ordre ne change pas l'état dans lequel le système se trouve à l'issu de ces propagations initiales.

Nous choisissons de poser en premier la contrainte $X < Y$. Sa propagation a pour effet de modifier les domaines respectifs de X et Y en $[1, 9]$ pour X et en $[2, 10]$ pour Y.

La deuxième contrainte ($X \times Y = Z$) impose à Z de se trouver dans l'intervalle $[2, 90]$. Ainsi la borne inférieure de son domaine passe de -100 à 2. Or la borne supérieure de Z est inférieure à 90 et vaut 81. Par conséquent, il y a des valeurs dans les domaines de X et Y qui ne conduiront pas à une solution. En l'occurrence, c'est la borne supérieure de Y qui va être réduite. Son domaine devient $[2, 9]$.

La propagation ne s'arrête pas ici car le domaine de Y ayant été modifié, il faut vérifier qu'il est toujours cohérent avec le reste des contraintes posées. Ce qui n'est pas le cas. La cohérence de l'ensemble étant obtenu par élimination de valeurs, son maintien passe par la réduction du domaine de X. Ainsi, on a $X \in [1, 8]$. Ainsi de suite jusqu'à ce que toutes les propagations possibles ai été effectuées.

À la fin des propagations initiales, nos variables sont définies par :

$$X \in [1, 8]; \quad Y \in [2, 9]; \quad Z \in [2, 72]$$

Les domaines des variables n'étant pas réduits à des singletons à l'issu des propagations initiales, on entre dans le processus d'énumération. On choisit une variable, selon une stratégie particulière ou non, et on lui affecte une valeur (là aussi selon une stratégie particulière ou non). Disons que l'on ait choisi X et que l'on décide de lui affecter la valeur 2. Cette instanciation de X relance le mécanisme de propagation qui fait les déductions menant à l'état suivant :

$$X = 2; \quad Y \in [3, 9]; \quad Z \in [6, 18]$$

Aucune des variables n'a son domaine vide, l'affection de la valeur 2 à X n'aboutit donc pas à une contradiction. Cependant toutes les variables ne sont pas instanciées. On continue alors l'énumération avec une autre variable.

L'instanciation de Y à la valeur 3 conduit, au travers des propagations, à une affectation cohérente de toutes les variables, *i.e.*, une solution. C'est-à-dire

$$X = 2; \quad Y = 3; \quad Z = 6$$

Si l'on cherche toutes les solutions de ce problème, le processus continue jusqu'à avoir atteint toutes les feuilles de l'arbre de recherche (*i.e.*, parcours exhaustif de toutes les instanciations conduisant à une solution).

Dans le cadre d'une optimisation (selon un critère donné) ce processus continue mais ne retient que les solutions dont le coût est meilleur, jusqu'à atteindre l'optimum. Par exemple, on peut chercher la solution qui maximise un critère W défini par $W = Z - (X + Y)$. On assimile cette fonction de coût à une contrainte comme les autres. W a donc pour domaine $[-1, 55]$. On cherche une première solution et lorsqu'on l'a trouvée on relance le processus de recherche avec la contrainte supplémentaire $W >$ *coût_trouvé*. S'il n'y a pas d'autre solution alors l'optimum a été atteint lors de la recherche précédente. C'est le principe du *Branch & Bound*. Dans cet exemple précis l'optimum est atteint pour $X = 8$ et $Y = 9$ ($Z = 72$ et le coût de cette solution est $W = 55$).

La modélisation et l'étude de certains problèmes peuvent suffire à déterminer des conditions d'atteignabilités d'un optimum pour un critère donné. Dans [24], Van Henrenryck et Deville montrent que, dans le cas où un problème est décrit par des contraintes binaires, la cohérence d'arc aux bornes des domaines permet de déduire trivialement deux solutions : l'instanciation des variables à leur borne inférieure d'une part, à leur borne supérieure d'autre part. Dans le cas d'un problème d'ordonnancement, la première donne l'ordonnancement au plus tôt, et la seconde donne l'ordonnancement au plus tard.

3.4 Conclusion

La PPC, de part sa nature et son histoire, peut être vue comme un langage de modélisation. En effet, l'exploitation d'un problème décrit formellement, moyennant tout de même une certaine adaptation de cette formalisation aux contraintes dont on dispose, est déclarative. La résolution de cet ensemble de contraintes est un processus séparé de cette déclaration.

La tendance depuis quelques années est d'associer des solveurs différents, traditionnellement de recherche locale ou de Programmation Linéaire, à la PPC. Cette dernière est exploitée pour ses capacités de déclarativité d'un problème, de génie logiciel et ses algorithmes de propagations. La recherche locale apporte tout son savoir faire en matière d'algorithmes de résolution efficaces.

4 Structure du document

Cette introduction a présenté le cadre de travail dans lequel nous avons évolué. Ainsi ont été définies les applications que nous désirons paralléliser et ce que nous entendons par *placement*. L'originalité et les composantes de notre approche, la combinaison *modélisation concurrente* + P*rogrammation* P*ar* C*ontraintes*, ont également été mises en avant.

Le développement du travail effectué durant cette thèse est structuré de la façon suivante :

Le chapitre 2 présente un état de l'art des outils disponibles actuellement pour répondre au problème du placement. Ces outils feront l'objet d'une analyse afin de pouvoir les comparer et d'expliquer en quoi notre approche est complémentaire.

Le chapitre 3 pose le domaine applicatif sur lequel nous avons travaillé en présentant un mode radar (le mode Moyenne Fréquence de Récurrence) et la prise en compte de ses spécificités.

Le chapitre 4 donne une vue du domaine architectural, en présentant une machine et une modélisation du mode de programmation M-SPMD.

Le chapitre 5 a pour objet les améliorations effectuées sur certains modèles de [32] (mémoire et dépendances), ainsi que la présentation de nouveaux (entrées/sorties et détection des communications) afin de pouvoir résoudre des problèmes de moins en moins idéalisés.

Le chapitre 6 est plus appliqué puisqu'on y trouve les différents modèles tels qu'ils sont pris en compte par la PPC pour la résolution. La stratégie de résolution adoptée y est également présentée, après avoir expliqué comment nous avons exprimé les opérateurs pgcd et ppcm en contraintes.

Le chapitre 7 fait état des expérimentations et des résultats obtenus.

Le chapitre 8 conclut cette thèse avec une ouverture sur la génération de code de contrôle à partir des directives de placement obtenues. Un résumé des contributions y trouve également sa place.

Note : Les contraintes des modèles que nous avons implémentées dans notre prototype sont facilement repérables grâce au symbole ✓figurant dans la marge.

Chapitre 2

État de l'art

Un nombre toujours croissant d'applications embarquées temps réel fait appel à des traitements parallèles. Ceci résulte à la fois des possibilités offertes par les technologies d'intégration (ainsi Texas Instruments prévoit d'intégrer en 2010 quelques dizaines de processeurs sur une puce) et des types de performances demandées (en terme de qualité de service). En effet les applications de télécommunications, multimédia ou d'accès réseau font de plus en plus appel à du traitement de signal ou d'images intensif qui requièrent un fort niveau de parallélisme : il s'agit là de réaliser un grand nombre de fois la même opération sur des séquences de données vectorielles. Ce parallélisme de données (grain fin) et ce traitement vectoriel expliquent le renouveau d'intérêt actuel pour les architectures SIMD et leur intégration dans un grand nombre de nouvelles architectures de traitement. Cette approche est en outre un moyen de concilier modularité et puissance de calcul.

Le développement efficace de ces applications et leur programmation en langage de haut niveau n'est pas sans poser de nouveaux défis. En effet les techniques de compilation classiques n'apportent pas en général les performances attendues et l'espace de développement sur ces types d'architecture est sensiblement plus vaste : parallélisation des algorithmes, placement sur des architectures variées et implémentation sont des tâches nécessitant une méthodologie et des outils supportant itération et démonstration de preuve.

Si la phase de parallélisation consiste à extraire le parallélisme potentiel d'un programme séquentiel, la problématique du placement est alors d'établir la projection d'un algorithme (architecture logicielle d'application), dont le parallélisme est ou non spécifié, sur une architecture cible (architecture matérielle multi-processeurs). Cela de manière efficace (respect des contraintes temps réel, minimisation des ressources matérielles) qui se traduit par un problème d'optimisation fortement combinatoire. Tout cela est réalisable dans un cadre formel, où algorithme, architecture et implantation sont décrits par des modèles permettant d'analyser (vérifier), de composer et de réduire (optimiser) l'application étudiée.

Cet état de l'art ne prétend pas fournir une réponse définitive quant à la manière de choisir un outil ayant cette fonctionnalité mais seulement présenter :

- différents outils disponibles soit dans le monde académique soit dans le monde industriel (section 1)
- et différents aspects qui permettent de comparer les outils étudiés apportant ainsi une aide pour établir ce choix. (section 2). Les différents critères de comparaison sont : les

algorithmes à placer, les machines et les modèles de programmation cibles, les méthodes de résolution et leurs implications sur les algorithmes et les machines, les ensembles de solutions dans lesquels la recherche d'une solution optimale est effectuée, et enfin la qualité des estimations temporelles effectuées.

Finalement, la dernière section de ce chapitre présentera une synthèse des travaux de [32], dont les résultats forment le point de départ de cette thèse.

1 Quelques outils et méthodologies de placement

1.1 Méthodologie RASSP

Une solution pour le traitement des obsolescences a été identifiée dans le cadre du projet RASSP. En effet, aux États-Unis d'Amérique, le DoD a lancé en 1994 le programme RASSP (**R**apid prototyping of **A**pplication **S**pecific **S**ignal **P**rocessors). Ce programme a permis de mettre sur pied un environnement et une méthodologie de développement **de calculateurs de traitement du signal** avec une division par 4 du coût de développement.

La méthodologie proposée consiste à implanter une application de traitement de signal sur la technologie numérique la plus performante du moment, «Model Year».

L'approche «Model Year» applique un type de développement itératif à l'équipement complet. Le «Model Year», grâce à plusieurs cycles de développement, permet de lever les risques de principes d'un équipement en utilisant des calculateurs disponibles commercialement : technologies à base de cartes ou de machines COTS (prototypage rapide). En prenant en compte l'amélioration «constante» des composants numériques (loi de Moore), l'équipement final est développé avec la technologie du moment la moins chère. Cette approche se démarque fortement de l'approche de développement traditionnelle. Dans l'approche classique, l'évolution des technologies numériques entraîne des obsolescences de composants et donc des surcoûts. Dans l'approche «Model Year» l'amélioration de la technologie des composants numériques commerciaux permet de diminuer :

- les risques et les coûts en phase de développement d'un équipement ;
- les coûts de fabrication en phase de production.

Pour mettre en œuvre un développement de type «Model Year», il faut pouvoir très rapidement implanter une application sur une nouvelle cible. Pour cela, il faut disposer d'un ensemble de technologies qui facilitent le portage de l'application :

- des calculateurs commerciaux pour le développement des prototypes ;
- des bibliothèques de ressources matérielles et logicielles pour faciliter la réutilisation ;
- des standards d'interfaces matérielles et logicielles pour améliorer la portabilité ;
- un atelier développement des calculateurs qui permet :
  - la modélisation fonctionnelle,
  - le prototypage rapide,
  - le prototypage virtuel,
  - la synthèse du matériel et du logiciel.

Cette méthode suppose l'existence de calculateurs commerciaux optimisés traitements du signal pour le développement des prototypes. La réutilisation permet de diminuer et de sécuriser les temps de développement. Elles reposent sur des bibliothèques de ressources matérielles et logicielles. La mise en œuvre de l'approche «Model Year» est facilitée par l'utilisation d'architectures matérielles et logicielles modulaires et d'interfaces standard qui

favorisent la portabilité. Une approche itérative du développement des calculateurs est établie. À chaque itération, les phases de définition, développement et validation sont réalisées pour une étape du développement. Les différentes étapes sont :

1. la modélisation fonctionnelle,
2. le choix d'architecture,
3. le développement Matériel/Logiciel.

Cette approche itérative a pour but de valider les choix au plus tôt, en particulier avant d'avoir fabriqué le matériel, pour diminuer les risques, de réutiliser les résultats des itérations précédentes pour réaliser les itérations suivantes et diminuer ainsi les temps et les coûts de développement.

1.2 Méthodologie AAA, SynDEx

SynDEx est l'acronyme de **Syn**chronized **D**istributed **Ex**ecutive. Il est développé par l'INRIA Rocquencourt (Institut National de Recherche en Informatique et en Automatique, dans le thème Simulation et Optimisation de Systèmes Complexes (projet SOSSO) [62].

C'est un logiciel de recherche non encore commercialisé (pour la version 5) dont les binaires du cœur et les sources de l'interface graphique sont disponibles par ftp à partir de son site web.

SynDEx est un logiciel de CAO niveau système, supportant la méthodologie «Adéquation Algorithme Architecture» (AAA), pour le prototypage rapide et pour optimiser l'implantation d'applications temps réel embarquées sur des architectures multi-composants. C'est un logiciel graphique interactif qui offre les services suivants :

- spécification et vérification d'un algorithme d'application saisi sous la forme d'un graphe flot de données conditionné, ou interface avec des langages de haut niveau orientés simulation, contrôle-commande, traitement du signal et des images ;
- spécification d'un graphe d'architecture multi-composants (processeurs et/ou composants spécialisés) ;
- heuristique pour la distribution (avec contraintes de placement) et l'ordonnancement de l'algorithme d'application sur le multi-composants, avec optimisation du temps de réponse et visualisation de la prédiction des performances temps réel pour comparaison aux contraintes temps réel et pour le dimensionnement du multi-composants ;
- génération des exécutifs distribués temps réel, sans interblocage et principalement statiques, avec mesure optionnelle des performances temps réel ; ceux-ci sont construits, avec un surcoût minimal, à partir d'un noyau d'exécutif dépendant du processeur cible ; actuellement, des noyaux sont fournis pour : SHARC, TMS320C40, i80386, MC68332, i80C196 et stations de travail Unix ou Linux ; des noyaux pour d'autres processeurs sont facilement portés à partir des noyaux existants.

Puisque les exécutifs sont générés automatiquement, leur codage et leur mise au point sont éliminés, réduisant de manière importante la durée du cycle de développement.

La méthodologie AAA et le logiciel SynDEx qui la supporte visent les applications complexes de contrôle-commande, comprenant du traitement du signal et des images, soumises à des contraintes temps réel fortes, embarquées sur des architectures matérielles multi-composants.

Ces applications sont fortement hétérogènes, autant au niveau des algorithmes, qui peuvent contenir des parties homogènes/régulières/répétitives (comme par exemple les parties de trai-

tement d'image), qu'au niveau des architectures, qui peuvent comprendre des composants de caractéristiques différentes, aussi bien des processeurs programmables (CISC, RISC, DSP, microcontrôleurs) que des composants spécifiques (FPGA reconfigurables, ASIC figés), interconnectés par des média de communication aussi bien point-à-point (FIFO, RS232, liens) que multi-point (CAN, PCI, bus).

La complexité de ces applications est due aux aspects temps réel, distribué, hétérogène, et surtout embarqué qui réclament une minimisation des ressources matérielles mises en oeuvre, et donc aussi une minimisation des surcoûts de l'exécutif, qui assure l'interface entre l'algorithme et l'architecture.

Le graphe de l'algorithme étant spécifié *à plat* (sans hiérarchie d'encapsulation), les conditions pratiques d'édition limitent sa taille (sous forme factorisée) à quelques dizaines de sommets. Aussi une décomposition simplement fonctionnelle de l'algorithme n'est pas toujours la meilleure et peut nécessiter quelques adaptations (décomposition plus fine de certaines opérations trop grosses, ou au contraire regroupement de certaines opérations trop petites). Deux axes de recherche et de développement tentent d'apporter une solution à ce problème : la défactorisation efficace des motifs répétitifs factorisés, et la hiérarchisation du graphe de l'algorithme, qui permettra une spécification à plusieurs niveaux de granularité et un choix automatique du niveau de granularité le mieux adapté à l'adéquation.

Implanter efficacement des algorithmes complexes, de contrôle-commande et de traitement du signal et des images, sur des architectures multi-composants hétérogènes, est un problème complexe en soi. SynDEx permet d'une part d'explorer rapidement l'espace des implantations à la recherche d'une solution efficace, et d'autre part d'éviter l'écriture et la mise au point manuelles du code de l'implantation distribuée sur une architecture multi-composants, ce qui représente un gain de temps considérable par rapport aux méthodes manuelles d'optimisation et d'implantation.

1.3 Trapper

Trapper est un produit commercial dont le nom signifie **TRA**ffonic[1] **P**arallel **P**rogramming **E**nvi**R**onment [61].

C'est un environnement de développement graphique pour les systèmes embarqués de type MIMD, initialement développé pour les applications automobiles. Il est basé sur le modèle de programmation des processus séquentiels communicants et permet la conception, le placement, la visualisation et l'optimisation de systèmes parallèles. Le principe retenu dans Trapper est de laisser au programmeur le soin de spécifier explicitement la structure parallèle de son programme sous forme d'un graphe de processus et de l'aider dans les différentes phases de développement du système embarqué. Un système embarqué typique est constitué de capteurs, d'unités de calcul et de périphériques de sortie.

Trapper est issu d'une collaboration entre le laboratoire de recherche Daimler-Benz et le Centre Allemand National de Recherche (GMD). La première version a été conçue en 1991 et ne supportait que les architectures à base de Transputers. En 1999, Trapper supporte aussi des architectures cibles à base de PowerPC (par le biais des outils Parsytec PowerTools) ainsi que les réseaux de stations de travail en utilisant PVM et MPI.

Cet outil génère une distribution statique et un ordonnancement dynamique. La distri-

[1] TRAFFONIC est un programme de recherche Daimler-Benz sur l'utilisation de l'électronique dans les véhicules.

bution des communications peut être statique. C'est surtout un outil de mise au point et d'analyse de performances, offrant de nombreuses fonctionnalités d'affichage graphique pour présenter les comportements logiciel et matériel du système. Par contre rien n'indique que cet outil permette de garantir une exécution respectant des contraintes temps réel.

1.4 Gedae

GEDAE (**G**raphical **E**ntry, **D**istributed **A**pplications **E**nvironment) est un produit commercial issu des laboratoires de recherches de Lockheed Martin [49].

GEDAE est un logiciel graphique d'aide au développement d'applications de traitement du signal (systèmes temps réel) sur architectures multi-processeurs embarquées. Il permet à l'utilisateur :

- la modélisation avec formalisme du flot de données,
- la définition du placement et du parallélisme,
- la génération de code automatique,
- la simulation sur station de travail,
- le contrôle de l'exécution.

Son domaine d'application couvre le Traitement de Signal (TS) sans restriction apparente.

GEDAE est un outil interne de Lockheed Martin ATL développé depuis 10 ans par les ingénieurs logiciels en charge des calculateurs de TS. GEDAE a été utilisé pour supporter le développement des applications radar et sonar.

Un budget d'environ 50 millions de dollars a été attribué par le gouvernement américain aux équipes de Lockheed Martin et de Lockheed Martin ATL dans le cadre du programme de recherche américain RASSP. L'équipe RASSP de Lockheed Martin ATL utilise GEDAE pour le co-design et le prototypage rapide des benchmarks du programme. Le financement de RASSP à GEDAE a duré moins d'un an et n'a entraîné aucun changement dans l'évolution de GEDAE. Toutefois suite à son utilisation dans le programme RASSP, Lockheed Martin ATL a décidé de le commercialiser afin d'en assurer sa pérennité et son évolution.

GEDAE, de par sa méthodologie co-design, permet :

- de définir une application de TS à partir d'un formalisme flot de données,
- de choisir l'architecture de la machine cible,
- de générer du code pour le prototypage rapide sur des cartes COTS Mercury, Alex, Ixthos à base de processeurs SHARC ou de i860, mais aussi sur les propres machines de TS de Lockheed Martin (e.g . HPCN).

GEDAE contient des fonctionnalités pour :

- décrire une application de TS hiérarchiquement à partir d'une bibliothèque de fonctions usuelles,
- développer des fonctions utilisateur,
- placer les traitements sur une architecture parallèle,
- décrire les transferts de données en tenant compte du parallélisme des traitements,
- générer automatiquement du code C,
- visualiser la place mémoire occupée par chaque traitement,
- visualiser le temps passé à exécuter chaque traitement.

Ces informations sont disponibles à partir :

- d'une simulation sur machine hôte,
- du fonctionnement en temps réel sur machine cible.

En ce qui concerne la génération de code, GEDAE utilise :

- un ordonnanceur statique (le programme C généré à partir du modèle de flot de données est un programme séquentiel),
- un ordonnanceur dynamique (des queues dynamiques sont insérées entre les différents programmes C pour gérer les communications inter-processeurs ou intra-processeurs).

GEDAE est considéré début 1999 comme un outil en cours de développement manquant de maturité. Le manuel utilisateur est incomplet.

Il semble que la génération de code C d'un point de vue performance soit un des points forts de GEDAE. Il est un des rares outils commerciaux qui permettent de décrire le parallélisme de données pour la génération de code.

GEDAE est un outil adapté au prototypage rapide des applications TS sur machines COTS et mixtes. Cependant, il manque de maturité et sa pérennité commerciale peut poser problème pour des systèmes à durée de vie assez longues.

1.5 Ptolemy Classic

Ptolemy est un outil de recherche développé par le *Department of Electrical Engineering and Computer Sciences* de l'université de Californie à Berkeley, sous la direction de Edward A. Lee. Ses sources sont disponibles sur son site web [63].

Il vise la conception et la simulation, hétérogènes et concurrentes, de systèmes embarqués réactifs. Son but est de supporter la construction et l'interopérabilité de *modèles* exécutables selon des *modèles d'exécution* («domaines») très variés.

Ptolemy est un environnement destiné à modéliser des systèmes hétérogènes, et propose donc plusieurs outils parfaitement intégrés (domaines). La version 0.7 propose ainsi en plus du domaine «Static Data Flow», et ses dérivés, un domaine «Discrete-Events» pour la modélisation des communications, un domaine «Synchronous/Reactive» pour la modélisation des systèmes temps réel, et un domaine «Process network» pour modéliser les applications multi-threads, et des domaines de synthèse de codes pour des cibles DSP (Motorola, Texas Instruments). Ptolemy est autant un environnement de modélisation qu'un laboratoire de recherches pour déterminer de futurs méthodes et outils pour la modélisation de systèmes hétérogènes.

Ptolemy est un environnement graphique et textuel de spécification, conception et implémentation de systèmes complexes hétérogènes. Les premières versions de Ptolemy ont été orientées vers les systèmes de traitement du signal à base de DSP, essentiellement d'origine Motorola (DSP 5600, 9600) et Texas Intrument (C50, C60). Certaines versions de Ptolemy (0.4 et 0.5) ont inclu un générateur de code pour le langage Silage de l'UC Berkeley. Il est ensuite possible à partir de Silage de générer du code C ou VHDL afin d'utiliser soit des DSPs ou des ASICs/FPGAs. Des outils comme Xpole ou Xgraph sont interfacés à Ptolemy et permettent respectivement de concevoir de façon graphique et interactive des filtres, et d'afficher des courbes de façon particulièrement efficace. De plus, les outils Matlab et Mathematica sont interfacés et permettent de réutiliser, si nécessaire, un existant parfois important. Ptolemy est aussi un «framework orienté objet». Cette notion s'est rapidement diffusée avec le langage Java, qui est un langage accompagné de nombreux «frameworks» dédiés à différents domaines : Interfaces graphiques, bases de données, multimédia, communications, etc. En particulier, Ptolemy propose une infrastructure réutilisable de générateurs de codes pouvant être adaptée à des besoins spécifiques, typiquement pour générer du code pour une architecture donnée de calculateurs. Cette caractéristique fait de Ptolemy un outil unique.

Chaque domaine dispose d'une bibliothèque de composants. Les domaines dataflow et événementiel sont les mieux pourvus. Cependant, il est possible d'ajouter rapidement des blocs supplémentaires adaptés aux besoins de chaque projet, ou permettant de réutiliser un existant souvent important lorsqu'il s'agit par exemple du traitement du signal.

Par nature Ptolemy est ouvert. Le code source est disponible, et son architecture a déjà permis d'intégrer plusieurs domaines ensembles (dataflow, événementiel, réactif par exemple) et des outils externes comme Matlab, Mathematica ou encore Leapfrog de Cadence juste en raffinant des services existants dans le noyau de Ptolemy.

Le noyau de Ptolemy a été conçu à l'origine pour faire cohabiter des modèles d'exécution différents. Ces modèles d'exécution implémentés sous la forme de domaines dans Ptolemy sont donc bien intégrés et permettent de modéliser sans aucun problème des systèmes avec une approche multi-formalismes. De plus, les domaines de génération de codes permettent par exemple de générer du code VHDL, ou d'assembler du code existant, et de le faire exécuter par des simulateurs externes à Ptolemy comme Leapfrog de Cadence. Le noyau de Ptolemy propose des services de base permettant de gérer l'interface avec ces simulateurs externes et de rendre ainsi transparent pour l'utilisateur leur utilisation.

Ptolemy est particulièrement adapté au développement d'applications de traitement du signal avec ses modèles DataFlow et à la simulation de cartes numériques ou de protocoles de communication avec son modèle événementiel. De plus, sa capacité d'utiliser plusieurs modèles dans une même simulation permet de prendre en compte conjointement plusieurs aspects d'un système embarqué et de faciliter les choix de conception avant toute implémentation, de part la possibilité d'observer les interactions entre matériels et logiciels avant développement et intégration.

1.6 Casch

CASCH est l'acronyme de **C**omputer **A**ided **SCH**eduling. Il est développé au Département Informatique de l'Université des Sciences et Technologies de Hong-Kong (Clear Water Bay) [46]. CASCH ne fonctionne que sur station de travail SUN.

CASCH est un outil graphique dont le but est d'automatiser l'extraction de parallélisme d'algorithmes (traitement du signal, vision) spécifiés textuellement et séquentiellement pour les exécuter sur des machines parallèles. Il génère automatiquement :

- un graphe acyclique direct,
- un partitionnement (mapping) et un ordonnancement,
- des primitives de communication et de synchronisation.

Cet outil propose une grande bibliothèque extensible d'algorithmes de partitionnement et d'ordonnancement. L'utilisateur peut ainsi rechercher parmi les algorithmes implantés, celui qui génère l'ordonnancement qui utilise au mieux les ressources. Chaque algorithme peut être interactivement analysé, testé et comparé en utilisant des données fournies par un simulateur. Pour faciliter ce travail de comparaison d'algorithmes de partitionnement et d'ordonnancement, CASCH possède aussi un générateur automatique et aléatoire de graphes.

Cet outil ne cible pas les domaines de l'embarqué et du temps réel. C'est un outil d'aide à l'équilibrage de charge statique de réseau de stations ou de machines multi-processeurs homogènes dans le contexte d'applications de traitement du signal et des images. C'est essentiellement un outil qui permet l'évaluation et la comparaison d'algorithmes de partitionnement et d'ordonnancement de graphes flots de données.

1.7 Méthodologie PLC^2, Apotres

APOTRES est l'acronyme d'**A**ide au **P**lacement **O**ptimisé de **T**raitement de signal **R**adar **E**t **S**onar. C'est l'outil qui supporte la méthodologie PLC^2 tout deux développés à Thales Research & Technology en étroite collaboration avec le Centre de Recherche en Informatique de l'ÉCOLE DES MINES DE PARIS [32][3].

APOTRES est un démonstrateur d'aide au placement d'applications TSS sur architectures parallèles homogènes. À partir des descriptions fonctionnelles de l'application et de l'architecture, l'outil produit des directives de placement. Il permet :

- la génération de solutions multiples ;
- la prise en compte des ressources de la machine ;
- la vérification du respect des contraintes temps réel et architecturales ;
- la visualisation du séquencement.

Il propose de résoudre le problème du *placement automatique* dans le cadre du traitement de signal homogène (application TSS et architecture SIMD/SPMD) déterministe et structuré (ex : compression d'impulsion, transformée de Fourier).

La résolution du problème du «placement» utilisant l'approche multi-modèles PLC^2 est le cœur des études menées depuis 1996, au Laboratoire Central de Recherches. Ainsi il permet d'appréhender globalement le problème du placement.

L'outil offre de choisir un critère d'optimisation sans changer les différents modèles inhérents à la fonction placement. Parmi ces critères, on trouve :

La latence : minimisation du temps d'exécution de l'application placée ;

La capacité mémoire : minimisation de la capacité mémoire consommée par l'application placée ;

L'efficacité : maximisation de l'utilisation des processeurs ;

Le coût : En intégrant un coût (financier par exemple) sur chacun des composants matériels (processeur, mémoire, etc), l'outil permet de trouver le (ou les) placement(s) de l'application qui minimise(nt) ce coût.

Cette approche s'inscrit dans un contexte de *co-design* et de *prototypage virtuel*. Il offre à l'utilisateur la possibilité

de saisir manuellement une solution partielle : l'outil poursuit la recherche en complétant la solution ;

de visualiser des solutions complexes : comme par exemple le séquencement, le découpage des données ou l'affectation, etc.

de dimensionner la machine : l'outil permet de spécifier les ressources nécessaires pour placer une application particulière sur un type de machine donnée sans violer les contraintes applicatives. Cela consiste à prendre en compte les dimensions et les caractéristiques de chaque composant matériel.

de choisir le critère d'optimisation du placement.

APOTRES ne fournit que des directives de placement. Il ne génère pas le code correspondant. Il ne permet ni la spécification de l'application à placer, ni la simulation comportementale de la solution proposée. Le domaine d'application est assez étroit dans la mesure où il ne s'intéresse qu'aux applications TSS et aux machines parallèles homogènes. C'est plutôt un logiciel fermé puisqu'il n'offre pas la possibilité de se brancher à des outils existants.

L'interface utilisateur est assez intuitive. Les différents paramètres d'entrée - concernant l'application à placer, l'architecture cible et les options de résolution (telles que le critère d'optimisation) - sont classés dans des onglets. L'utilisation du logiciel se fait par quelques clics de souris. Accompagnant les interprétations graphiques des résultats, l'utilisateur a la possibilité d'obtenir un rapport d'exécution (rédigé en LaTeX ou HTML) résumant les conditions initiales (description de l'architecture cible, critère d'optimisation et autres options pre ou post-résolution) et détaillant la solution proposée.

1.8 Conclusion

Cette section a présenté quelques outils ainsi que le contexte dans lequel chacun d'eux évolue. Ils sont issus d'horizons différents mais sont tous liés d'une façon ou d'autre à une phase de développement d'applications TSS sur machines parallèles (prototypage rapide, simulation, placement,...). Nous allons maintenant comparer ces outils selon divers aspects, parmi lesquels :

1. l'algorithme à placer ;
2. les machines et les modèles de programmation cibles ;
3. les méthodes de résolution et leurs implications sur les algorithmes et les machines ;
4. les ensembles de solutions pour lesquels on cherche un optimum ;
5. la qualité des estimations temporelles effectuées.

2 Analyse

2.1 Classification des algorithmes

Tous les outils ne résolvent pas le problème du placement pour tous les algorithmes de traitement du signal. Ils sont plus ou moins spécialisés. Les algorithmes à placer étant des algorithmes de traitements du signal, ils sont soumis à certaines contraintes de latence qui doivent être respectées une fois placés sur les machines cibles. Il est aussi possible de les comparer selon des propriétés syntaxiques des algorithmes, ainsi que sur la génération de code (pour ceux le permettant).

2.1.1 Garantie sur la latence

La latence correspond au temps physique écoulé entre l'acquisition d'une donnée et le dernier résultat auquel elle a contribué dans une chaîne de calculs. La durée des communications non recouvertes par des calculs est prise en compte dans la latence.

Les calculs sont des occurrences de fonctions (ou tâches élémentaires) reliées par les arcs d'un Graphe de Flot de Données. Ce graphe permet de suivre les valeurs intermédiaires reliant une entrée à une sortie.

Comme il faut pouvoir garantir une ou plusieurs latences en temps physique, les questions qui se posent naturellement à partir de cette définition sont :

1. Qu'est-ce qu'une tâche élémentaire (dans le dialecte de l'outil) ? Ses arguments doivent-ils tous être des scalaires ou peut-on utiliser des tableaux ? En d'autres mots, peut-on prendre en compte des sous-programmes ou seulement des opérations ? Ou au contraire faut-il impérativement que les tâches élémentaires aient des temps d'exécution longs pour absorber des surcoûts dus à la technique de génération de code ?
2. Est-il possible de connaître exactement le temps d'exécution des tâches élémentaires ? Dans la négative, est-il possible de le maximiser ?
3. Le graphe de flots de données est-il statique ou dépendant des données ? Est-il calculable et utilisable par des algorithmes de placement automatique ?
4. Est-ce que le nombre des tâches impactant une sortie est constant dans le temps ou non ? Est-il bornable ? Dans la négative, la latence calculée est au mieux une latence dans le cas le pire.

Ces quatre questions définissent quatre critères permettant de classer différentes applications et différents outils les uns par rapport aux autres.

Remarque : L'algorithme mis sous forme de graphe de flot de données est un objet infini, dépendant potentiellement d'entrées inconnues a priori. Il est impossible dans le cas général de garantir une latence.

Les techniques de résolution peuvent imposer différentes contraintes au graphe de flot de données ou à certains de ses graphes quotients[2] : absence de *fork*, de *join* ou de circuits.

La parallélisation automatique traite tous les cas possibles d'algorithmes en utilisant des approximations supérieures sur les prédécesseurs des calculs quand le graphe de

[2]Un graphe quotient est un graphe replié selon une relation d'équivalence donnée.

flot de données n'est pas statique ou n'est pas calculable ; mais ceci n'est pas possible pour la dimension infinie du temps.

2.1.1.1 Qu'est-ce qu'une tâche élémentaire ? Le «grain» d'une tâche élémentaire diffère d'un outil à l'autre.

Par exemple, les applications de Traitement de Signal Systématique prises en compte par l'outil APOTRES sont formées de déclarations de *tâches* que l'on peut exprimer par des nids de boucles bien structurés et parallèles. Chaque nid de boucles contient un appel à une *Tâche* (ou *Transformation*) *Élémentaire* (TE) correspondant en général à une fonction d'une librairie de traitement du signal telle qu'une FFT. C'est pourquoi une TE est considérée comme une boîte noire (issue d'une bibliothèque de transformations) dont les interfaces avec l'extérieur sont des tableaux (finis) de données d'entrées/sorties de la TE. Chaque TE lit un ou plusieurs tableaux de données multidimensionnels et range le résultat dans un unique tableau distinct des tableaux d'entrées.

Les tâches élémentaires des applications prises en compte par des outils tels que FX, CASCH, sont des procédures (au sens Fortran du terme). En effet, les applications visées par CASCH sont spécifiées sous forme textuelle séquentielle (type Fortran). C'est une séquence de boucles imbriquées avec appels de procédures.

La tâche élémentaire pour TRAPPER est un processus caractérisé par des propriétés (attributs) héritées ou communes avec d'autres processus ainsi que des propriétés propres : chaque processus possède un identificateur unique et un champ pour le type du processus. C'est par ce dernier que l'on associe le code séquentiel à un processus (plusieurs processus peuvent donc être de même type et ainsi faire appel à un même code). TRAPPER supporte ainsi implicitement l'approche SPMD. D'autres propriétés sont stockées au moyen du champ type de processus, par exemple la taille du tas pour ce code, la valeur d'attribut autorisant le traçage ou non, etc.

Pour SynDEx, la tâche élémentaire est définie comme étant *l'opération atomique*, c'est-à-dire l'opération de granularité la plus fine. Le choix de cette granularité est du ressort de l'utilisateur. Ce choix n'est pas sans conséquences sur les performances des implantations qu'il permet. Des grains trop gros permettent rarement une distribution équilibrée de la charge (de calcul entre processeurs et/ou de communication entre média), conduisant à une efficacité médiocre. Cependant, comme le surcoût de durée d'exécution dû à l'encapsulation de chaque grain dépend peu de la taille du grain, des grains très petits en grand nombre peuvent entraîner un surcoût total important.

La tâche élémentaire pour GEDAE est *la boîte fonctionnelle primitive*. Il y a deux classes de boîtes primitives : une dite *hôte* et l'autre *embarquable*. Les premières sont conçues à partir d'un modèle objet distribué ; elles sont utilisables dans un environnement de stations de travail mais ne peuvent être distribuées sur un système embarqué. Quant aux primitives embarquables, elles sont assimilables à des processus légers ; elles sont aussi utilisables sur une station de travail multi-processeurs. Ces boîtes primitives sont issues de bibliothèques ou créées par l'utilisateur.

2.1.1.2 Nature du graphe de flots Dans le graphe de précédence, un programme est représenté par un graphe orienté. Dans ce graphe, les sommets sont l'ensemble des tâches

élémentaires et les arcs, les contraintes de précédence. Chaque sommet peut être valué[3] par le temps d'exécution de la tâche représentée, et chaque arc peut être valué par un coût de communication représentant par exemple la quantité d'informations à échanger entre les tâches. La largeur du graphe représente le parallélisme potentiel sous-jacent à l'application. En d'autres termes, si on n'éclate pas les nœuds de ce graphe, celle-ci est égale au nombre maximal de processus que l'on peut placer en parallèle. Si on ignore les relations de précédence entre les tâches, l'arc non orienté représentera uniquement un lien de communication, et le graphe résultant est appelé graphe de tâches.

L'application traitée par APOTRES est sous forme d'*assignation unique*, c'est-à-dire que chaque élément de tableau n'est mis qu'une seule fois à jour par l'application. Cette dernière est globalement représentée par un *graphe de flot de données sans circuit*. Dans le graphe de flot de données, un nœud représente une fonction de traitement du signal et une arête un flux de données transitant entre deux nœuds. Le calcul des différentes données permettant de dimensionner l'application est résolu statiquement. Un nœud (TE) accepte en entrée et en sortie un ou plusieurs flux.

Pour CASCH, chaque nœud du graphe de l'algorithme possède un poids qui correspond à la durée d'exécution de la procédure qu'il représente. Chaque arc du graphe possède aussi un poids qui correspond à la durée de transmission du message qu'il représente.

Le graphe de flots dans GEDAE est paramétré, conditionnel, acyclique. La notion de retard est modélisable à l'aide des primitives cycliques.

Pour SynDEx, le graphe d'algorithme est un *graphe factorisé de dépendances de données entre opérations conditionnées*, où chaque sommet appelé «opération» est :

- soit un calcul fini et sans effet de bord (les résultats du calcul en sortie de l'opération ne dépendent que des données en entrée de l'opération : pas d'état interne mémorisé entre exécutions de l'opération, ni d'accès à des signaux externes de capteurs ou d'actionneurs),
- soit un délimiteur frontière de factorisation marquant une dépendance de données entre des sommets appartenant à deux motifs ayant des facteurs de répétition différents, l'un «rapide» multiple de l'autre «lent» et où chaque hyper-arc (c'est-à-dire pouvant relier plus de deux sommets) est une dépendance de donnée entre une (et une seule) sortie d'une opération émettrice en amont et une (des) entrée(s) d'une (de plusieurs) opération(s) réceptrice(s) en aval ; noter qu'il n'y a pas de notion de «variable» au sens des langages impératifs qui spécifient un ordre total d'exécution sur les opérations d'où sont extraites pour chaque variable des dépendances écriture(w)/lecture(r), w/w, r/w et r/r. La spécification «flot de données» est déclarative et les dépendances de données déterminent un ordre partiel d'exécution ; il ne peut y avoir de cycle dans le GFD qu'à travers un retard.

2.1.1.3 Temps d'exécution des tâches élémentaires et des communications

Le temps d'exécution ou de communication est en général soit simulé, soit estimé.

Pour l'outil APOTRES, les TEs ont un coût d'exécution fixé inclus dans la spécification de l'application. Ce coût est mesuré en cycles machine. La durée des communications est estimé. Elle est fonction du volume à communiquer et de la bande passante.

[3] Valuer dans le sens affecter une valeur.

Dans CASCH, chaque nœud du graphe de flots de l'algorithme possède un poids qui correspond à la durée d'exécution de la procédure qu'il représente. Chaque arc du graphe possède aussi un poids qui correspond à la durée de transmission du message qu'il représente. L'estimation des durées de communication (obtenue expérimentalement) est basée sur le coût de chaque primitive de communication (*send, receive, broadcast*). Elle est estimée en utilisant le temps de *startup*, la longueur du message et la bande passante du canal de communication.

L'attribut *time* dans TRAPPER permet de spécifier une estimation de la charge processeur qu'occasionnera la tâche élémentaire (ici le processus).

La durée d'exécution de toute opération (TE) dans SynDEx est bornée (pas de boucles ou récursions potentiellement infinies, et pour un capteur ou un actionneur, pas d'attente indéfinie d'un signal externe de synchronisation).

2.1.2 Caractéristiques syntaxiques

Les caractéristiques syntaxiques des langages de spécification d'algorithmes ont un impact sur les quatre critères de la page 30 :

Impact sur le critère 1 Peut-on seulement désigner une valeur (ex : **Alpha**) ou bien peut-on manipuler tout un intervalle de valeurs d'indice (**Fortran**/APOTRES) ? Peut-on utiliser une bibliothèque de sous-programmes ? Peut-on éventuellement hiérarchiser la définition ?

Exemple : encapsulation d'une FFT suivie d'une exploitation parallèle dans la dimension des fréquences. SynDEx le permet mais à travers des boucles d'adaptation dont on ne sait pas comment elles sont prises en compte au niveau placement/ordonnancement. GEDAE semble le permettre au moins au niveau syntaxique.

Impact sur le critère 2 Y a-t-il des restrictions sur les fonctions utilisables dans les expressions d'indices ? Ceci est lié à la calculabilité du graphe de flot de données et à l'assignation unique. Les restrictions peuvent être différentes dans les membres droits et gauches pour assurer l'assignation unique :

1. indices de boucles $A(I) = f(B(I))$
2. indices de boucles plus un décalage, i.e. un retard (SynDEx pour la dimension infinie) $A(I) = f(B(I) + B(I-1))$
3. expressions affines des indices de boucles, intervalles avec ou sans recouvrement $A(I) = f(B(2*I\ : 2*I+1))$
4. expressions affines des indices de boucles avec pas $A(I) = f(B(2{*}I\ : 2{*}I{+}10\ : 2))$
5. expressions pseudo-affines (modulo et division entière par une constante)
6. n'importe quelle expression fonctionnelle utilisant les indices de boucles comme argument.
7. n'importe quelle expression fonctionnelle utilisant n'importe quelle valeur calculée par l'algorithme *Exemple :* $A(i) = B(X(i))$, *quel que soit* X.
   Exemple : $B(X(i)) = A(i)$, *si* X *est une permutation assurant l'assignation unique.*

Impact sur les critères 2 et 3 Présence de conditionnelles :

1. dans une tâche élémentaire : la durée n'est généralement plus constante (critère 2) ;
2. dans le flot de contrôle ;

3. par rapport à un signal de contrôle permettant l'exécution d'une tâche A ou une tâche B : la durée n'est généralement plus constante (critère 2) que le contrôle soit d'origine commande/paramètre ou calculé ;
4. par rapport à une valeur signal permettant l'exécution d'une tâche A uniquement si le signal est vrai : la durée n'est plus constante (critère 2) et la graphe de flot de contrôle n'est plus calculable (critère 3) parce que l'ensemble des tâches à effectuer n'est plus connu statiquement.

Impact sur le critère 4 Paramétrage dynamique de l'algorithme :

1. paramètre de traitement modifiant la nature et donc la durée d'une tâche élémentaire ;
2. paramètre de traitement modifiant le chemin de données (ou nécessitant l'introduction de tâches *identité* pouvant conduire à des copies de données inutiles) ; *Exemple : si l'antenne radar est pointée vers le haut, on ne prend pas en compte l'effet de sol.*
3. paramètre de traitement modifiant le nombre de tâches à exécuter pour obtenir un résultat ; par conséquent, le graphe de flot de données, probablement la taille de l'entrée et donc la durée d'une tâche en aval.

Remarque : La IF-conversion, i.e. le remplacement des conditionnelles par des gardes, revient à générer des accès indirects quand il faut exprimer les accès aux valeurs par les seules fonctions d'accès.

Exemple : génération d'une alarme quand une valeur dépasse un seuil et traitement de l'alarme.

La formule *Algorithme = Données + Contrôle* rappelle qu'on ne peut pas s'en sortir en général en ne traitant que l'aspect donnée ou que l'aspect contrôle.

Les restrictions syntaxiques ne sont pas les mêmes pour les membres gauches et les membres droits dans la mesure où l'on veut garantir l'assignation unique.

Les restrictions ne sont pas toujours les mêmes pour les dimensions infinies et pour les dimensions finies, les techniques de résolution pouvant être différentes.

Les possibilités de calcul en place ne sont pas bien exprimées au niveau fonctionnel auquel on se place ici ; il faudrait que le mécanisme d'allocation mémoire en prévoit la possibilité.

Un système traitant réellement des expressions d'indices affines est *multirate*.

Le paramétrage de la spécification n'implique pas que le programme produit soit aussi paramétrique : GEDAE instancie les paramètres de la spécification avant de dériver le code exécutable.

2.1.2.1 Expression d'indices

Dans APOTRES, il s'agit d'expressions pseudo-linéaires (présence d'un modulo). De plus, APOTRES est multirate. SynDEx est monorate. Il ne traite donc pas les expressions affines dans leur généralité ; avec SynDEx, le passage du multirate au monorate est à la charge de l'utilisateur qui doit ajouter à la spécification des boucles sur le temps assurant cette conversion.

GEDAE : les expressions d'indices sont placées dans des *boites de routage*. Il n'y a aucune restriction sur les fonctions d'indices. Il n'y a pas de différences entre membre droit (fonction F) et membre gauche (fonction G) ; il faudrait donc pouvoir inverser formellement l'une des deux pour placer l'arc correspondant (les arcs correspondants) dans le graphe de flots de données.

2.1.2.2 Retard et conditionnement Les outils APOTRES, CASCH, TRAPPER n'admettent pas de notion de retard explicite, ni de notion de conditionnement.

Le graphe de l'algorithme dans GEDAE est paramétré, conditionnel, acyclique. La notion de retard est modélisable à l'aide des primitives cycliques.

Le retard, dans SynDEx, marque une dépendance de donnée inter-itérations (effet mémoire, ou état, entre itérations d'un motif répétitif), avec une seconde entrée «initiale» pour la première itération, et (pour les répétitions finies) une seconde sortie «finale» pour la dernière itération.

2.1.2.3 Contraintes sur la génération de code GEDAE classe les algorithmes en fonction des techniques utilisées dans son exécutif pour contrôler l'exécution :

1. consommation et production connues ;
2. consommation et production bornées : *on ne peut plus calculer la latence mais juste une borne sur la latence* ;
3. consommation et production inconnues : *on ne peut rien garantir statiquement.*

2.2 Machine cible, contrôle généré et exécutif

Dans tous les cas, il faut réduire le graphe de flot de données infini à un code fini. Cette réduction peut être vue mathématiquement comme un quotientage de graphe faisant passer d'une famille infinie d'instances de tâches élémentaires à une instruction, à plusieurs instructions ou à un nid de boucles suivant la richesse de la famille de relations d'équivalence utilisée par l'outil. Cette réduction est étudiée dans la section suivante : "Méthode de résolution du problème". On s'intéresse ici à la nature de l'objet fini qu'il faut produire.

Les différents outils ne donnent pas d'information précises sur la nature du code généré. APOTRES ne comprend pas encore de phase de génération de code. GEDAE considère que c'est confidentiel : il faut aller regarder le code généré pour comprendre et différents modes sont utilisés en fonction de la catégorie d'applications et en fonction du placement. SynDEx s'appuie sur un petit exécutif et utilise des modes différents suivant les placements respectifs des producteurs et des consommateurs.

2.2.1 Machine cible

L'ensemble des architectures cibles n'est borné que par l'imagination des concepteurs.

Au-delà des problèmes de topologie et de parallélisme entre communications et calculs, il ne reste plus qu'un modèle de programmation.

Les modèles utilisés sont encore a peu près ceux de la classification de Flynn :

SIMD : contrôle centralisé par boucles ; machine cible homogène,

SPMD : contrôle centralisé par boucles ou contrôles locaux par boucles plus synchronisation inter-processeur par échanges de messages ; machine cible a priori homogène,

MIMD : contrôles locaux par boucles plus synchronisation inter-processeur par échanges de messages ; machine cible a priori non homogène.

Le code SIMD peut être transformé en code SPMD qui peut être transformé en code MIMD efficacement. L'inverse est aussi possible mais conduit à du code inefficace.

Le modèle de programmation de l'outil APOTRES est de type SPMD . En fait, c'est une alternance de phases de parallélisme de données et de parallélisme de tâches. Celui de CASCH, GEDAE, SynDEx est MIMD . TRAPPER accepte les deux modèles de programmation SPMD ou MIMD.

2.2.2 Contrôle

On reconnaît trois familles de contrôles :

1. **Entièrement compilé**
   - utilisation de séquences (SynDEx sur un processeur)
   - utilisation de boucles
   - utilisation d'un automate
   - combinaisons des techniques précédentes (boucles et séquences pour APOTRES)
2. **Interprété** (machine à cadencement par les données)
   - conservation des données dans des files d'attente en entrée des tâches (duplication si plusieurs consommateurs)
   - conservation de pointeurs vers les données dans des files d'attente en entrée des tâches
   - comptage du nombre d'entrées disponibles pour activation sur événement
   - utilisation d'un bit de présence des données
   - pooling sur les entrées
   - priorité implicite ou explicite quand deux tâches peuvent être activées sur un unique processeur (problème pour l'optimisation de la mémoire et de la latence)
3. **Mixte compilé/interprété** (SynDEx, GEDAE)

On peut aussi faire du contrôle local par échange de messages (i.e. par files d'attente) mais le surcoût est considérable par rapport à l'utilisation d'une structure de contrôle comme la boucle. Le contrôle par files d'attente ne suppose pas que les producteurs soient identifiés par les consommateurs.

La compilation du contrôle suppose que les temps de calcul et de communication soient bien connus.

L'interprétation du contrôle doit pouvoir toujours fonctionner. Elle peut conduire à des blocages si les files d'attente ne sont pas assez profondes ou si les spécifications sont erronées (consommation insuffisante, trop importante ou désynchronisée entre deux variables) ou si les priorités sont mal choisies. Son surcoût à l'exécution ne permet pas de l'utiliser pour des tâches très fines comme une simple addition.

L'estimation des durées de communication dans CASCH (obtenue expérimentalement) est basée sur le coût de chaque primitive de communication (*send, receive, broadcast*). Elle est estimée en utilisant le temps de *startup*, la longueur du message et la bande passante du canal de communication. Les poids des nœuds et des arcs sont obtenus par un module de CASCH (*estimator*) qui est spécifique à chaque architecture cible.

2.2.3 Allocation mémoire

La technique d'allocation mémoire est liée à la technique de contrôle. L'allocation mémoire est généralement faite signal par signal plutôt que globalement.

L'utilisation de buffers circulaires introduit un surcoût qui n'est pas forcément acceptable pour de très petites tâches élémentaires. Si le contrôle est effectué avec des boucles, un déroulage partiel ou une technique de pipeline logiciel devraient permettre d'éliminer la gestion des accès circulaires.

2.2.4 Hiérarchie mémoire

Le problème de la gestion d'une hiérarchie mémoire n'a pas été encore abordé par les outils. Le temps réel et les temps d'exécution variables dus aux caches ne font d'ailleurs pas bon ménage. La présence des caches conduit aussi à renoncer à connaître exactement les temps d'exécution. La borne supérieure est cependant trop mauvaise par rapport au cas moyen pour être acceptable.

2.2.5 Entrée-Sortie

Les entrées-sorties peuvent être bufferisées pour rendre le contrôle relativement indépendant du temps réel et/ou pour réduire le surcoût du aux interruptions. Ceci augmente la latence exprimée en nombre de périodes et l'espace mémoire nécessaire tout en améliorant l'efficacité d'utilisation des processeurs. Cette dernière peut donc conduire à une diminution de la latence exprimée en secondes.

2.3 Méthodes de résolution du problème

Les méthodes de résolution diffèrent par la taille des problèmes qu'elles peuvent aborder (traitement en extension ou en compréhension), par la liste des décisions qu'elles peuvent prendre pour passer d'une spécification à un code, et par la liste des fonctions de coût qu'elles peuvent chercher à optimiser.

Elles se distinguent aussi par les espaces de solutions qu'elles explorent. Ceci est lié aux décisions qui peuvent être prises.

2.3.1 En extension ou en compréhension ?

Les méthodes de résolution diffèrent non seulement par la richesse du quotientage utilisé ou par les techniques d'optimisation mises en œuvre mais aussi par le moment auquel il est appliqué. Certains outils comme SynDEx ne travaillent que sur des ensembles définis en extension. Ils supposent donc qu'un premier quotientage du graphe de flot de données soit effectué manuellement par l'utilisateur pour fournir son entrée à l'outil. Des décisions irréversibles sont donc prises entre les spécifications et l'entrée de l'outil ; mais du coup, de simples techniques énumératives permettent d'arriver à un code exécutable optimal choisi dans un ensemble fini de solutions.

Par exemple, l'approche utilisée dans CASCH pour équilibrer la charge des processeurs et minimiser les communications consiste à partitionner le graphe de l'algorithme afin de regrouper les procédures en autant de tâches qu'il y a de processeurs. Chaque tâche est ensuite assignée à un processeur. C'est encore une approche en extension.

En général, les modèles utilisés nécessitent une connaissance de l'ensembles des tâches, de leurs relations de précédence et de leurs coûts de communication et/ou mémoire. On

suppose alors que ces valeurs ne se modifieront pas en cours d'exécution. On distingue alors deux cas :

1. Le placement est fait sur une architecture dont le réseau d'interconnexion est prédéfini[4]. La plupart des algorithmes existants traitent le cas où le nombre de processeurs et le réseau d'interconnexion sont fixés. La topologie est fixée mais le nombre de processeurs lui, n'est pas fixé et dépend du nombre de processeurs disponibles et/ou du compromis entre nombre de processeurs et gain obtenu (*speed-up*).
2. Le placement s'accompagne d'une réorganisation de l'architecture[5].

Le «**placement statique sur architecture statique optimale**» est basé, soit sur la théorie des graphes (comme dans SynDEx), soit sur des techniques de programmations mathématiques (comme la programmation linéaire en nombre entier ou la modélisation concurrente et la programmation par contraintes comme dans l'approche PLC^2).

Le placement basé sur la théorie des graphes modélise le problème par des graphes (logiciel pour l'application et matériel pour la machine cible) et utilise des techniques de traitements de graphes pour le résoudre (recherche d'homomorphisme, partitions de graphes, etc).

Le placement basé sur des techniques de programmations mathématiques repose sur une modélisation mathématique fine du problème. On se ramène en fait à un problème d'optimisation combinatoire, où l'on cherche à minimiser une fonction de coût tout en respectant les contraintes du problème.

Le «**placement statique sur architecture statique non optimale**» utilise la plupart du temps des stratégies basées sur l'utilisation d'heuristiques[6] qui permettent de réduire la dimension du problème mais ne fournissent que des solutions approchées. Citons dans ce contexte les algorithmes dits «gloutons» et les algorithmes itératifs types recuit simulé, ou encore les algorithmes génétiques.

D'autres outils comme APOTRES traitent les ensembles uniquement en compréhension ce qui ne leur permet pas d'aborder la phase de génération de code de manière efficace. Mais cette méthode donne un maximum de liberté dans la phase de prises de décision, et permet de traiter de très grands nombres de tâches et des machines ayant un grand nombre de processeurs, pour lesquelles les méthodes heuristiques énumératives sont trop lentes.

2.3.2 Ensemble des décisions prises

On peut voir cinq champs de prise de décision :

1. Adaptation/sélection de la granularité des tâches en fonction des surcoûts et de l'équilibrage de charge (APOTRES) ;
2. Placement des calculs sur les processeurs (APOTRES, SynDEx) ;
3. Allocation mémoire ;
4. Placement des communications sur des routes ou sur des dispositifs (SynDEx) ;
5. Aucune décision n'est prise par l'outil mais il laisse l'utilisateur choisir explicitement ou implicitement une solution particulière ; cette solution peut être exécutée ou simulée par l'outil et l'utilisateur peut alors réagir en modifiant sa solution.

[4]Cette hypothèse est faite dans APOTRES car on suppose l'architecture cible de type SIMD complètement connectée.

[5]Nous n'avons pas fait d'état de l'art sur ce type de placement.

[6]Les heuristiques sont des critères, des principes ou des méthodes permettant de déterminer parmi plusieurs stratégies, celle qui permet d'être la plus efficace pour atteindre un but.

GEDAE ne prend aucune décision et laisse l'utilisateur modifier lui-même les tâches et le graphe de flots de données jusqu'à ce que le code généré réponde à ses spécifications opérationnelles.

2.3.2.1 Gestion de la mémoire La structure principale du modèle mémoire dans APOTRES est un modèle capacitif qui dépend du nombre de blocs alloués sur la longueur du chemin critique, ainsi que de ses volumes de références en écriture et en lecture. L'allocation mémoire n'est pas un modèle actuellement pris en compte mais devrait se faire via un post-traitement.

Dans CASCH la capacité mémoire se fait en post-évaluation. Cependant cet outil permet d'afficher le découpage et la répartition des données sur les différents processeurs.

GEDAE combine les techniques de programmation séquentielles ou flot de données et flot de contrôle pour traiter spécifiquement l'utilisation mémoire. La capacité d'une boite fonctionnelle est calculée en sommant les produits des capacités des queues d'un type de données par la taille des queues de ce type de données. Dans le cas d'une application répartie (i.e. les ports d'E/S d'une fonction primitive sont sur des processeurs différents), les données sont dupliquées. Par conséquent, le nombre de copies dépend de la destination des données. Lorsque les ports sont sur le même processeur ou plus spécifiquement dans la même partition embarquable, les données peuvent être partagées. L'utilisation mémoire peut aussi être contrôlée par les développeurs de boîtes fonctionnelles. On peut ainsi écraser après calcul l'espace mémoire réservé pour l'entrée par la sortie.

Les variables systèmes et globales sont allouées identiquement pour tous les modules et les processeurs pour toute la durée de l'exécution. Les variables locales sont allouées à l'exécution dans la pile. Suffisamment de mémoire est alloué pour la communication et autres buffers à l'entrée dans la tâche et désalloué à la sortie.

Dans SynDEx, la capacité mémoire n'est pas prise en compte lors de du placement mais elle se fait en post-évaluation. L'allocation mémoire est actuellement effectuée lors de la génération de code.

La capacité mémoire nécessaire sur chaque processeur est estimée a priori dans l'outil TRAPPER . En revanche, l'allocation mémoire est totalement dynamique, et est prise en charge par le système d'exploitation sous-jacent.

2.3.3 Fonctions de coût (optimisation)

L'ensemble des fonctions de coût est plus ou moins grand :
- minimisation du nombre de processeurs ;
- minimisation de la mémoire ;
- minimisation de la latence ;
- maximisation de l'efficacité des processeurs ;
- simple garantie de latence (SynDEx).

2.4 Ensembles de solutions potentielles

Les ensembles de solutions possibles dépendent des techniques utilisées pour passer d'une spécification graphe flot de données à un code. Il faut réduire le graphe infini en compré-

hension à un graphe fini, adapter la granularité des tâches qui vont faire l'objet d'un ordonnancement, placer les calculs sur des processeurs et les placer dans le temps (placement et ordonnancement). Il faut aussi pouvoir respecter certaines contraintes de validité.

Les différents outils peuvent être comparés par rapport à leurs capacités dans ces différents domaines.

2.4.1 Quotientage

Le quotientage des occurrences de tâches a pour objectif :

1. de réduire le graphe de flot de données infini à un graphe fini pouvant être mis en correspondance avec un code fini ;
2. d'ajuster la taille des tâches ordonnancées en regroupant des tâches élémentaires en une tâche unique pour réduire le poids des surcoûts.

Le quotientage des occurrences de tâches peut être fait :

- sur la base des tâches : deux occurrences de la même tâche sont confondues (SynDEx). *On dépend entièrement de la granularité choisie par l'utilisateur pour la spécification.*
- en suivant les techniques habituelles du calcul scientifique pour arriver à des répartitions cycliques, blocs ou bloc-cycliques (APOTRES) *Exemple : recouvrement des calculs et des communications pour les occurrences d'une unique tâche élémentaire ; compromis entre surcoût de lancement et équilibre de charge entre processeurs ; plus fine est la spécification de l'utilisateur, plus grande est la marge de manœuvre d'*APOTRES.
- récursivement pour augmenter le nombre de degrés de liberté (PTOLEMY 0.7). *Exemple : répondre aux besoins d'une hiérarchie mémoire.*

2.4.2 Classe des fonctions de placement

Une fonction de placement dépend de son ensemble de départ, de son ensemble d'arrivée et de la nature des fonctions utilisées.

Le placement peut être automatique ou manuel. Le deuxième cas n'est pas pris en compte puisqu'il s'agit alors d'une spécification en entrée.

Le placement des occurrences de tâches peut donc :

- ne dépendre que de la tâche élémentaire (SynDEx). L'ensemble des fonctions de placement est alors fini et peut être énuméré. L'utilisateur peut aussi le faire lui-même (GEDAE) ;
- ne dépendre que de l'occurrence de la tâche mais non de la tâche elle-même (APOTRES). En fait, chaque tâche a une fonction de placement différente et chaque tâche peut utiliser un sous-ensemble de processeurs différents ; la limitation vient plutôt au niveau de l'ordonnancement : on ne peut activer qu'une seule tâche à la fois, i.e. un seul ensemble virtuel de processeurs à la fois ;
- permettre de spécifier des sous-ensembles de processeurs quelconques pour chaque tâche (FORTRAN HPF[43]) ;
- permettre de virtualiser la géométrie des processeurs (APOTRES, FORTRAN HPF).

2.4.3 Classe des fonctions d'ordonnancement

Il est possible d'en faire ressortir trois :

1. ordonnancement sur un graphe quotient (heuristique de listes : SynDEx)
2. ordonnancement affine monodimensionnel (APOTRES)
3. ordonnancement affine de dimension non bornée (langage de programmation procédural)

Remarque : APOTRES ne peut pas générer un ordonnancement équivalent à celui de SynDEx ou d'Array-OL hiérarchique parce que ces derniers forcent la notion d'itération *maîtresse*. Il faudrait davantage de dimensions temporelles pour obtenir les mêmes ordonnancements ; au moins deux : une dimension pour l'itération dite *maîtresse* et une pour les itérations propres à une itération *maîtresse*.

L'inverse est vrai aussi mais il n'y a apparemment pas de techniques automatiques permettant à SynDEx d'obtenir des ordonnancements de type APOTRES (déroulage de la boucle maîtresse et renommage sous différents noms d'une unique tâche, par exemple FFT0, FFT1,... FFT1023, pour éviter le quotientage trop violent par le nom de tâche, au risque d'une explosion combinatoire)

2.5 Qualité de la modélisation

La finesse de la modélisation du temps physique est plus ou moins bonne. APOTRES et SynDEx supposent que le temps d'exécution de deux tâches est égal à la somme de leurs temps d'exécution individuel. APOTRES suppose que le temps d'une communication est une fonction affine de son volume (ceci est équivalent à considérer qu'un surcoût de contrôle doit être ajouté au temps d'exécution de N occurrences de la même tâche ou au temps de communication). APOTRES et SynDEx supposent que les temps d'exécution élémentaires ne dépendent pas du cache.

La finesse de la modélisation de la topologie de la machine cible (et donc du temps aussi) est plus ou moins bonne. APOTRES suppose que tous les processeurs peuvent communiquer simultanément entre eux sans créer de conflits. SynDEx modélise finement des routes permettant à deux processeurs non directement connectés de communiquer via d'autres processeurs ou mécanismes (DMA par exemple). SynDEx prend en compte d'éventuels conflits de ressources dans les communications.

2.6 Conclusion

La dimension du problème est bien trop grande pour que l'on puisse vraiment comparer tous ces outils. En fournir une comparaison graphique en deux ou trois dimensions (algorithme traités, architectures traitées, fonctionnalités offertes) n'est pas raisonnable. Il existe cependant des points de rupture permettant de comparer de manière non continue les différents outils proposés. Les différents points de vue qu'on peut prendre sont partiellement redondants mais donnent des visions complémentaires d'une situation très complexe.

Cette analyse permet de constater qu'il n'existe pas, aujourd'hui, d'outils permettant le développement d'applications de traitement de signal radar ou sonar, allant de la conception à la compilation sur machine cible. À cela il y a plusieurs raisons, la première est tout simplement que peu d'outils de développement s'intéressent à cette famille d'algorithmes assez peu répandue, le SIMD et le SPMD étant des modes de programmation de moins en moins utilisés. La deuxième raison concerne la difficulté que comporte la résolution de ce

problème. Il n'est donc pas surprenant de constater que chacun des outils s'intéresse à des parties différentes, pouvant éventuellement avoir des intersections non vides les uns avec les autres. Certains sont voués à la spécification de ces applications, d'autres sont utilisés pour la validation, au travers de simulations, de leur placement, ou encore pour générer le code final.

Le point le moins abordé est la partie concernant le placement, *i.e.*, la parallélisation automatique de l'algorithme en considérant l'architecture cible. Ce travail est laissé au soin du programmeur qui n'a que son expérience et des outils de simulation pour valider son travail. APOTRES, au travers de PLC^2, a pour ambition de combler ce vide en offrant un support de développement dès la conception de l'algorithme et/ou la machine cible, dont les résultats sont les directives de placement garantissant le respect des contraintes opérationnelles souhaitées.

3 PLC^2

PLC^2 est la méthodologie de placement développée conjointement par THALES et ÉCOLE DES MINES DE PARIS. L'objectif de cette section est de présenter rapidement l'ensemble des modèles établis dans [32]. Nous rappellerons les modèles principaux de partitionnement, d'ordonnancement, de dépendances et de latence en prenant comme hypothèse architecturale une machine de type SPMD, sur laquelle tous les processeurs sont connectés les uns aux autres.

3.1 Le partitionnement

Nous avons vu dans la section 1.2 que le partitionnement est lié aux trois principales macro-contraintes à respecter, qui sont l'architecture (nombre de processeurs, capacité mémoire), le temps réel (latence, cadence au travers de l'ordonnancement des différentes partitions des différents nids de boucles) et bien sûr les dépendances entre nids de boucles. De ce fait, le partitionnement recherché doit permettre d'influencer les résultats pour l'une ou l'autre de ces macro-contraintes.

Le partitionnement choisi fait intervenir trois paramètres c, p et l permettant de le lier aux différents modèles définissant les trois grandeurs citées ci-dessus. Une hiérarchie a été définie entre ces divers paramètres : à un instant logique c, un processeur p effectuera les l itérations dont il a les données en mémoire. Nous parlerons de partitionnement 3D. Il correspond à une distribution bloc/cyclique d'HPF.

Puisqu'un nid de boucles comporte généralement plusieurs boucles, ces paramètres sont vectoriels : chaque composante de ces vecteurs correspond à une boucle du nid.

Comme il faut qu'il n'y ait ni calculs redondants, ni calculs oubliés, que le partitionnement soit choisi parallèle aux axes d'itérations, il a été défini deux matrices diagonales P et L telles qu'à chaque instant logique au moins un processeur effectue au moins un calcul[7] :

$$\text{nombre de processeurs utilisés} = \det(P) > 0 \quad \text{et} \tag{2.1}$$

$$\text{volume de calculs par événement par processeur} = \det(L) > 0 \tag{2.2}$$

Ainsi, nous avons la relation suivante entre le vecteur d'itérations i d'un nid de boucles d'espace d'itérations $\mathcal{I}$ et nos trois composantes de partitionnement ($\prec, \succ, \succeq$ et $\preceq$ définissent les opérateurs de comparaison pour l'ordre lexicographique) :

$$\forall i \in \mathcal{I} \quad \exists!(c,p,l) \begin{cases} 0 \preceq p \prec P.1 \\ 0 \preceq l \prec L.1 \\ 0 \preceq c \prec c_{max} \end{cases} \qquad i = LPc + Lp + l \tag{2.3}$$

$$\forall c \; \exists!(p,l) \begin{cases} 0 \preceq p \prec P.1 \\ 0 \preceq l \prec L.1 \\ 0 \preceq c \prec c_{max} \end{cases} \qquad \exists! i \in \mathcal{I} \qquad i = LPc + Lp + l \tag{2.4}$$

[7] Rappelons que le déterminant d'une matrice $m \times m$ A est égal au volume du parallélépipède d'un espace de dimension m, tel que les arêtes génératrices sont les lignes de A (notons que l'on peut prendre également les colonnes de A, obtenant ainsi un autre parallélépipède de volume néanmoins égal).

1 est le vecteur dont toutes les composantes sont égales à 1. Pour que 2.3 et 2.4 soient vraies, il faut que L, P et c_{max} vérifient :

machine SIMD

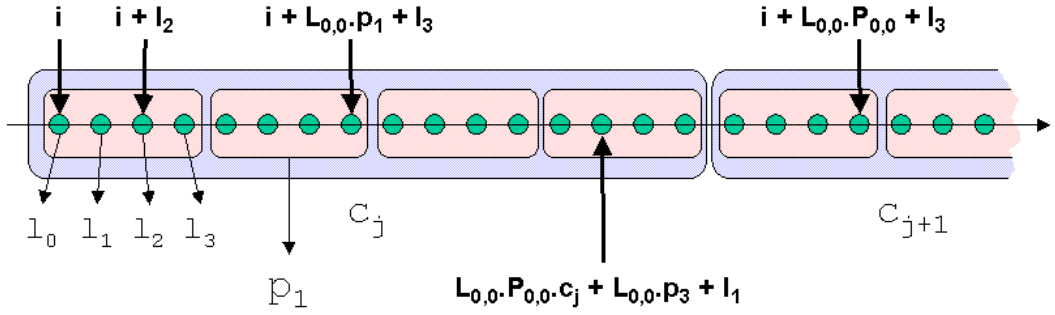
$$\begin{aligned} i_{max} &= LPc_{max} \\ \text{ou } c_{max} &= \frac{i_{max}}{LP} \end{aligned} \tag{2.5}$$

machine SPMD

$$c_{max} = \left\lceil \frac{i_{max}}{LP} \right\rceil \tag{2.6}$$

Les nids de boucles sont complètement parallélisables (sec. 1.1 p.4). Il est donc possible de considérer le partitionnement boucle par boucle. La figure 2.1 explique les contraintes 2.3, 2.4 et 2.5 pour une seule boucle.

FIG. 2.1 – Partitionnement d'une boucle.



Soit une itération i d'une k$^{\text{ième}}$ boucle séquentielle d'un nid de boucles. Supposons que cette itération corresponde au premier calcul exécuté sur le processeur 0. Sachant qu'un processeur exécute $L_{k,k}$ itérations, le premier calcul d'un processeur p correspond à l'itération $i + L_{k,k}$.p. De même, le premier calcul du processeur p au cycle logique suivant, puisque l'on utilise $P_{k,k}$ processeurs à chaque cycle logique, correspond à l'itération $i + L_{k,k}.P_{k,k} + L_{k,k}$.p. Enfin, le l$^{\text{ième}}$ calcul du processeur p au c$^{\text{ième}}$ cycle logique correspond donc à l'itération $L_{k,k}.P_{k,k}$.c + $L_{k,k}$.p + l.

3.2 L'ordonnancement

Le partitionnement définit des blocs de calculs par nid de boucles qu'il faut ordonner les uns par rapport aux autres. La composante temporelle du partitionnement est la composante c. Les fonctions d'ordonnancement jouent ici le rôle de projecteur sur un axe temporel logique commun à tous les nids de boucles de l'ordre total (ordre lexicographique) déjà existant (mais sur une échelle locale propre à chaque tâche) sur cette composante c. Ainsi il est plus aisé de comparer les dates d'exécutions pour satisfaire les dépendances.

C'est un ordonnancement monodimensionnel affine qui a été choisi [26]. Un bloc de calculs d'un $k^{\text{ième}}$ nid de boucles, ayant lieu à la date locale c^k, s'exécutera à la date logique d^k (c^k)

telle que :

$$\mathrm{d}^k\left(c^k\right) = \alpha^k.c^k + \beta^k \quad \text{avec } \alpha^k \text{ un vecteur et } \beta^k \text{ un scalaire.} \tag{2.7}$$

Pour s'assurer que cette projection conserve l'ordre total initial sur les axes temporels locaux, les $\alpha^k = \begin{pmatrix} \alpha_0^k \\ \vdots \\ \alpha_{n-1}^k \end{pmatrix}$ sont contraints par :

$$\begin{cases} & \alpha_{n-1}^k \geqslant 1 \\ \forall\, 0 \leqslant i < n-1, & \alpha_i^k \geqslant \alpha_{i+1}^k \times c_{max_{i+1}}^k \end{cases} \tag{2.8}$$

Ainsi,

$$\forall (c_1^k, c_2^k), \quad (c_1^k \prec c_2^k) \Leftrightarrow (\mathrm{d}^k\left(c_1^k\right) \ < \mathrm{d}^k\left(c_2^k\right)) \tag{2.9}$$

Le modèle de programmation SIMD/SPMD est introduit dans le modèle d'ordonnancement par le biais du modèle machine. Rappelons que ces modes de programmation signifient que tous les processeurs actifs effectuent au même instant la même instruction (pour le mode SIMD) ou le même programme (pour le mode SPMD). D'où la nécessité de garantir que deux blocs de calcul issus des partitionnements de deux nids de boucles différents ne s'exécutent pas à la même date logique. Pour ce faire, la solution adoptée consiste à poser les contraintes suivantes (N est le nombre de tâches contenues dans l'application et $k \in [0, N-1]$) :

$$\begin{cases} \forall\, 0 \leqslant i < n-1 & \text{N divise } \alpha_i^k \\ & \text{N divise } (\beta^k - k) \end{cases} \tag{2.10}$$

Ainsi, on a :

$$\forall k, k', \quad \forall i, j \qquad \left(\mathrm{d}^k(c_i^k) = \mathrm{d}^{k'}(c_j^{k'})\right) \Leftrightarrow ((k = k') \wedge (i = j))$$

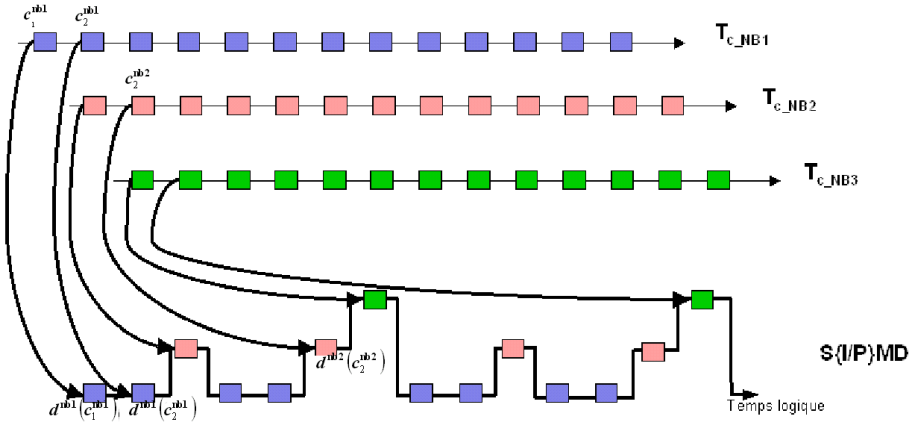
Preuve :

$$\begin{array}{rrcl} & \mathrm{d}^k(c_i^k) & = & \mathrm{d}^{k'}(c_j^{k'}) \\ \Leftrightarrow & \alpha^k.(c_i^k) + \beta^k & = & \alpha^{k'}.(c_j^{k'}) + \beta^{k'} \\ \Leftrightarrow & \mathrm{N}(\delta^k.(c_i^k)) + \mathrm{N} \times \gamma^k & = & \mathrm{N} \times (\delta^{k'}.(c_j^{k'})) + \mathrm{N} \times \gamma^{k'} \\ \Leftrightarrow & k - k' & = & \mathrm{N} \times (\delta^{k'}.(c_j^{k'}) - \delta^k.(c_i^k)) \end{array} \tag{2.11}$$

Or on a $k \in [0, N-1]$ et $k' \in [0, N-1]$, par conséquent (2.11) n'est vrai que si $k - k' = \mathrm{N} \times (\delta^{k'}.(c_j^{k'}) - \delta^k.(c_i^k)) = 0$. D'où $k = k'$ et comme on a également 2.9, alors $i = j$.

Cette méthode introduit des dates de non-événements qui ne sont, heureusement, pas prises en compte lors du calcul de latence en temps réel.

FIG. 2.2 – Projection de l'ordonnancement local vers l'ordonnancement global



Le rôle de la fonction d'ordonnancement est de pouvoir repérer l'ordre d'exécution des cycles de calculs de chacune des tâches sur un axe de temps commun. Ainsi si l'on ne sait pas comparer la date c_2^{nb1} avec la date c_2^{nb2}, chacune n'ayant de sens que sur les axes de temps locaux aux nids de boucles auxquels elles sont rattachées, leur projection respective sur l'axe de temps événementiel commun, $d^{\mathrm{nb1}}\left(c_2^{\mathrm{nb1}}\right)$ et $d^{\mathrm{nb2}}\left(c_2^{\mathrm{nb2}}\right)$, permet de le faire, et ainsi les ordonner.

3.3 Les dépendances

Par définition, il existe une dépendance entre deux blocs de calculs c^w et c^r issus du partitionnement de deux nids de boucles différents, lorsque l'un (r) consomme les données produites par l'autre (w). Nous noterons $\mathcal{D}\left(c^w, c^r\right)$ le prédicat prenant la valeur *vrai* lorsque cette dépendance existe. La contrainte de dépendance (ou de précédence) est alors simple :

$$\forall c^w, c^r \quad \mathcal{D}\left(c^w, c^r\right) \;\Rightarrow\; \left(\mathrm{d}^w\left(c^w\right) + 1 \leqslant \mathrm{d}^r\left(c^r\right)\right) \tag{2.12}$$

Remarque : Par extension, nous utiliserons la même notation lorsque nous parlerons de dépendance entre deux nids de boucles w et r : $\mathcal{D}\left(w, r\right) \Leftrightarrow \left(\exists c^w, c^r \quad \mathcal{D}\left(c^w, c^r\right)\right)$

D'autre part, si l'on appelle $m^{\{w,r\}}$ les vecteurs d'itérations internes de chacun des deux nids de boucles, $\Omega_{pav}^{\{w,r\}}$ et $\Omega_{fit}^{\{w,r\}}$ les matrices des fonctions d'accès (Chap. 1, sec. 1) on a :

$$\begin{array}{l} \forall c^w, c^r \\ \left(\begin{array}{l} \exists l^w, p^w, l^r, p^r, m^w, m^r \text{ t.q.} \\ \Omega_{pav}^w.\left(L^w P^w c^w + L^w p^w + l^w\right) + \Omega_{fit}^w.m^w = \Omega_{pav}^r.\left(L^r P^r c^r + L^r p^r + l^r\right) + \Omega_{fit}^r.m^r \end{array} \right) \\ \Leftrightarrow \\ \mathcal{D}\left(c^w, c^r\right) \end{array} \tag{2.13}$$

D'un point de vue contraintes, cette modélisation n'est pas satisfaisante à cause du quantificateur universel de la contrainte 2.12. En effet, il oblige à vérifier pour tous les couples possibles (c^w, c^r) s'ils sont ou non en dépendance. Ce nombre de couples peut être très grand voir infini si les nids de boucles contiennent une boucle infinie.

C'est donc par intention que les dépendances ont été modélisées. Cela a consisté à exprimer, dans la partie gauche de la contrainte 2.13, les c^w, c^r en fonction du reste. Après quelques

changements de variables et projections [32, Chap. 5], on obtient le système d'inéquations suivant pour chaque dimension du tableau sur lequel s'exercent les dépendances :

$$\begin{cases} 0 \leqslant c^w \leqslant c^w_{max} - 1 & \\ 0 \leqslant c^r \leqslant c^r_{max} - 1 & \\ -\Omega^r_{pav} L^r P^r . c^r + \Omega^w_{pav} L^w P^w . c^w \geqslant -\Omega^w_{fit}(m^w_{max} - 1) - \Omega^w_{pav}(L^w P^w - 1) & (D_{inf}) \\ -\Omega^r_{pav} L^r P^r . c^r + \Omega^w_{pav} L^w P^w . c^w \leqslant \Omega^r_{fit}(m^r_{max} - 1) + \Omega^r_{pav}(L^r P^r - 1) & (D_{sup}) \end{cases} \tag{2.14}$$

Ainsi l'espace des dépendances est délimité par le polytope définit par les six droites d'équations $c^w = 0$, $c^w = c^w_{max} - 1$, $c^r = 0$, $c^r = c^r_{max} - 1$, D_{inf} et D_{sup} (fig. 2.3).

Par ailleurs, il a été montré dans [32] que poser la contraintes 2.12 uniquement sur les sommets générateurs[8] du polytope de dépendance obtenu est équivalent à la poser sur tous les points contenus dans ce polytope. Or ces sommets générateurs ont des coordonnées rationnelles, ce qui est incompatible avec le langage de contraintes que nous utilisons (CLP(FD), sec. 3, p.15). Ce sont donc les sommets générateurs à coordonnées entières de la plus petite enveloppe convexe, contenant le polytope, qui sont utilisés.

Le respect des dépendances est assuré grâce à la contrainte

$$\forall c_s = \left(\begin{array}{cc} c^r_s, & c^w_s \end{array} \right) \qquad (\mathrm{d}^w(c^w_s) + 1 \leqslant \mathrm{d}^r(c^r_s)) \tag{2.15}$$

où c_s désigne les sommets générateurs, et $\left(\begin{array}{cc} c^r_s, & c^w_s \end{array} \right)$ leurs coordonnées. Le nombre de points à considérer ici est fini et égal à 6.

Signalons simplement que cette approximation introduit des dépendances fictives, que nous essayons d'éliminer dans le chapitre 5.

3.4 La latence

Le modèle de latence est assez simple. Rappelons que le GFD est sans circuit, par conséquent sa longueur est bornée par le plus long chemin. Le modèle de latence consiste à trouver le plus grand chemin de dépendances de l'application partitionnée (le chemin critique du GFD), de compter le nombre d'occurrences de chaque partition, et d'y introduire les durées en temps physique des calculs et des communications.

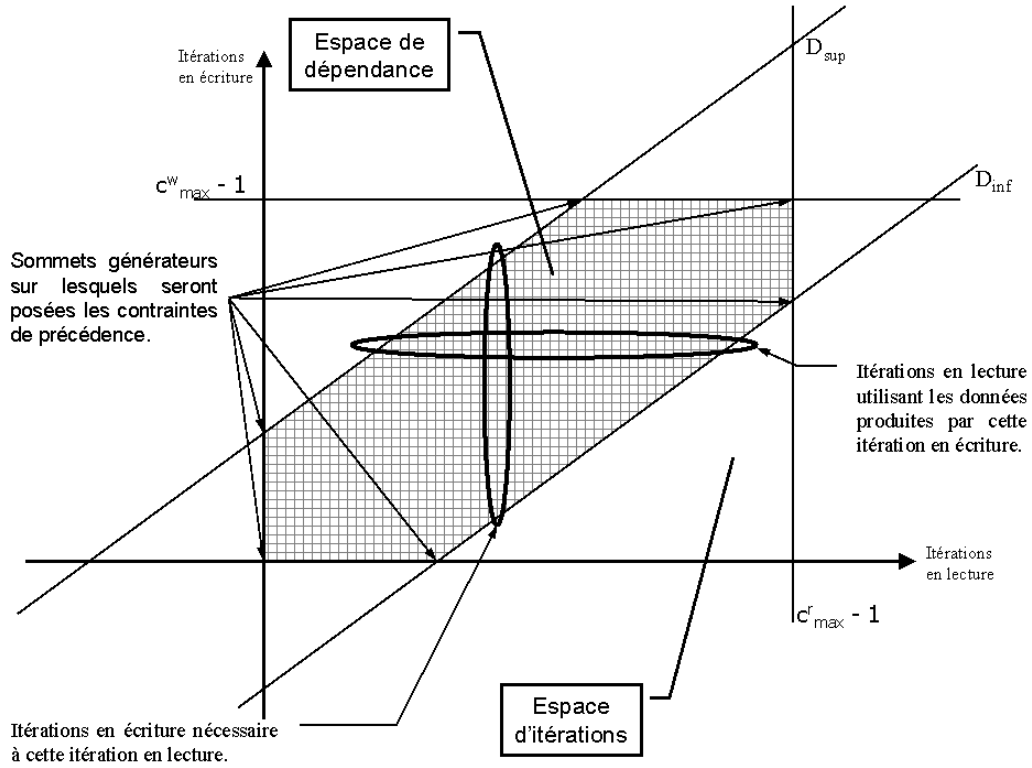
Les applications traitées dans [32] ont un GFD en forme de chaîne. Aussi, le chemin le plus long du graphe est constitué des arcs portant les plus grandes distances de dépendance entre deux nœuds. Là encore les sommets générateurs évoqués ci-dessus interviennent. La durée événementielle du chemin critique du GFD (notée Δ) est obtenue par :

$$\Delta = 1 + \sum_{w, \exists r, \mathcal{D}(w,r)} \max_s \left(\mathrm{d}^r(c^r_s) - \mathrm{d}^w(c^w_s) \right) \quad s \in \{\text{Sommets générateurs}\} \tag{2.16}$$

On définit la fonction $\mathrm{nbc}^k(\mathrm{deb}, \mathrm{fin})$ qui donne le nombre d'occurrences de la tâche k entre les dates événementielles deb et fin. Afin de majorer les durées on prend deb = 0 et fin = Δ (la phase initiale de start-up est plus longue que la phase stationnaire). Si l'on appelle

[8] Ce sont les points d'intersections des droites $c^j = cst$ (pour $j \in \{w, r\}$), D_{inf} et D_{sup}.

FIG. 2.3 – Le polytope de dépendances dans un cas monodimensionnel



$ct(k)$ le coût d'un bloc de calcul de la tâche k, intégrant sa communication, la contrainte de latence s'écrit :

$$\text{latence} \leqslant \sum_{k=0}^{N-1} \text{nbc}^k\,(0,\Delta) \times ct(k) \leqslant \text{latence}_{\max} \tag{2.17}$$

Nous verrons comment calculer Δ lorsque le GFD n'est pas une chaîne dans le chapitre 3.

3.5 Conclusion

Nous avons résumé ici quelques résultats de [32]. Il faut garder à l'esprit que la plupart des paramètres (P, L, c, α, β) ne sont pas connus. C'est ce qui fait toute la difficulté de la résolution globale du placement qui va consister à chercher une solution pour tous ces paramètres simultanément.

Chapitre 3

Le domaine applicatif

Les phases de définition, développement et validation d'une application sous contraintes de temps réel et d'embarquabilité comprennent trois étapes :

1. une étude théorique du problème conduisant à la modélisation fonctionnelle d'un algorithme ;
2. une étude sur le choix de l'architecture matérielle cible construite à partir de processeurs ou/et de dispositifs matériels spécialisés ;
3. le développement matériel/logiciel induisant l'implantation de l'algorithme (sous forme d'un programme) sur l'architecture cible.

Si la phase de parallélisation consiste à extraire le parallélisme potentiel d'un programme séquentiel, la problématique du placement est alors d'établir la projection d'un algorithme (architecture logicielle d'application), dont le parallélisme est ou non spécifié, sur une architecture cible (architecture matérielle multi-processeurs) et cela de manière efficace (respect des contraintes temps réel, minimisation des ressources matérielles), ce qui se traduit par un problème d'optimisation fortement combinatoire. Tout cela est réalisable dans un cadre formel, où algorithme, architecture et implantation sont décrits par des modèles permettant d'analyser (vérifier), de composer et de réduire (optimiser) l'application étudiée. L'objet de ce chapitre est de présenter les extensions prises en compte dans le domaine applicatif (description fonctionnelle de l'algorithme).

En effet, les applications rencontrées dans le monde Radar et même Télécom[1], ont fait émerger différentes voies d'extension du modèle applicatif. Dans ce chapitre, nous allons présenter une application radar, les extensions applicatives qu'elle entraîne, ainsi que leurs impacts sur les différents modèles.

Le domaine applicatif couvert était initialement celui du TS de type intensif, systématique, déterministe et structuré (ex : compression d'impulsion), que l'on rencontre plus fréquemment dans le monde du sonar. Ces applications de TSS sont composées de déclarations de tâches, que l'on peut exprimer par des nids de boucles parfaitement imbriqués et

[1]Parallèlement aux travaux de cette thèse, le projet RNRT PROMPT[7], impliquant une partie de notre équipe, a permis d'appréhender le domaine d'applications Télécom. Nous n'avons pas traité ce domaine dans cette thèse, mais suivi avec intérêt les résultats de ces travaux.

parallèles. Dans le cadre du traitement du signal de bas niveau, c'est à dire celui TSS, il est courant d'utiliser le tableau comme structure de données principale. En effet, le tableau est un élément fondamental de la classe des applications de TS de bas niveau. Il permet de représenter l'ensemble des données de bas niveau. Le tableau s'adapte facilement aux dimensions du système de capteurs, et permet de donner informatiquement la formulation mathématique des traitements. Ainsi, les indices des variables composant les formules deviennent des indices de tableaux. La linéarité des domaines d'itérations sous-jacents aux différents traitements permet de définir un graphe de contrôle, et de spécifier les formes linéaires des différents accès pour chaque traitement élémentaire. Les accès élémentaires à un tableau sont alors des fonctions affines des indices de ce tableau, des constantes et des variables scalaires privées. Les bornes de ces scalaires sont des constantes déterminées à la compilation. L'espace d'itération d'un traitement, est quant à lui complètement défini par des fonctions affines portant uniquement sur les indices des différents tableaux. La méthodologie mise en œuvre dans [32] ne supporte pas de notion de retard, ni de notion de conditionnement et ne considère que des applications spécifiées comme une séquence de tâches.

Dans la suite de ce chapitre, nous allons dans un premier temps (cf. section 1) présenter en détail l'application Radar MFR (Moyenne Fréquence Récurrence) qui est représentative des applications Radar et suffisamment dimensionnante pour nous servir d'application de référence. Nous exhiberons ensuite les extensions applicatives qu'elle implique, soulignant les caractéristiques non traitées dans [32] au travers des applications sonar. Finalement dans la section suivante, nous analyserons l'impact de la prise en compte de ces nouvelles spécificités sur les différents modèles de la fonction placement, modèles décrits dans [32] et dont une synthèse a été donnée au chapitre précédent.

1 Un mode Radar : Moyenne Fréquence de Récurrence

Cette section a pour objet la présentation du mode radar **M**oyenne **F**réquence de **R**écurrence (MFR). Celle-ci se fera au travers de l'étude que nous avons effectuée à partir de la description des tâches élémentaires (*i.e.*, les fonctions d'une bibliothèque de traitement du signal) qui la compose.

Cette étude a tout d'abord consisté à «reconstituer» l'application pour en obtenir une description fonctionnelle dans le formalisme utilisé par PLC^2. C'est-à-dire que :

- toutes les données (*i.e.*, les signaux échantillonnés et les coefficients) sont stockées dans des tableaux ;
- les fonctions d'accès sont positives ;
- chaque nid de boucles ne produit qu'un seul tableau de données ;
- toutes les boucles commencent à 0 ;

Nous donnons une description générale du fonctionnement du mode MFR. Nous voyons ensuite les spécificités de cette application RADAR et les problèmes rencontrés pour utiliser notre formalisme de description fonctionnelle d'application TSS.

Remarque : Toutes les valeurs numériques utilisées ne sont pas celles utilisées en situation réelle, pour des raisons évidentes de confidentialité. Cependant elles ont été choisies dimensionnantes pour donner un caractère crédible à cette étude.

1.1 Descriptif général

Le traitement du signal exploite indépendamment les signaux reçus pour des pointages successifs. Il peut y avoir un changement de mode entre deux pointages. Le mode MFR est l'un d'eux. Le traitement fait l'acquisition du signal fourni par les capteurs pendant la durée d'un pointage. Les derniers résultats seront transmis après la fin du pointage. En règle générale, tout signal acquis est traité jusqu'au bout, y compris en cas de changement de mode. Le signal est reçu via N_{ca} canaux (les voies) spécialisés ou non. Le signal a une dimension temporelle et une dimension spatiale.

Nous appellerons M le pointage courant. Il comporte N_g groupes de N_{ra} rafales d'impulsions émises. Deux groupes de rafales sont séparés par des temps morts.

La durée d'une rafale est variable. Chaque rafale correspond à N_{rec} impulsions émises (*i.e.*, récurrences). Chacune de ces dernières correspond à un temps d'émission et à un temps d'écoute.

Une impulsion émise est composée de N_m moments de durée constante. La phase d'écoute correspond à N_{cd} cellules de résolutions en distance.

De ces paramètres (récapitulés dans le tableau 3.1) nous en déduisons les grandeurs physiques qui vont servir d'itérateurs de boucles (répertoriés dans le tableau 3.2).

Paramètre	**Signification**	**Domaine de définition**
M	le pointage courant	-
N_{ca}	le nombre de canaux spécialisés	entier
$N_g(M)$	le nombre de groupes de rafales par pointage	entier
$N_{ra}(M)$	le nombre de rafales par groupe	entier
$N_{rec}(M)$	le nombre de récurrences	entier
$N_{cd}(M, g)$	le nombre de cellules de résolution (g indique le groupe de rafale)	entier
N_m	le nombre de moments	entier

TAB. 3.1 – Récapitulatif des différents paramètres du problème

Indice	**Signification**	**Domaine de définition**
ca	la voie S, E ou C,	$0 \leqslant ca < N_{ca}$
g	le groupe de rafales,	$0 \leqslant g < N_g(M)$
ra	la rafale du groupe g,	$0 \leqslant ra < N_{ra}(M)$
rec	la récurrence,	$0 \leqslant rec < N_{rec}(M)$
d	le doppler ambigu,	$0 \leqslant d < N_{rec}(M)$
cd	la case distance,	$0 \leqslant cd < N_{cd}(M, g)$
xy	la valeur de la donnée dans $\mathbb{C}$,	$0 \leqslant xy < 2$

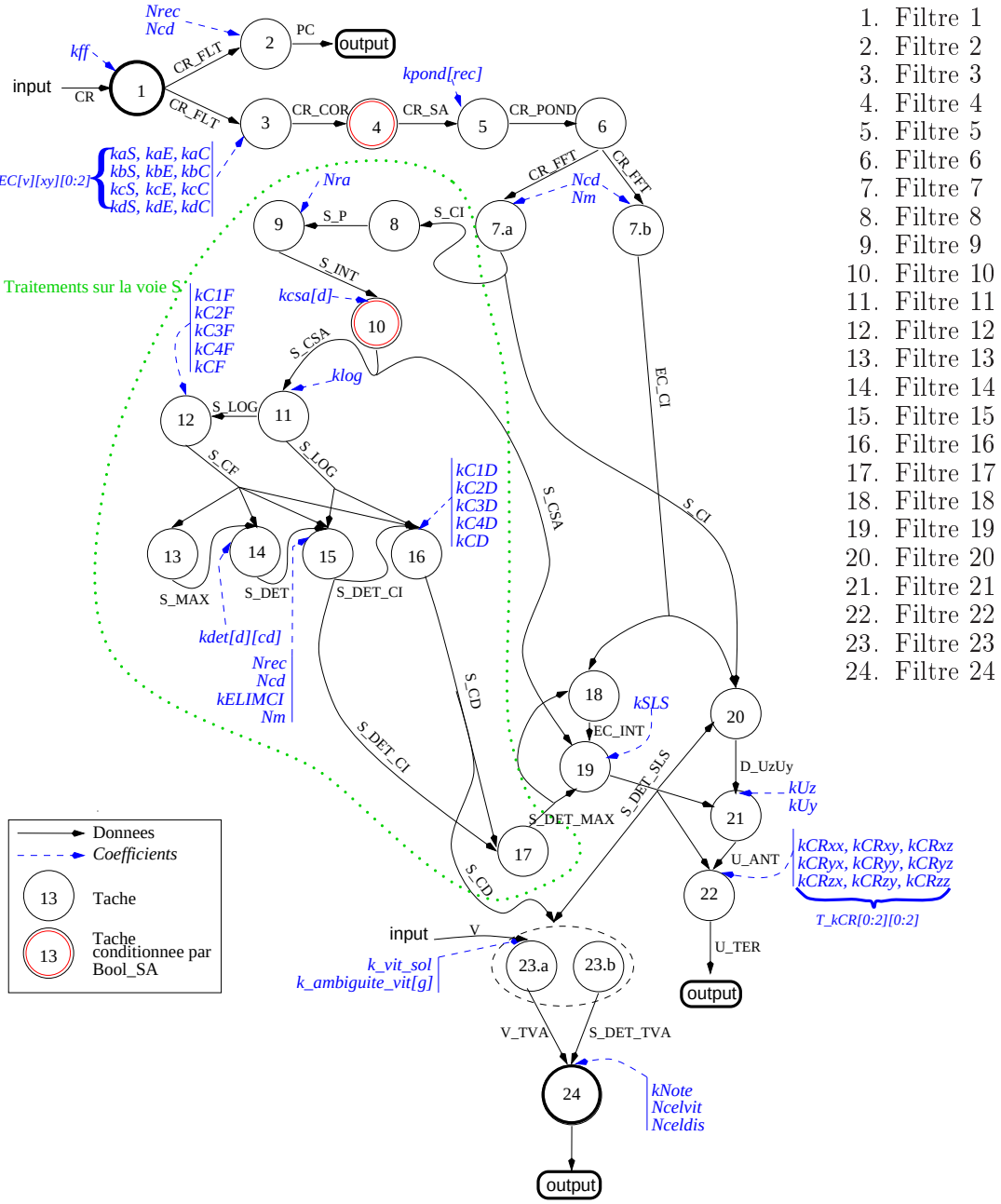
TAB. 3.2 – Liste des indices de boucles

Notons que N_{cd} varie de façon pseudo-aléatoire, de groupe à groupe dans un pointage (*i.e.*, N_{cd} est fonction de M et de $\mathtt{g}$). C'est-à-dire que sur cette dimension la borne supérieure de l'itérateur correspondant n'est pas une constante connue, mais une donnée d'un tableau indexé par $\mathtt{g}$. Or PLC^2 ne supporte pas ce type d'accès puisqu'une des hypothèses

est que les boucles issues d'un même nid sont indépendantes les unes des autres. En première approximation, nous pourrons considérer que les $N_{cd}(M, g)$ sont égaux pour tout g.

Le mode MFR est constitué de 24 tâches (*i.e.*, les nids de boucles). Ce sont des filtres de nature différente que nous ne pouvons définir davantage. Le GFD associé est donné en page 54.

FIG. 3.1 – Graphe de précédence du mode MFR.



1. Filtre 1
2. Filtre 2
3. Filtre 3
4. Filtre 4
5. Filtre 5
6. Filtre 6
7. Filtre 7
8. Filtre 8
9. Filtre 9
10. Filtre 10
11. Filtre 11
12. Filtre 12
13. Filtre 13
14. Filtre 14
15. Filtre 15
16. Filtre 16
17. Filtre 17
18. Filtre 18
19. Filtre 19
20. Filtre 20
21. Filtre 21
22. Filtre 22
23. Filtre 23
24. Filtre 24

(Les numéros dans les cercles font référence aux positions dans la liste des tâches page 54)

1.2 Structure de données

La première des choses à faire est de changer les structures en tableaux. En effet, le formalisme de description d'application de PLC^2 n'accepte comme structure de données que les tableaux multidimensionnels. Deux transformations sont possibles : (1) séparation des voies, (2) regroupement des voies. Le choix de l'une ou l'autre dépend de l'utilisation des tableaux permise et souhaitée.

1.2.1 Un tableau par voie

Une possibilité est de séparer les voies en leur attribuant un tableau. C'est-à-dire que nous considérerons 3 tableaux (1 par voie) **S**, **E** et **C**, chacun de dimensions 4 (**g**, **ra**, **rec**, **cd**) + 1 (de taille 2 pour **x** et **y**).

Soit,

$$\begin{array}{ll} & \texttt{S}[N_g(m)][N_{ra}(m)][N_{rec}(m)][N_{cd}(m,g)][2] \\ & \texttt{E}[N_g(m)][N_{ra}(m)][N_{rec}(m)][N_{cd}(m,g)][2] \\ & \texttt{C}[N_g(m)][N_{ra}(m)][N_{rec}(m)][N_{cd}(m,g)][2] \\ \text{où} & \texttt{\{S,E,C\}}[\ldots][\ldots][\ldots][\ldots][0] = \texttt{\{S,E,C\}.x} \\ \text{et} & \texttt{\{S,E,C\}}[\ldots][\ldots][\ldots][\ldots][1] = \texttt{\{S,E,C\}.y} \end{array}$$

Cette structuration des données implique nécessairement que certaines tâches seront triplées. Mais elle conserve une indépendance directe des voies les unes par rapport aux autres.

1.2.2 Un vecteur de voies

Une autre possibilité est de ne prendre qu'un seul tableau qui contiendrait les 3 précédents. La première dimension permettrait de choisir la voie sur laquelle le calcul s'effectuera. C'est ce qui se rapproche le plus de l'aspect "structure" initial. Il suffit alors de rajouter une boucle pour les voies (au lieu de dupliquer la tâche). La partie des calculs ne traitant que la voie `S` verra ses tableaux perdre cette dimension supplémentaire. Nous avons alors un tableau initial à 6 dimensions :

$$\texttt{SEC}[N_{ca}][N_g(m)][N_{ra}(m)][N_{rec}(m)][N_{cd}(m,g)][N_{xy}]$$

avec $N_{ca} = 3$. Nous prendrons la convention suivante : $0 =$ `S`, $1 =$ `E` et $2 =$ `C`.

C'est cette deuxième approche que nous choisissons pour la suite. Cela évite de dupliquer la plupart des tâches (comme la conversion réel/complexe, par exemple), ce qui aurait pour effet d'augmenter la complexité de l'ensemble. De plus, la nature du graphe de précédence (Fig. 3.1 p.54) représentant l'application ne permet pas de traiter les voies de manière concurrente. En effet, avant d'effectuer les tâches 18 et 20, il faut exécuter les tâches 8 à 17 (traitements sur la voie S).

1.2.3 Tableaux utilisés

Ci-dessous la liste des tableaux utilisés avec leur dimension et les références aux sections de ce document décrivant les tâches productrices et consommatrices. Le tiret signifie qu'il

s'agit d'un signal d'entrée ou de sortie. Les coefficients n'apparaissent pas dans cette table (*c.f.*, §1.2.4).

Nom	**Dimension**	**Type** (taille en octet)	**Produit par**	**Consommé par**
tab1	6 : [ca, g, ra, rec, cd, xy]	entier (2)	-	1
tab2	6 : [ca, g, ra, rec, cd, xy]	réel (4)	1	2, 3
tab3	2 : [g, ra]	réel (4)	2	-
tab4	6 : [ca, g, ra, rec, cd, xy]	réel (4)	3	4
tab5	6 : [ca, g, ra, rec, cd, xy]	réel (4)	4	5
tab6	6 : [ca, g, ra, rec, cd, xy]	réel (4)	5	6
tab7	6 : [ca, g, ra, rec, cd, xy]	réel (4)	6	7.a, 7.b
tab8	5 : [g, ra, rec, cd, xy]	réel (4)	7.a	8, 20
tab9	4 : [g, ra, d, cd]	réel (4)	8	9
tab10	3 : [g, d, cd]	réel (4)	9	10
tab11	3 : [g, d, cd]	réel (4)	10	11, 19
tab12	3 : [g, d, cd]	réel (4)	11	12, 15, 16
tab13	3 : [g, d, cd]	réel (4)	12	13, 14, 15, 16
tab14	3 : [g, d, cd]	booléen (2)	13	14
tab15	3 : [g, d, cd]	booléen (2)	14	15
tab16	3 : [g, d, cd]	booléen (2)	15	16, 17
tab17	3 : [g, d, cd]	réel (4)	16	17, 23
tab18	3 : [g, d, cd]	booléen (2)	17	18, 19
tab19	6 : [ca, g, ra, rec, cd, xy]	réel (4)	7.b	18, 20
tab20	4 : [g, ra, d, cd]	réel (4)	18	19
tab21	3 : [g, d, cd]	booléen (2)	19	20, 21, 22, 23
tab22	5 : [zy, g, ra, d, cd]	réel (4)	20	21
tab23	5 : [g, ra, d, cd, xyz]	réel (4)	21	22
tab24	5 : [g, ra, d, cd, xyz]	réel (4)	22	-
tab25	3 : [g, d, cd]	réel (4)	-	23
tab26	3 : [g, d, cd]	réel (4)	23.a	24
tab27	3 : [g, d, cd]	booléen (2)	23.b	24
tab28	2 : [100, 2]	entier (2)	24	-

TAB. 3.3 – Liste des tableaux utilisés.

Nous pouvons constater, au vu du graphe de précédence (fig.3.1, p.54) issu de ce tableau, que nous allons être confrontés à un problème de mémoire. En effet, considérons, par exemple, la tâche portant le numéro 10 (appelée T10). Elle produit des résultats utilisés par la T19. Mais pour s'exécuter, cette dernière doit attendre les résultats de T17, qui dépend de T16 et T15, qui dépendent de T14, T13, T12 et T11. Or T11 dépend également de T10. C'est-à-dire que les résultats de T10 seront stockés en mémoire pendant un certain temps durant lequel elles ne seront pas exploitées. On retrouve la même situation avec les tâches 7.{a,b}.

1.2.4 Les coefficients (paramètres)

De la même manière que précédemment, la table ci-dessous présente les coefficients utilisés. Ces coefficients sont calculés à chaque pointage et constants pendant sa durée.

Nom	Dimension	Type (taille en octet)	Utilisé par
Nrec	-	entier (2)	2, 15
Ncd	-	entier (2)	2, 7, 15
Nm	-	entier (2)	7, 15
Nra	-	entier (2)	9
k1	-	réel (4)	1
k200, k210, k220, k230 k201, k211, k221, k231 k202, k212, k222, k232	T_k2 3 :[ca][2][2]	réel (4)	3
k5	1 :[rec]	réel (4)	5
k10	1 :[d]	réel (4)	10
k11	-	réel (4)	11
k12, k121, k122, k123, k124	-	entier (2)	12
k14	2 :[d][cd]	réel (4)	14
k15	-	réel (4)	15
k16, k161, k162, k163, k164	-	entier (2)	16
k19	-	réel (4)	19
k211, k212	-	réel (4)	21
k2200, k2201, k2202 k2210, k2211, k2212 k2220, k2221, k2222	T_k22 2 :[3][3]	réel (4)	22
k231	-	réel (4)	23
k232	1 :[g]	entier (2)	23
k24	-	entier (2)	24
N241	-	entier (2)	24
N242	-	entier (2)	24

TAB. 3.4 – Liste des différents coefficients utilisés.

1.3 Les spécificités du mode MFR

La première spécificité du mode MFR à prendre en compte est la structure de son DFG qui n'est pas une chaîne. Certains modèles de [32] considèrent implicitement que le flot des données est linéaire. Cela influence directement la modélisation de la latence qui repose sur la recherche du chemin le plus long (Chap. 2, section 3.4) du GFD. L'ordonnancement est également concerné de part le modèle machine SPMD/SIMD puisque l'on exploite un numéro attribué à chacune des tâches pour que leurs calculs ne s'exécutent pas simultanément. Si dans le cas d'une chaîne, ce numéro est déterminé, dans le cas d'un graphe quelconque acyclique, numéroter a priori les tâches oriente dans une direction la recherche d'une solution.

La description fonctionnelle de ce mode radar a fait apparaître plusieurs problèmes d'ordre et de complexité différents. Tout d'abord, nous avons constaté qu'il était nécessaire de prendre en compte des fonctions d'accès aux données affines. Soit pour pouvoir faire des translations du domaine d'itération d'un nid de boucles pour revenir à 0, soit parce que ce type d'accès est clairement exprimé dans la description fonctionnelle d'une tâche.

Ensuite, nous avons rencontré des situations de débordement de tableau (tâches : 4,13,17) dévoilant un manque de spécifications pour les cas particuliers des début et fin de tableau.

Le point le plus spécifique du mode radar MFR est son aspect dynamique qui s'exprime par le conditionnement de l'exécution de TE. On a deux types de conditionnement :

1. par une constante (tâches 4,10) ;
```
cste = 0
 (...)
if (cste = 0)
  TE(IN, OUT)
end if
```

2. par les données (tâches 13, 15, 16, 18, 19, 20, 21, 22, 13).
```
if (IN[i] > IN[i + 1])
  TE(IN, OUT)
end if
```

La constante est déterminée sur un pointage et change potentiellement d'un pointage à l'autre. Aussi, tant que l'on ne s'intéresse qu'à une instance du mode MFR, ce paramètre peut être considéré comme les autres (*i.e.*, on prend alors en compte la valeur qu'il a pour cette instance). Le plus difficile est le conditionnement par les données. PLC^2 est, initialement, étudié pour les applications statiques que sont celles du traitement du signal systématique. L'étude de ce comportement dynamique dans le placement, tel que nous le concevons, n'est pas l'objet principal de cette thèse. Nous verrons dans la suite quel compromis nous avons choisi.

Par ailleurs, la description fonctionnelle du mode radar MFR utilise des fonctionnalités non définies dans notre formalisme. Notamment, les coefficients utilisés dans certaines tâches (3, 22) sont scalaires (PLC^2 ne sait manipuler que des tableaux). Certaines fonctions d'accès (nids de boucles 4, 16) sont à coefficients négatifs, alors que les matrices d'accès sont supposées définies positives (section 1.1, p. 4).

Les tâches ayant plus d'un tableau de sortie (7, 18, 20, 23) sont également problématique, car nous supposons qu'une tâche ne produit qu'un tableau. De même, l'utilisation de manière différente d'un tableau (*i.e.*, possédant plusieurs fonctions d'accès) dans un même nid de boucles n'est pas une utilisation autorisée par notre ensemble d'hypothèses initial (tâches 15, 16).

Enfin, les derniers problèmes posés par la description formelle initiale de ce mode radar concernent les bornes de boucles pouvant être paramétrées (tâche 15) ou à croissance non linéaire (tâches 12, 16).

La suite de ce chapitre expliquera comment nous avons, soit augmenté notre formalisme de description, soit quel compromis nous avons fait entre degré de formalisation et difficulté du problème soulevé.

1.4 Récapitulatif quantitatif des différents nids de boucles

Le tableau 3.6 donne, pour chacune des tâches, la durée[2] de la tâche élémentaire (TE), le nombre théorique d'itérations de la TE, ses valeurs minimale et maximale et enfin la durée de la tâche. Ces valeurs sont calculées, en considérant une exécution monoprocesseur, à partir des coefficients du tableau 3.5.

Nca	3	Nrec	128	Nd	128
Ng	13	Ncd	32	k122=k124	5
Nra	7	Nm	11	k162=k164	5
NDETDOP	5	NDETFRE	128		

Ces valeurs permettent de trouver des résultats cohérents mais sont non représentatives d'une quelconque situation réelle.

TAB. 3.5 – Valeur numériques des paramètres utilisées pour les expérimentations.

Ces informations sur les probabilités moyennes d'exécution des nids de boucles permettent d'identifier les tâches prépondérantes. Celles-ci permettent de savoir où se trouvent les calculs importants. Non exploitées ici, ces probabilités pourrait être utilisées pour cibler davantage la partie de l'application où la parallélisation permettrait de gagner (en temps de calcul) de façon importante. En l'occurrence, la majorité des calculs ont lieu au début du mode MFR, principalement par le nid de boucles 6 qui représente un peu plus de 46% du total. L'optimisation du placement de cette application pourrait tenir compte de ces indications. En effet, une distribution optimale des données utilisées par un nid de boucles en fin de traitement donnera un gain global faible.

1.5 Conclusion

Nous avons illustré, au travers de la description du mode radar MFR, l'élargissement du domaine applicatif couvert par PLC^2 depuis [32], ainsi que sa complexité induite (en termes de nombre de nids de boucles, de profondeur de nids de boucles, de tableaux, de structure du GFD). En effet, nous sommes quelque peu éloignés des traitements sonar d'origine de type *veille bande large* (VBL). Nous sommes passés d'applications comportant peu de nids de boucles (7 pour la VBL) à une application composée de 24 nids de boucles (30 après une réécriture permettant d'utiliser le formalisme de PLC^2). De plus, les nids de boucles du mode radar MFR ont une profondeur allant de trois à cinq (contre au plus trois pour la VBL). Cet accroissement de complexité permet de tester la robustesse de l'approche modélisation concurrente/contraintes.

L'aspect dynamique, donné par les bornes de boucles paramétrées et l'exécution de TE conditionnée par des données, du mode MFR sera approximé par le pire cas (toutes les itérations exécutent la TE). Cependant cette considération ne peut que nous empêcher de trouver une solution optimale de placement. Car dans le cas où nous en trouvons un, celui-ci fera une forte sur-consommation de la mémoire par rapport à la quantité utile, et sur-mesurera la latence réelle. Dans le cas où nous ne trouvons pas de placement, cela ne signifiera pas pour autant qu'il n'en existe pas, mais que l'accumulation des pires cas nous empêche de les trouver (du fait des sur-approximations). Malgré tout, le pire cas est une situation

[2]Toutes les durées sont exprimées en cycles.

Tâches	Coût de la TE	Nombre d'itération de la TE		Nombre d'itération de la TE donnant lieu à un calcul	Cout de la tache	
		Théorique	A.N.	A.N. * % exec	Nb_it * Cout_TE	Nb_it_actives * Cout_TE
1	2	Nca*Ng*Nra*Nrec*Ncd	1 118 208	1 118 208	2 236 416	2 236 416
2	20 480	Ng*Nra	91	91	1 863 680	1 863 680
3	6	Nca*Ng*Nra*Nrec*Ncd	1 118 208	1 118 208	6 709 248	6 709 248
4	2	Nca*Ng*Nra*Nrec*Ncd	1 118 208	1 118 208	2 236 416	2 236 416
5	2	Nca*Ng*Nra*Nrec*Ncd	1 118 208	1 118 208	2 236 416	2 236 416
6	4 480	Nca*Ng*Nra*Ncd	8 736	8 736	39 137 280	39 137 280
7.a1	22	Ng*Nra*Nrec*(Ncd-Nm)	244 608	244 608	5 381 376	5 381 376
7.a2	22	Ng*Nra*Nrec*Nm	128 128	128 128	2 818 816	2 818 816
7.a	0	Ng*Nra*Nrec	11 648	11 648	0	0
7.b1	22	(Nca-1)*Ng*Nra*Nrec*(Ncd-Nm)	489 216	489 216	10 762 752	10 762 752
7.b2	22	(Nca-1)*Ng*Nra*Nrec*Nm	256 256	256 256	5 637 632	5 637 632
7.b	0	(Nca-1)*Ng*Nra*Nrec	23 296	23 296	0	0
8	3	Ng*Nra*Nrec*Ncd	372 736	372 736	1 118 208	1 118 208
9	14	Ng*Nrec*Ncd	53 248	53 248	745 472	745 472
10	2	Ng*Nrec*Ncd	53 248	53 248	106 496	106 496
11	12	Ng*Nrec*Ncd	53 248	53 248	638 976	638 976
12	15	Ng*Nrec*Ncd	53 248	53 248	798 720	798 720
13	7	Ng*Nrec*Ncd	53 248	53 248	372 736	372 736
14	5	Ng*Nd*Ncd	53 248	26 624	266 240	133 120
15	40	Ng*Nd*Ncd	53 248	2 080	2 129 920	83 200
16	15	Ng*Nd*Ncd	53 248	1 664	798 720	24 960
17	6	Ng*Nd*Ncd	53 248	1 664	319 488	9 984
18	10	(Nca-1)*Ng*Nrec*Ncd	106 496	1 664	1 064 960	16 640
19	2	Ng*Nrec*Ncd	53 248	832	106 496	1 664
20	15	Ng*Nra*Nd*Ncd	372 736	40 768	5 591 040	611 520
21	10	Ng*Nra*Nd*Ncd	372 736	40 768	3 727 360	407 680
22	15	Ng*Nra*Nd*Ncd	372 736	40 768	5 591 040	611 520
23.a	5	Ng*Nd*Ncd	53 248	832	266 240	4 160
23.b	5	Ng*Nd*Ncd	53 248	832	266 240	4 160
24	53 248	Ng*Nd*Ncd	1	1	53 248	53 248
TOTAUX			7 871 228	6 432 284		

Tâches	Cout cumulé		Nombre de données en entrée	Nombre de données en sortie OK	% exécution	Poids calculé de la tache (%)	
	sans % Exec	avec % Exec			NbCalc effectifs / NbCalc théoriques	sans % Exec	avec % Exec
1	2 236 416	2 236 416	1 118 208	1 118 208	100,00	2,17166	2,63845
2	4 100 096	4 100 096	372 736	91	100,00	1,80972	2,19871
3	10 809 344	10 809 344	1 118 208	1 118 208	100,00	6,51499	7,91535
4	13 045 760	13 045 760	1 118 208	1 118 208	100,00	2,17166	2,63845
5	15 282 176	15 282 176	1 118 208	1 118 208	100,00	2,17166	2,63845
6	54 419 456	54 419 456	1 118 208	1 118 208	100,00	38,00414	46,17287
7.a1	59 800 832	59 800 832	244 608	244 608	100,00	5,22557	6,34877
7.a2	62 619 648	62 619 648	128 128	128 128	100,00	2,73720	3,32555
7.a	62 619 648	62 619 648	372 736	372 736	100,00	0	0
7.b1	73 382 400	73 382 400	489 216	489 216	100,00	10,45114	12,69754
7.b2	79 020 032	79 020 032	256 256	256 256	100,00	5,47441	6,65109
7.b	79 020 032	79 020 032	745 472	745 472	100,00	0,00000	1,00000
8	80 138 240	80 138 240	372 736	372 736	100,00	1,08583	1,31922
9	80 883 712	80 883 712	372 736	53 248	100,00	0,72389	0,87948
10	80 990 208	80 990 208	53 248	53 248	100,00	0,10341	0,12564
11	81 629 184	81 629 184	53 248	53 248	100,00	0,62048	0,75384
12	82 427 904	82 427 904	53 248	53 248	100,00	0,77559	0,94230
13	82 800 640	82 800 640	53 248	26 624	100,00	0,36194	0,43974
14	83 066 880	82 933 760	26 624	2 080	50,00	0,25853	0,15705
15	85 196 800	83 016 960	2 080	1 664	3,91	2,06825	0,09816
16	85 995 520	83 041 920	1 664	1 664	3,13	0,77559	0,02945
17	86 315 008	83 051 904	1 664	832	3,13	0,31024	0,01178
18	87 379 968	83 068 544	832	832	1,56	1,03413	0,01963
19	87 486 464	83 070 208	832	832	1,56	0,10341	0,00196
20	93 077 504	83 681 728	832	832	10,94	5,42916	0,72145
21	96 804 864	84 089 408	832	832	10,94	3,61944	0,48097
22	102 395 904	84 700 928	832	832	10,94	5,42916	0,72145
23.a	102 662 144	84 705 088	832	832	1,56	0,25853	0,00491
23.b	102 928 384	84 709 248	832	832	1,56	0,25853	0,00491
24	102 981 632	84 762 496			100,00	0,05171	0,06282
TOTAUX	102 981 632	84 762 496					102,00

Les différents compromis réalisés nous ont conduits à dupliquer ou couper certains nids de boucles. Pour rester sémantiquement cohérent, les nids de boucles coupés sont ensuite fusionnés. Afin de ne pas perturber les calculs, ces tâches de fusion sont pondérées de durées nulles. Ce qui conduit à des coûts nuls également.
A.N. : *Application Numérique*

TAB. 3.6 – Récapitulatif quantitatif de la proportion des nids de boucles.

possible qui ne doit pas empêcher le radar de fonctionner. Aussi trouver une solution de placement, sans prendre en compte ces situations de pire cas, signifie que la machine cible utilisée ne garantit pas un fonctionnement sans risque de la détection radar. Ce qui n'est pas tolérable. Il est donc nécessaire de ne pas négliger ces pires cas, même si cela correspond à des situations peut probable.

Enfin, quelques modèles doivent prendre en considération deux changements majeurs, à savoir un GFD non réduit à une chaîne et des fonctions d'accès aux données affines (et non plus linéaires). Il reste encore quelques traitements préliminaires à appliquer pour trouver une solution de placement d'applications du type *mode radar MFR*, comme l'élimination des accès distincts à un même tableau dans un nid de boucles.

2 Prise en compte des spécificités du mode MFR

La thèse de Christophe Guettier ([32]) avait pour objectif principal de montrer la faisabilité de l'utilisation de la PPC pour résoudre le problème du placement d'applications de traitement de signal systématique, sur des architectures multi-processeurs homogènes. De ce fait, les applications traitées et les architectures considérées étaient simples et idéales[3].

Les différences entre le mode MFR et les applications traitées dans [32] (comme, par exemple, la **V**eille **B**ande **L**arge) sont remarquables. La VBL est une séquence de tâches alors que l'algorithme du mode MFR est un graphe[4]. Une autre différence est que les fonctions d'accès aux données deviennent affines et ne sont plus seulement linéaires. Par ailleurs, certaines tâches du mode MFR font des accès distincts à un même tableau. C'est-à-dire qu'un tableau possède plusieurs fonctions d'accès pour un même calcul. Il y a aussi l'apparition de traitements conditionnels dans les tâches élémentaires. Enfin, certains nids de boucles ont des bornes inférieures non nulles, ou encore des bornes paramétrées.

Cette section regroupe les différentes réflexions que nous avons pu avoir sur ces sujets. Ainsi nous verrons le passage d'une chaîne à un graphe de tâches, le problème d'une fonction d'accès aux données affine et, enfin, les pré-traitements effectués sur la spécification de l'application, permettant ainsi de retrouver un cadre hypothétique favorable.

2.1 D'une séquence de tâches vers un graphe de tâches acyclique

Dans [32], la première hypothèse posée concernant le graphe de tâches d'une application de traitement de signal systématique est qu'il soit acyclique. Cependant, afin de simplifier l'ensemble de la modélisation qui n'était alors pas encore définie, l'ensemble des graphes acycliques considérés a été réduit aux chaînes. Dès lors, une hypothèse forte est de dire que *deux tâches dépendantes sont consécutives*. Cette hypothèse intervient dans la linéarisation des nids de boucles partitionnés, l'ordonnancement, les communications, la mémoire ou encore certains pré-traitements tels que la fusion triviale de nids de boucles. Cependant, l'impact n'est pas direct pour tous les modèles évoqués à l'instant. Seuls l'ordonnancement (dans le contexte SIMD/SPMD) et la latence sont impliqués directement. On pourrait croire qu'il en va de même pour les dépendances, mais il suffit de changer le point de vue du problème. C'est-à-dire qu'il ne faut pas aborder le problème en parlant de *tâches dépendantes* mais de *tableaux sur lesquels s'exerce une dépendance*. La notion de graphe ou de chaîne disparaît alors et laisse place à un ensemble de points.

C'est pourquoi nous ne considérerons ici que les modèles d'ordonnancement et de latence. Nous proposons un principe permettant de supprimer la restriction aux séquences de tâches, à moindre coût pour le modèle d'ordonnancement établi. Puis, nous indiquons comment prendre en considération la structure du graphe pour calculer le chemin le plus long à l'aide de contraintes.

[3] Ce qui ne veut pas dire pour autant que les problèmes à résoudre l'étaient également, bien au contraire.

[4] Le mode MFR ne compte pas moins d'une trentaine de tâches contre *seulement* sept pour la VBL. Ce surplus de complexité permet également de tester la robustesse de l'approche contrainte.

2.1.1 Définitions

L'ensemble des tâches appartenant au graphe est divisé en trois sous-ensembles. Chacun des ces sous-ensembles est défini ci-dessous.

Définition 3.2-1 :
L'ensemble des tâches (ou nids de boucles) appartenant au graphe de tâches est noté NB.

Définition 3.2-2 : Tâche initiale
On dira d'une tâche $T \in \mathrm{NB}$ qu'elle est **initiale** si et seulement s'il n'existe pas de tâche dans le graphe de tâches T^w produisant des données consommées par T .

$$(T \text{ est initiale }) \Leftrightarrow (\nexists\, T^w \in \mathrm{NB}, \mathcal{D}(T^w, T))$$

L'ensemble des tâches initiales est noté NB^s et on a $\mathrm{NB}^s \subset \mathrm{NB}$.

Définition 3.2-3 : Tâche terminale
On dira d'une tâche $T \in \mathrm{NB}$ qu'elle est **terminale** si et seulement s'il n'existe pas de tâche T^r consommant des données produites par T dans le graphe de tâches.

$$(T \text{ est terminale }) \Leftrightarrow (\nexists\, T^r \in \mathrm{NB}, \mathcal{D}(T, T^r))$$

L'ensemble des tâches terminales est noté NB^e et on a $\mathrm{NB}^e \subset \mathrm{NB}$.

Définition 3.2-4 : Tâche intermédiaire
On dira d'une tâche $T \in \mathrm{NB}$ qu'elle est **intermédiaire** si et seulement si elle n'est ni initiale, ni terminale.

$$(T \text{ est intermédiaire }) \Leftrightarrow (T \in \mathrm{NB} \setminus \{\mathrm{NB}^e \cup \mathrm{NB}^s\})$$

Remarques : – De ces définitions nous déduisons[5] aisément que :

$$(\mathrm{card}(\mathrm{NB}) > 1) \Leftrightarrow (\mathrm{NB}^s \cap \mathrm{NB}^e = \emptyset)$$

– Dans le cas où le graphe de tâches se réduit à une et une seule tâche, on a :

$$\mathrm{NB} = \mathrm{NB}^s = \mathrm{NB}^e$$

2.1.2 D'une chaîne à un graphe : l'ordonnancement

La fonction d'ordonnancement d^T des blocs de calcul d'une tâche $T \in \mathrm{NB}$ est une fonction affine du paramètre temporel c^T issu du partitionnement 3D de l'espace d'itérations de la dite tâche (chap. 2, sec. 3). C'est-à-dire $\mathrm{d}^T\left(c^T\right) = \alpha^T.c^T + \beta^T$.

Rappelons les contraintes posée sur les variables α^T et β^T pour prendre en compte le phénomène SIMD/SPMD de la machine :

$$\begin{cases} \forall\, 0 \leqslant i < n-1 & \mathrm{card}\,(\mathrm{NB}) & \text{divise} & \alpha_i^T \\ & \mathrm{card}\,(\mathrm{NB}) & \text{divise} & \beta^T - k^T \end{cases} \tag{3.1}$$

[5]Rappelons que le graphe est orienté, sans circuit et connexe (pour tout couple de sommets du graphe non orienté induit, il existe un chemin allant de l'un à l'autre).

Ce qui peut se réécrire en considérant deux paramètres γ/δ tels que :

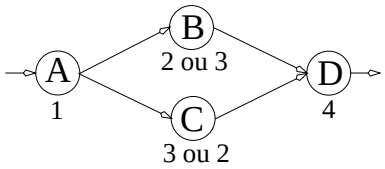
$$\begin{aligned} \alpha^T &= \text{card}\,(\text{NB})\,.\gamma^T \\ \beta^T &= \text{card}\,(\text{NB})\,.\delta^T + k^T \end{aligned}$$

k^T est le numéro attribué à la tâche T. Ainsi, deux blocs de calculs distincts ne peuvent s'exécuter simultanément (ce qui correspond à un ordonnancement SPMD/SIMD).

L'attribution d'un numéro à une tâche est arbitraire. Il s'agit la plupart du temps de l'étiqueter afin de la distinguer des autres. La seule contrainte est donc que ces étiquettes soient uniques, la façon dont on les pose importe peu. Cependant de manière générale et naturelle, cet étiquetage est dictée par la succession des tâches les unes par rapport aux autres en suivant le sens des arcs du graphe. On obtient alors une *liste topologique* des sommets du graphe, définissant un ordre partiel. Dans notre cas, cet étiquetage est une numérotation.

Suivant cette «règle naturelle», dans le cas où le graphe de tâches est une chaîne, la numérotation correspond exactement à la séquence. Ainsi le paramètre k^T est déterminé dès que l'on prend connaissance de la structure du GFD. Dans le cas d'un graphe acyclique quelconque, il arrive que deux chemins dans le graphe soient parallèles[6]. La décision de l'attribution d'un numéro à une tâche devient alors partiellement[7] arbitraire (cf. Fig. 3.2). Comme ce numéro k^T intervient dans la fonction d'ordonnancement, il n'est pas possible de décider de sa valeur sans favoriser du même coup la recherche vers une partie de l'espace des solutions. Il convient donc de rendre ce paramètre variable et de laisser le système décider de sa valeur. Sans pour autant lui donner trop de liberté sous peine de perdre l'intérêt de l'utilisation de ce numéro dans les fonctions d'ordonnancement.

FIG. 3.2 – Chemins parallèles dans un graphe : plusieurs possibilités de numérotation.



Mais prenons garde, car le fait de laisser le choix au système sans autres indications ne peut qu'augmenter la combinatoire. Il faut alors la couper au mieux. C'est dans cette optique que nous avons cherché des méthodes de numérotation inspirées de la théorie des graphes [10].

Avoir une stratégie de numérotation des tâches permet d'accélérer la recherche de solution dans la mesure où les valeurs sont contraintes et dans des intervalles réduits à quelques valeurs. De plus une bonne numérotation permettra d'aboutir rapidement à un bon ordonnancement.

[6]Nous dirons que deux chemins dans un graphe orienté sont parallèles s'ils partagent leur sommet de départ et/ou d'arrivée.

[7]La liste topologique établie, si l'on en tient compte, élimine de fait certaines valeurs.

2.1.3 Conditions sur la fonction de numérotation

Tout d'abord, la fonction de numérotation doit donner à toute tâche un numéro, et à chaque numéro est associée une unique tâche. Ainsi, si l'on note $\mathcal{N}$ cette fonction et $\mathfrak{D}_k$ le domaine de numérotation, on a :

$$\forall T \in \mathrm{NB}, \exists! k^T \in \mathfrak{D}_k, \quad \mathcal{N}(T) = k^T \tag{3.2}$$
$$\forall k^T \in \mathfrak{D}_k, \exists! T \in \mathrm{NB}, \quad \mathcal{N}^{-1}(k^T) = T \tag{3.3}$$

Il reste encore à définir $\mathfrak{D}_k$.

Le premier calcul de la première tâche[8] marque l'origine de l'axe temporel d'exécution. Il est, par conséquent, effectué à l'instant logique 0. La fonction d'ordonnancement étant $\alpha^T.c^T + \beta^T = \mathrm{card}\,(\mathrm{NB}) \times \left(\gamma^T.c^T + \delta^T\right) + k^T$, cela signifie que le numéro de la première tâche doit être 0. En effet, le premier calcul d'une tâche T a lieu au temps logique correspondant à $c^T = 0$, c'est-à-dire à $\beta^T = \mathrm{card}\,(\mathrm{NB}) \times \delta^T + k^T$. Si T est la première tâche, on a alors $\beta^T = \mathrm{card}\,(\mathrm{NB}) \times \delta^T + k^T = 0$. Or δ^T et k^T étant par définition des entiers positifs et $card\,(\mathrm{NB})$ étant strictement positif, on en déduit que :

$$\left(\mathrm{card}\,(\mathrm{NB}) \times \delta^T + k^T = 0\right) \Leftrightarrow \left\{ \begin{array}{rcl} \delta^T & = & 0 \\ k^T & = & 0 \end{array} \right.$$

Ainsi nous définissons l'intervalle de numérotation $\mathfrak{D}_k$:

$$\mathfrak{D}_k = [0, \mathrm{card}\,(\mathrm{NB}) - 1]$$

La fonction de numérotation est ainsi définie par :

$$\begin{array}{rccl} \mathcal{N} : & \mathrm{NB} & \longrightarrow & \mathfrak{D}_k = [0, \mathrm{card}\,(\mathrm{NB}) - 1] \\ & T & \longmapsto & k^T = \mathcal{N}(T) \end{array}$$

Cependant, nous n'avons toujours pas choisi **comment** numéroter les tâches du graphe.

2.1.4 Quel ordre choisir ?

Dans la suite nous présentons trois façons d'*ordonner* le graphe :

1. parcours de graphe (*en profondeur et en largeur*).
2. en fonction de la position respective des tâches dans le graphe (*tri topologique 1*) ;
3. et une version relâchée de la précédente (*tri topologique 2*) ;

2.1.4.1 Parcours de graphe *(en profondeur, en largeur)* La première idée que l'on a pour numéroter les tâches du graphe est de suivre un chemin et de numéroter les tâches qui ne le sont pas déjà. En fait, il s'agit des algorithmes de parcours de graphe traditionnels dits *en largeur* ou *en profondeur*. La figure 3.3.c illustre un parcours en profondeur d'un graphe.

[8] La *première tâche* est la tâche à qui appartient le bloc de calcul dont la date d'exécution est l'origine temporelle (logique ou physique). Pour les modes de programmations SIMD, SPMD et M-SPMD, cette tâche est unique.

Pour que ce soit le système qui décide de la numérotation et, surtout, pour qu'il soit capable de remettre en cause sa décision, il est nécessaire que l'algorithme de parcours soit exprimé en contraintes. Ce qui n'est pas possible.

De plus, le résultat obtenu après ce type de parcours n'est pas très intéressant et ne préfigure pas l'ordre donné par les précédences.

2.1.4.2 Selon la position d'une tâche dans le graphe (*tri topologique 1*) Le tri topologique des nœuds d'un graphe orienté sans circuit consiste à déterminer une extension linéaire d'un ordre partiel établi sur ces nœuds.

Dans le cadre de notre graphe de tâches, l'ordre partiel découle des précédences :

$$\forall (T^w, T^r) \in \mathrm{NB} \times \mathrm{NB}, / \ (\mathcal{D}(T^w, T^r)) \Rightarrow (\mathcal{N}(T^w) < \mathcal{N}(T^r)) \tag{3.4}$$

L'extension linéaire définissant le tri topologique est laissé au choix du système. Cependant, afin de limiter le choix des valeurs, nous définissons un intervalle de valeurs possibles pour chacune des tâches du graphe respectant l'ensemble des conditions présentées ci-dessus.

Appelons $\mathcal{A}^T$ l'ensemble des tâches placées dans le graphe avant la tâche T et $\mathcal{P}^T$ l'ensemble des tâches situées sur des chemins parallèles à celui sur lequel est T et n'étant pas dans $\mathcal{A}^T$. Le numéro de T sera alors dans l'intervalle $\left[\mathrm{card}(\mathcal{A}^T), \mathrm{card}(\mathcal{A}^T) + \mathrm{card}(\mathcal{P}^T)\right]$.

Dans l'exemple b de la figure 3.3, considérons le sommets F. On a alors $\mathcal{A}^F = [A, C, D]$ et $\mathcal{P}^F = [B, E]$. Ainsi l'intervalle des valeurs possibles pour numéroter F est

$\left[Card(\mathcal{A}^F), Card(\mathcal{A}^F) + Card(\mathcal{P}^F)\right] = [3, 5]$.

Si le sommet traité T se trouve sur un chemin pour lequel $\mathcal{P}^T = \emptyset$ alors son numéro est déterminé et correspond à sa position dans le graphe.

La difficulté réside ici dans la détermination automatique, pour tous les nœuds, de l'ensemble $\mathcal{P}$ associé.

2.1.4.3 Version relâchée (*tri topologique 2*) Comme précédemment, on garde l'ordre partiel imposé par les précédences.

Mais, afin d'augmenter l'entropie de la quantité d'information transmise, et du fait que nous avons défini trois classes de tâches dans le graphe, les tâches initiales, terminales et intermédiaires, nous associons à chacune d'elle des domaines de définition tels que :

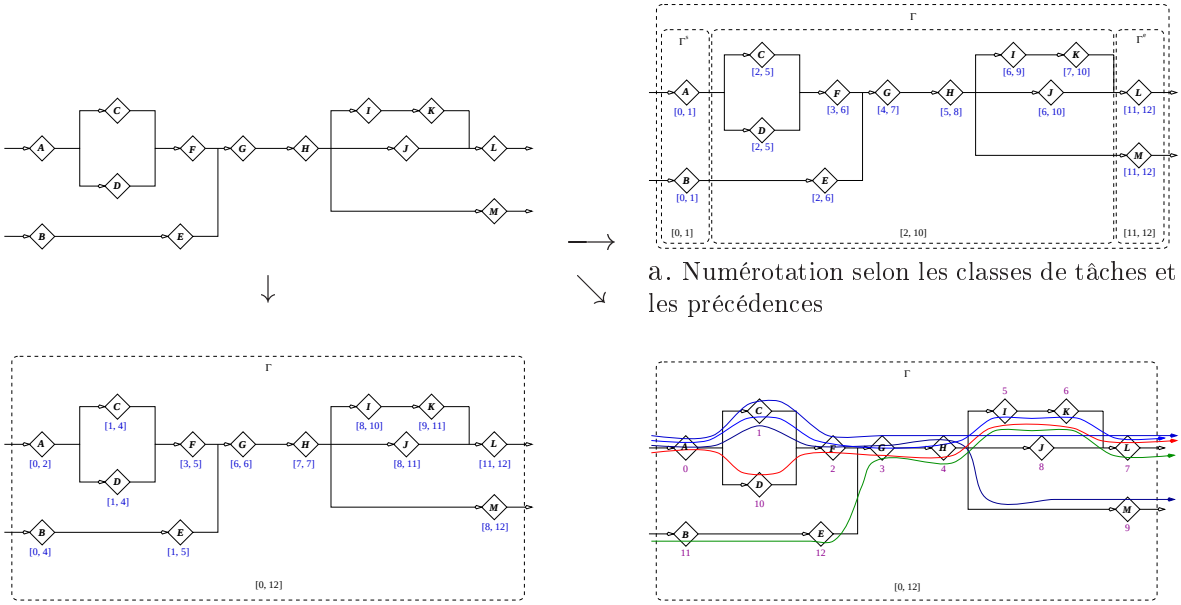
$$\forall T \in \mathrm{NB}^s, \quad k^T \in [0, \mathrm{card}(\mathrm{NB}^s) - 1] \tag{3.5}$$

$$\forall T \in \mathrm{NB}^e, \quad k^T \in [\mathrm{card}(\mathrm{NB}) - \mathrm{card}(\mathrm{NB}^e), \mathrm{card}(\mathrm{NB}) - 1] \tag{3.6}$$

$$\forall T \in \mathrm{NB}/(\mathrm{NB}^s \cup \mathrm{NB}^e), \quad k^T \in [\mathrm{card}(\mathrm{NB}^s), \mathrm{card}(\mathrm{NB}) - \mathrm{card}(\mathrm{NB}^e) - 1] \tag{3.7}$$

Considérons le graphe acyclique représenté par la figure 3.3. On a

$$\begin{aligned} \mathrm{NB} &= \{A, B, C, D, E, F, G, H, I, J, K, L, M\} \\ \mathrm{NB}^s &= \{A, B\} \\ \mathrm{NB}^e &= \{L, M\} \end{aligned}$$



a. Numérotation selon les classes de tâches et les précédences

b. Numérotation selon la positions des tâches dans le graphes et les précédences

c. Numérotation sur un chemin : parcours en profondeur

FIG. 3.3 – Effet des différentes méthodes de numérotation sur un exemple.

Ainsi, par cette méthode, la tâche A a comme domaine de numérotation $[0, 1]$, tout comme la tâche B. Les tâches L et M se verront attribuer, quant à elles, un numéro appartenant à l'intervalle $[11, 12]$. Les intervalles de définition de la numérotation des autres tâches est déduit des précédences. On obtient alors le résultat de la figure 3.3.a après une propagation initiale. La détermination complète des numéros attribués aux nœuds se fera pendant la résolution. Ainsi, le système a toujours la possibilité de revenir sur ses décisions.

2.1.4.4 Le choix Pour des raisons d'implémentation, nous avons choisi cette dernière méthode. Elle présente tout de même un inconvénient. Elle suppose que les tâches initiales (resp. terminales) sont situées plutôt en début (resp. fin) de graphe. Aussi, les tâches initiales (resp. terminales) auront un petit (resp. grand) numéro.

Les conséquences de cet inconvénient se retrouvent dans les ordonnancements en étirant, dans l'espace de temps logique, les dates d'exécution des différentes tâches. En effet, la date de première exécution est donnée par le paramètre β. Ce dernier étant défini par $\beta = \text{card(NB)}.\delta + k$, une tâche initiale (avec un petit k) située plutôt en fin de graphe (qui sera donc exécutée plutôt tardivement) aura une date de première exécution très éloignée des autres du fait de la compensation du paramètre δ sur k.

Illustrons cette situation à l'aide d'un exemple. Considérons l'ordonnancement $ADCFBE$ des 6 premières tâches de notre graphe. Pour obtenir cet ordonnancement, la méthode 2 donnera comme première solution :

	A	B	C	D	E	F
k	0	4	2	1	5	3
δ	0	0	0	0	0	0
β	0	4	2	1	5	3

alors que la méthode 3, pour obtenir le même ordre d'exécution, donnera plutôt :

	A	B	C	D	E	F
k	0	1	3	2	5	4
δ	0	1	0	0	1	0
β	0	13	3	2	17	4

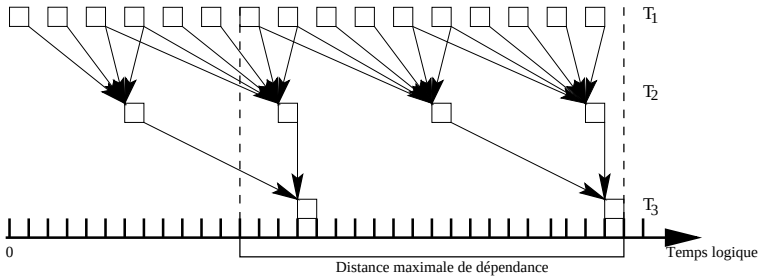
Comme il ne s'agit que de temps logiques, la dilatation de cet axe temporel n'a pas d'impact sur le reste (calcul de latence...) puisqu'il n'est question que de définir un ordre d'exécution. Les temps logiques de **NOP** sont éliminés dès que l'on s'intéresse à des durées ou à la génération de code de contrôle.

En fait, l'influence de telle ou telle méthode de numérotation des tâches du graphe est très faible sur la qualité de la solution. En effet, il est toujours possible de trouver des valeurs pour les coefficients δ telles que l'on obtienne des ordonnancements équivalents d'une méthode à l'autre (quitte à introduire des événements de **NOP**). Ce qui est important est le nombre d'événements actifs qui apparaissent sur une période et non pas sa longueur.

2.1.5 La latence

La latence est évaluée à partir de la longueur du chemin *critique* issu du calcul des dépendances. Dans le cas où le graphe est une chaîne, la longueur de ce chemin critique correspond à la somme des liens de dépendance les plus longs entre deux tâches (fig. 3.4), dans un contexte de modèle SIMD ou SPMD.

FIG. 3.4 – Évaluation du chemin critique dans une chaîne.



Dans le cas d'un graphe autre, mais toujours dans un modèle SIMD ou SPMD, il n'est pas question de calculer cette somme pour déterminer le chemin de coût maximum. Le résultat serait une sur-approximation de ce coût, puisque l'on ajouterait les coûts des chemin parallèles alors qu'il s'agit d'un maximum.

Soit $G = (S, A)$ le graphe orienté sans circuit, représentant le graphe de précédences d'une application de traitement du signal. S est l'ensemble des sommets de G (les tâches) et A l'ensemble des arcs (relations de dépendance) de G.

Chaque arc (i,j) liant deux sommets (de i vers j) est valué d'un coût $a_{i,j}$. Il représente la plus longue distance événementielle de dépendance entre i et j. Elle est obtenue à partir des sommets générateurs du polytope de dépendance entre ces deux nœuds.

$$\forall i, \forall j \text{ t.q. } \mathcal{D}(i,j), \quad a_{i,j} = \max_{s \in \left\{ \begin{array}{c} \text{sommets du polytope} \\ \text{de dépendance} \end{array} \right\}} (d^j(c_s^j) - d^i(c_s^i)) \tag{3.8}$$

À chaque sommet i, nous associons un poids n_i, qui est la longueur du chemin critique du sous graphe issu du nœud i. Les sommets représentant des tâches terminales ont donc un poids nul.

$$\forall i \in \mathrm{NB} \setminus \mathrm{NB}^e, \quad n_i = \max_{\{j | j \in \mathrm{succ}(i)\}} (a_{i,j} + n_j) \tag{3.9}$$

$$\forall i \in \mathrm{NB}^e, \quad n_i = 0 \tag{3.10}$$

La longueur événementielle du chemin critique du graphe est le maximum des poids des racines (*i.e.*, les tâches initiales) incrémentés de leur date de première exécution respective (*i.e.*, les β). La première exécution étant à la date 0, il faut ajouter 1.

$$\Delta = \max_{i \in \mathrm{NB}^s} (\beta^i + n_i) + 1 \tag{3.11}$$

De ces formules, nous dérivons directement les contraintes permettant de calculer la longueur du chemin critique dans un graphe sans circuit.

Notons que lorsque le graphe est une chaîne, nous retrouvons la formule 2.16 de la page 47. En effet, lorsque le graphe est une séquence de calcul, on a alors :

$$\begin{aligned} \Delta &= n_0 + 1 \\ &= a_{0,1} + n_1 + 1 \\ &= 1 + \sum_{i=0}^{n-2} a_{i,i+1} \end{aligned}$$

2.1.6 Conclusion

Ainsi, grâce à ces règles de numérotation des tâches, il est possible de modéliser toutes les applications dont le graphe de tâches est un graphe acyclique quelconque.

De plus, le seul ordre imposé est celui donné par le graphe de précédence. Ce qui laisse le système libre de choisir les valeurs adéquates des coefficients, sans pour autant augmenter d'un facteur exponentiel la combinatoire de l'ensemble.

Par ailleurs, nous avons généralisé le calcul du coût du chemin critique dans le graphe partitionné aux graphes acycliques quelconques, à partir duquel nous calculons la latence de l'application placée.

2.2 Une fonction d'accès affine

La fonction d'accès aux données d'un tableau considérée dans [32] est une application du type $(\Omega_{pav}.i + \Omega_{fit}.m) \bmod \Phi$ (chapitre 2, sec. 3).

La question est d'étudier l'impact du passage d'une fonction d'accès linéaire à une fonction d'accès affine sur le modèle dépendances, modèle principalement impliqué dans leur utilisation. Le modèle mémoire est un modèle capacitif. L'ajout de ce paramètre constant ne modifie pas la taille du motif mais effectue une translation de ce dernier dans le tableau. Par conséquent, il n'est pas touché par cette extension. Nous considérerons désormais une fonction d'accès du type

$$(\Omega_{pav}.i + \Omega_{fit}.m + \mathcal{K}) \bmod \Phi$$

$\mathcal{K}$ est un vecteur de constantes de $\mathbb{Z}$.

L'intérêt est de pouvoir représenter ce type d'accès et prendre en compte des boucles dont la borne inférieure n'est pas zéro, en effectuant une translation de l'espace d'itérations correspondant.

Nous commençons par l'impact que peut avoir cette translation $\mathcal{K}$ sur les dépendances. C'est le modèle le plus concerné puisqu'il est nécessaire de connaître précisément la fonction d'accès aux données pour déterminer les dépendances entre nids de boucles.

Nous voyons ensuite l'effet du modulo sur la fonction d'accès et les dépendances.

2.2.1 Impact sur les dépendances

Nous reprenons ici la méthode utilisée dans [32] pour calculer les dépendances en intention. Nous avons ajouté les paramètres affines $\mathcal{K}$ dans les fonctions d'accès.

2.2.1.1 Définition d'une dépendance entre deux nids de boucles Soient deux nids de boucles N^w et N^r dépendant l'un de l'autre. N^w produit des données consommées par N^r. Appelons T le tableau sur lequel repose cette dépendance. Soient $\begin{pmatrix} i^w \\ m^w \end{pmatrix}$ et $\begin{pmatrix} i^r \\ m^r \end{pmatrix}$ les vecteurs d'itérations de chacun des deux nids de boucles, $\begin{pmatrix} \Omega^w_{pav} & \Omega^w_{fit} \end{pmatrix}$ et $\begin{pmatrix} \Omega^r_{pav} & \Omega^r_{fit} \end{pmatrix}$ les matrices d'accès associées, et $\mathcal{K}^w$ et $\mathcal{K}^r$ deux vecteurs de constantes. On définit φ^w et φ^r les vecteurs de références aux données de T tels que $\forall j \in \{w, r\} \quad \varphi^j = \Omega^j_{pav}.i^j + \Omega^j_{fit}.m^j + \mathcal{K}^j$

Par définition du partitionnement 3D, nous avons $\forall j \in \{w, r\} \quad i^j = P^j.L^j.c^j + L^j.p^j + l^j$ pour un partitionnement bloc/cycle. Ainsi, l'ensemble des éléments du tableau T référencés par une partition temporelle c^j du nid de boucles j, $\mathfrak{P}_{c^j}$ est :

$$\mathcal{R}\left(\mathfrak{P}_{c^j}\right) = \Big\{ \varphi^j \mid \varphi^j \in \Phi^j, \quad \exists (l^j, p^j), \quad 0 \leqslant L^{j^{-1}}.l^j < 1, \quad 0 \leqslant P^{j^{-1}}.p^j < 1 \quad \exists m^j \in \mathcal{M}^j, \\ \varphi^j = \Omega^j_{pav}\left(P^j.L^j.c^j + L^j.p^j + l^j\right) + \Omega^j_{fit}.m^j + \mathcal{K}^j \Big\}$$

En posant $s^j = L^j.p^j + l^j$, on a une vue plus simple de la distribution temporelle des données [32, Ch. 4]. L'ensemble précédent devient :

$$\mathcal{R}\left(\mathfrak{P}_{c^j}\right) = \Big\{ \varphi^j \mid \varphi^j \in \Phi^j, \quad \exists s^j, \quad 0 \leqslant \left(L^j.P^j\right)^{-1}.s^j < 1, \quad \exists m^j \in \mathcal{M}^j, \\ \varphi^j = \Omega^j_{pav}\left(P^j.L^j.c^j + s^j\right) + \Omega^j_{fit}.m^j + \mathcal{K}^j \Big\}$$

Il existe une dépendance écriture / lecture entre N^w et N^r pour un couple (c^w, c^r) si et seulement si $\mathcal{R}\left(\mathfrak{P}_{c^w}\right) \cap \mathcal{R}\left(\mathfrak{P}_{c^r}\right) \neq \emptyset$ [32, Chap. 5]

C'est-à-dire :

$$\mathcal{D}(c^w, c^r) \Rightarrow \begin{cases} \exists (m^w, m^r) \in \mathcal{M}^w \times \mathcal{M}^r \\ \exists s^w, \quad 0 \leqslant (L^w.P^w)^{-1}.s^w < 1 \\ \exists s^r, \quad 0 \leqslant (L^r.P^r)^{-1}.s^r < 1 \\ \Omega^w_{pav}.(P^w.L^w.c^w + s^w) + \Omega^w_{fit}.m^w + \mathcal{K}^w = \Omega^r_{pav}.(P^r.L^r.c^r + s^r) + \Omega^r_{fit}.m^r + \mathcal{K}^r \end{cases}$$

Les fonctions d'accès étant linéairement indépendantes et les matrices de partitionnement étant diagonales, on peut aborder le problème dimension de tableau par dimension de tableau. De ce fait, l'espace décrivant les dépendances pour chaque dimension q de T est un polygone caractérisé par le système suivant (pour simplifier la notation, l'indice q dénotant les dimensions a été omis sur chacun des termes) :

$$\begin{cases} 0 \leqslant c^w \leqslant c^w_{max} - 1 \\ 0 \leqslant c^r \leqslant c^r_{max} - 1 \\ 0 \leqslant m^w \leqslant m^w_{max} - 1 \\ 0 \leqslant m^r \leqslant m^r_{max} - 1 \\ 0 \leqslant s^w \leqslant L^w.P^w - 1 \\ 0 \leqslant s^r \leqslant L^r.P^r - 1 \\ \Omega^w_{pav}.(P^w.L^w.c^w + s^w) + \Omega^w_{fit}.m^w + \mathcal{K}^w = \Omega^r_{pav}.(P^r.L^r.c^r + s^r) + \Omega^r_{fit}.m^r + \mathcal{K}^r \end{cases}$$

L'élimination des variables s^j et m^j, par projection, engendre le système de contraintes suivant :

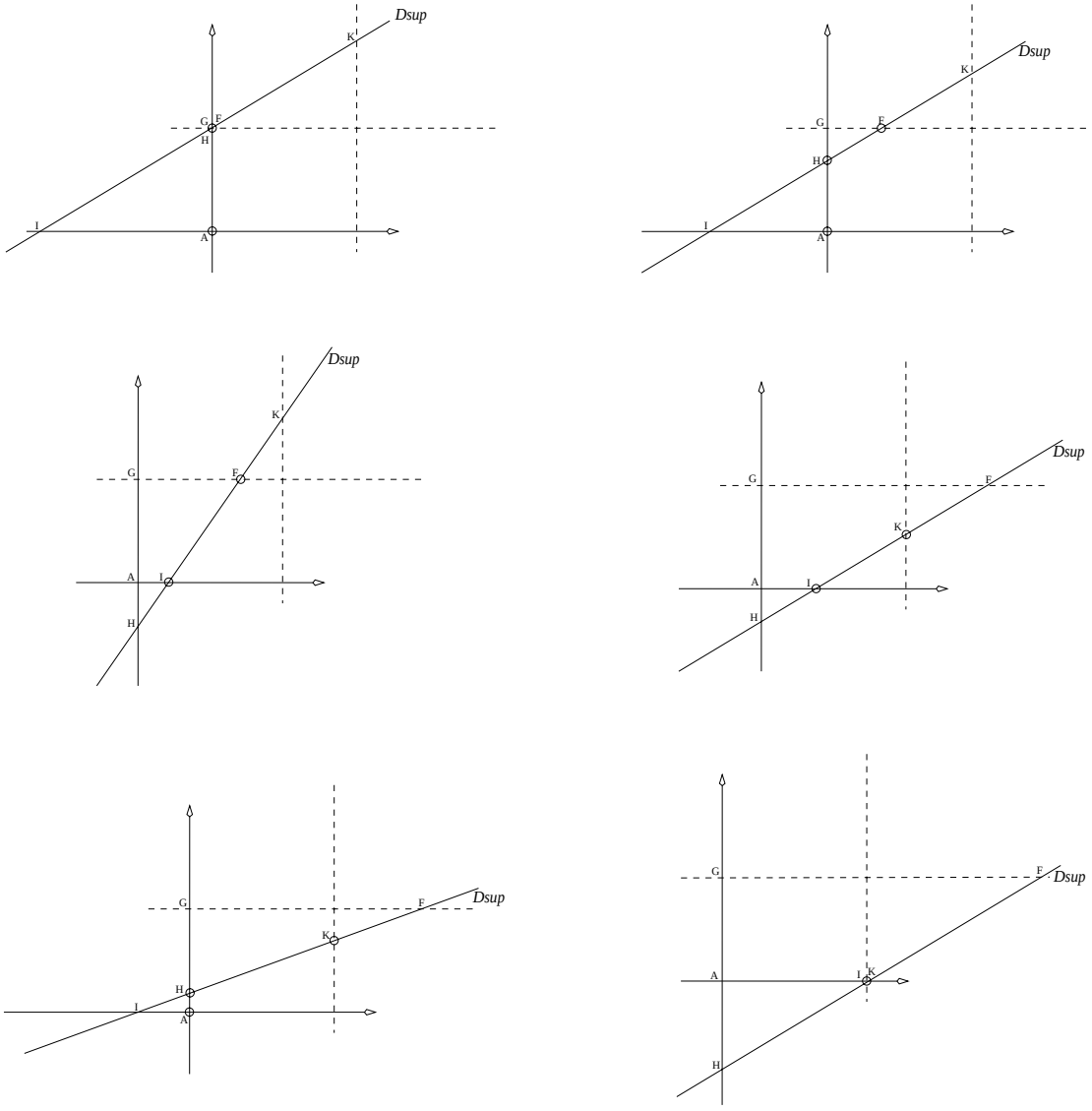
$$\begin{cases} 0 \leqslant c^w \leqslant c^w_{max} - 1 \\ 0 \leqslant c^r \leqslant c^r_{max} - 1 \\ -\Omega^r_{pav}L^rP^r.c^r + \Omega^w_{pav}L^wP^w.c^w \geqslant -\Omega^w_{fit}(m^w_{max} - 1) - \Omega^w_{pav}(L^wP^w - 1) + \mathcal{K}^r - \mathcal{K}^w & (D_{inf}) \\ -\Omega^r_{pav}L^rP^r.c^r + \Omega^w_{pav}L^wP^w.c^w \leqslant \Omega^r_{fit}(m^r_{max} - 1) + \Omega^r_{pav}(L^rP^r - 1) + \mathcal{K}^r - \mathcal{K}^w & (D_{sup}) \end{cases}$$

L'espace de dépendance, dimension par dimension, est donc délimité par les droites d'équations $c^j = cst$, D_{inf} et D_{sup}. Il est cependant à noter que cet espace ainsi délimité est une sur-approximation de l'espace de dépendance réel. En effet, les différentes projections et éliminations de variables encadrent par un ensemble convexe l'ensemble des points de dépendances. Pour illustrer, l'approximation introduite ici correspond à celle que l'on fait lorsque l'on remplace l'ensemble $\{2, 4, 6, 8\}$ par l'intervalle $[2, 8]$. Tous les points contenus dans le polygone convexe ne sont pas nécessairement des points de dépendances.

2.2.1.2 Effet du paramètre constant sur le polygone des dépendances

La présence de cet offset $\mathcal{K}^r - \mathcal{K}^w$ donne une plus grande liberté quant à la position des droites D_{inf} et D_{sup}. En effet, si l'on considère D_{sup} (resp. D_{inf}), $\Omega^r_{fit}(m^r_{max} - 1) + \Omega^r_{pav}(L^rP^r - 1)$ est toujours positif ou nul (resp. $-\Omega^w_{fit}(m^w_{max} - 1) - \Omega^w_{pav}(L^wP^w - 1)$ est toujours négatif ou nul), par définition des différents coefficients intervenant dans ces termes. Il est possible d'obtenir un partitionnement tel que $\mathcal{K}^r - \mathcal{K}^w$ change le signe du membre droit des équations des droites. La figure 3.5 illustre les différents cas possibles d'intersection de la droite D_{sup} avec l'espace d'itérations partitionné. On obtient la même chose avec la droite D_{inf} sachant que l'on a toujours D_{sup} "au dessus" de D_{inf}.

FIG. 3.5 – Énumération des différentes positions possible pour la droite D_{sup}.



Les différentes possibilités de positionnement de la droite D_{inf} sont les mêmes que celles de la droites D_{sup}. Aussi, il y a $6 \times 6 = 36$ polygones de dépendances probables alors qu'il n'y en a que $3 \times 3 = 9$ si l'on ne prend pas en compte les offsets dans les fonctions d'accès. En effet, sans les offsets, les points I et J ont des coordonnées toujours négatives (ou nulles) ce qui n'est plus le cas selon les valeurs de ces décalages.

Par ailleurs, la relaxation dans les entiers des différents points d'intersections (définis dans $\mathbb{Q}$) oblige à sur-approximer le polygone de dépendance par la plus petite enveloppe convexe dont les sommets générateurs sont à coordonnées entières[9]. C'est-à-dire :

$$\begin{cases} x_A &= 0 \\ y_A &= 0 \end{cases} \qquad \begin{cases} x_E &= c^r_{max} - 1 \\ y_E &= c^w_{max} - 1 \end{cases}$$

$$\begin{cases} x_C &= c^r_{max} - 1 \\ y_C &= 0 \end{cases} \qquad \begin{cases} x_G &= 0 \\ y_G &= c^w_{max} - 1 \end{cases}$$

$$\begin{cases} x_I &= \left\lfloor \frac{-\Omega^r_{fit}(m^r_{max}-1)-\Omega^r_{pav}(L^rP^r-1)+\mathcal{K}^w-\mathcal{K}^r}{\Omega^r_{pav}L^rP^r} \right\rfloor \\ y_I &= 0 \end{cases} \qquad \begin{cases} x_L &= \left\lceil \frac{\Omega^w_{fit}(m^w_{max}-1)+\Omega^w_{pav}(L^wP^w.c^w_{max}-1)-\mathcal{K}^r+\mathcal{K}^w}{\Omega^r_{pav}L^rP^r} \right\rceil \\ y_L &= c^w_{max} - 1 \end{cases}$$

$$\begin{cases} x_J &= 0 \\ y_J &= \left\lfloor \frac{-\Omega^w_{fit}(m^w_{max}-1)-\Omega^w_{pav}(L^wP^w-1)+\mathcal{K}^r-\mathcal{K}^w}{\Omega^w_{pav}L^wP^w} \right\rfloor \end{cases} \qquad \begin{cases} x_K &= c^r_{max} - 1 \\ y_K &= \left\lceil \frac{\Omega^r_{fit}(m^r_{max}-1)+\Omega^r_{pav}(L^rP^r.c^r_{max}-1)+\mathcal{K}^r-\mathcal{K}^w}{\Omega^w_{pav}L^wP^w} \right\rceil \end{cases}$$

$$\begin{cases} x_B &= \left\lceil \frac{\Omega^w_{fit}(m^w_{max}-1)+\Omega^w_{pav}(L^wP^w-1)-\mathcal{K}^r+\mathcal{K}^w}{\Omega^r_{pav}L^rP^r} \right\rceil \\ y_B &= 0 \end{cases} \qquad \begin{cases} x_H &= 0 \\ y_H &= \left\lceil \frac{\Omega^r_{fit}(m^r_{max}-1)+\Omega^r_{pav}(L^rP^r-1)-\mathcal{K}^w+\mathcal{K}^r}{\Omega^w_{pav}L^wP^w} \right\rceil \end{cases}$$

$$\begin{cases} x_F &= \left\lfloor \frac{-\Omega^r_{fit}(m^r_{max}-1)-\Omega^r_{pav}(L^rP^r-1)+\Omega^w_{pav}L^wP^w(c^w_{max}-1)+\mathcal{K}^w-\mathcal{K}^r}{\Omega^r_{pav}L^rP^r} \right\rfloor \\ y_F &= c^w_{max} - 1 \end{cases}$$

$$\begin{cases} x_D &= c^r_{max} - 1 \\ y_D &= \left\lfloor \frac{-\Omega^w_{fit}(m^w_{max}-1)-\Omega^w_{pav}(L^wP^w-1)+\Omega^r_{pav}L^rP^r(c^r_{max}-1)+\mathcal{K}^r-\mathcal{K}^w}{\Omega^w_{pav}L^wP^w} \right\rfloor \end{cases}$$

La suppression des restrictions sur les points I et J, pouvant désormais appartenir au quart de plan $x \geqslant 0$, $y \geqslant 0$, confère une propriété de symétrie au polygone de dépendances. Plus précisément, il s'agit d'une symétrie de *configuration* dans laquelle on a les relations suivantes :

$$\begin{array}{lcl|lcl} A & \leftrightarrow & E & D & \leftrightarrow & H \\ B & \leftrightarrow & F & I & \leftrightarrow & L \\ C & \leftrightarrow & G & J & \leftrightarrow & K \end{array}$$

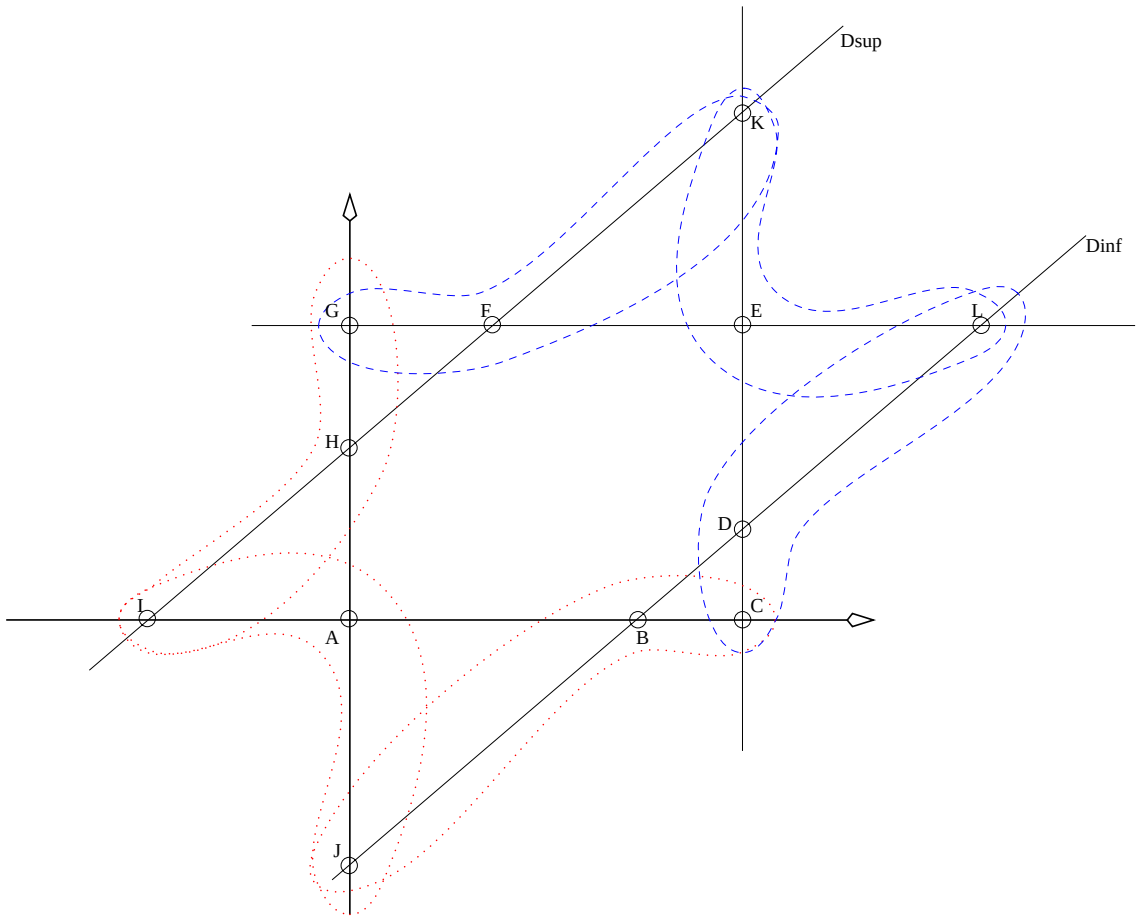
Selon la configuration obtenue, les sommets générateurs de l'enveloppe convexe seront tels ou tels points d'intersection listés ci-dessus. De ce fait, les sommets générateurs du polygone de dépendance sont (comme illustrés par la figure 3.6) :

$$\begin{array}{lll} AIJ & = (\max(\min(x_I, x_C), x_A) & , \quad \max(\min(y_J, y_G), y_A)) \\ BCJ & = (\max(\min(x_B, x_C), x_A) & , \quad \max(\min(y_J, y_G), y_A)) \\ CDL & = (\max(\min(x_L, x_C), x_A) & , \quad \max(\min(y_D, y_G), y_A)) \\ EKL & = (\max(\min(x_L, x_C), x_A) & , \quad \max(\min(y_K, y_G), y_A)) \\ FGK & = (\max(\min(x_F, x_C), x_A) & , \quad \max(\min(y_K, y_G), y_A)) \\ GHI & = (\max(\min(x_I, x_C), x_A) & , \quad \max(\min(y_H, y_G), y_A)) \end{array}$$

L'écriture AIJ signifie que, selon le cas, ce sommet est confondu avec A, I ou J.

[9] L'élimination de cette deuxième sur-approximation fait l'objet de la section 2 du chapitre 5.

FIG. 3.6 – Regroupement des points déterminant les sommets du polygone.



L'introduction d'un paramètre affine dans une fonctions d'accès n'a d'autres impacts que la redéfinition des coordonnées des sommets générateurs du polytope de dépendance.

2.2.2 La fonction *modulo* dans la fonction d'accès

Certaines fonctions d'accès font apparaître un modulo. Que se passe-t-il alors ?

Remarque : De part l'hypothèse de tableaux à assignation unique, les seuls accès *modulo* autorisés sont des accès en lecture.

La figure 3.7 page 76 montre un exemple d'accès avec modulo. Dans cet exemple, le tableau monodimensionnel accédé en écriture et en lecture est noté T. Sa taille Φ est de 15 mots. La fonction d'accès en écriture est $3.i^w : 3.i^w + 2$ avec $i^w \in [0, 4]$. Celle en lecture est : $(4.i^r : 4.i^r + 3) \bmod \Phi$ avec $i^r \in [0, 14]$.

En supposant des partitionnements tels que pour chaque espace d'itération on a $P^w = P^r = 1$, $L^w = L^r = 1$ et $c^w_{max} = i^w_{max} = 5$, $\quad c^r_{max} = i^r_{max} = 15$, on obtient les équations des droites D_{inf} et D_{sup} suivantes :

$$\begin{aligned} D_{inf} :&\ \ 3.c^w - 4.c^r = -2 \\ D_{sup} :&\ \ 3.c^w - 4.c^r = 3 \end{aligned}$$

Sur la figure, les points noirs représentent les itérations sur lesquelles il y a vraiment une dépendance de données. Les points blancs représentent les points de dépendances que l'on aurait s'il n'y avait pas de modulo.

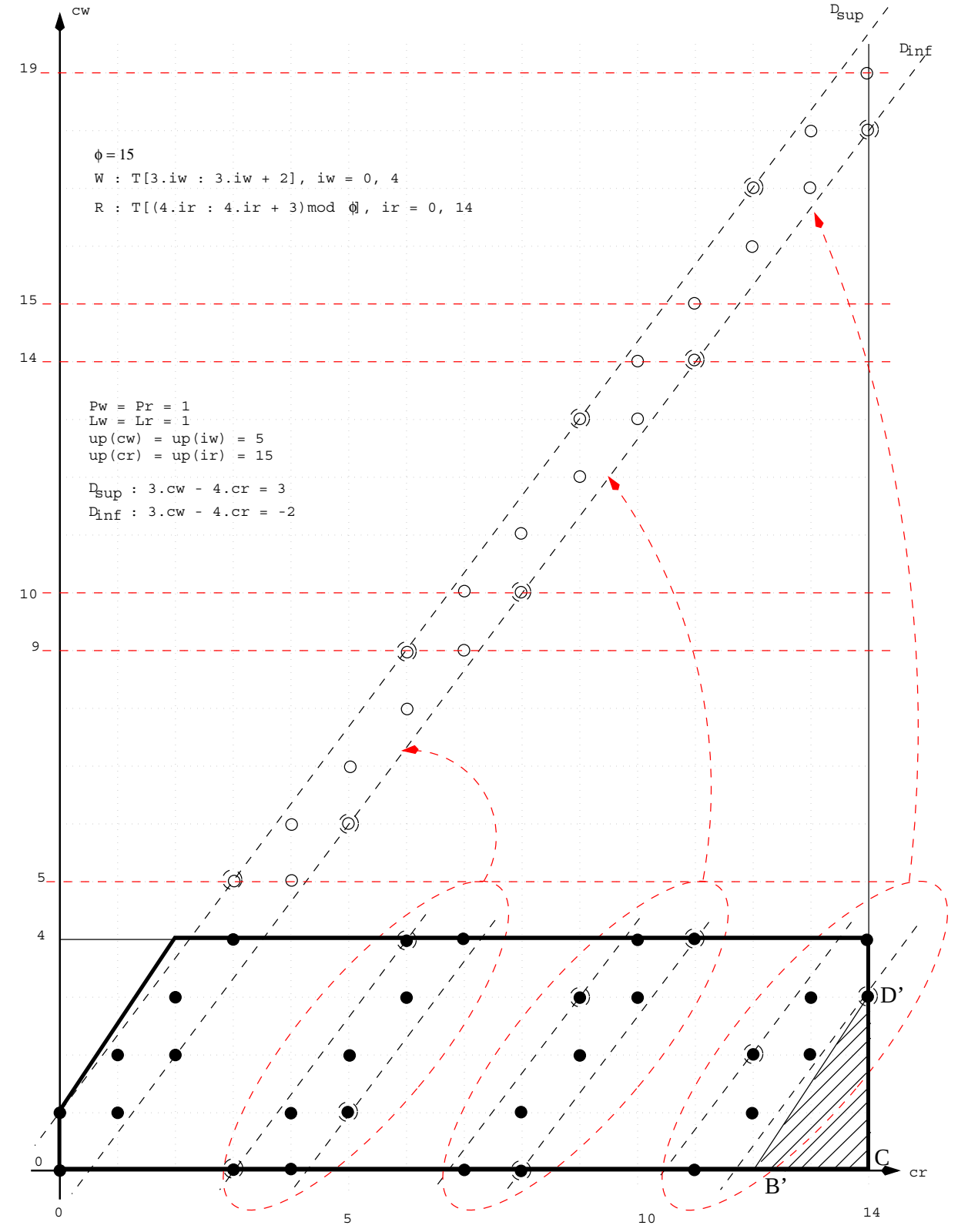
On constate alors que les *bandes* de dépendances correspondent à des morceaux de la bande virtuelle. Plus précisément, elles sont les morceaux issus de l'application du modulo.

Selon le partitionnement, les équations des droites délimitant le polygone des dépendances changent. Aussi, s'il est possible d'obtenir une représentation formelle de chacune de ces bandes, il est difficile, voire impossible, d'en extraire une modélisation en contraintes[10]. C'est pourquoi la solution adoptée est de sur-approximer cet espace de dépendances par la plus petite enveloppe convexe recouvrant l'ensemble de ces sous polygones. Dans notre exemple, cette sur-approximation est matérialisée sur la figure par un polygone en trait gras.

De plus, chercher à représenter exactement l'espace de dépendance dans le cas d'un modulo n'est pas intéressant, au contraire. Le polygone de dépendance représente l'espace des itérations en écriture et en lecture dépendantes les unes des autres pour l'accès à un tableau donné. Schématiquement, on peut traduire un modulo comme définissant les accès à des données, ou blocs de données, non contiguës d'un tableau (par exemple en début et en fin du tableau). Les itérations effectuant les opérations en écriture sont consécutives et s'exécutent dans l'ordre croissant de leurs itérations. Aussi, si pour une itération r, effectuant une opération de lecture sur un tableau, les dépendances ne font apparaître que la première et la dernière itérations du calcul effectuant l'opération d'écriture dans ce tableau, w_1 et w_2, cela ne signifie pas que les autres itérations w_i (intermédiaires) n'ont pas été exécutées. De ce fait, les données produites au cours de ces itérations w_i sont stockées en mémoire attendant d'être utilisées, mais par une itération r' différente de r. De ce point de vue,

[10] Une modélisation mathématique est tout du moins possible, mais elle fait apparaître une énumération trop importante de points définis dynamiquement selon le partitionnement.

FIG. 3.7 – Effet d'un modulo dans la fonction d'accès



l'approximation donne un meilleur résultat qu'une représentation exacte puisqu'elle prend en compte ce phénomène.

Nous pouvons croire qu'il est tout de même possible d'améliorer la sur-approximation. En effet, celle effectuée dans [32] n'est pas exactement la plus petite enveloppe convexe recouvrant l'ensemble des dépendances. Pour reprendre notre exemple, lorsque l'itération en lecture $c^r = 11$ est terminée, il est possible de libérer l'espace mémoire occupé par les données produites à $c^w = 0$ car elles ne seront plus utilisées par la suite. L'espace occupé par ces dépendances superflues correspond à la partie hachurée de la figure 3.7 (le triangle B'CD'). Les points B' et D' sont définis par :

$$B' = \left(\left\lceil \frac{-a.x_{D'} + b.y_{D'}}{-a} \right\rceil , 0 \right) \text{ et } D' = (c^r_{max} - 1, y_D \bmod c^w_{max})$$

Ainsi, on définit deux sommets B'C et CD' tels que :

$$\begin{aligned} B'C = &\ (\min(x_{B'}, c^r_{max} - 1), 0) \\ D'C = &\ (c^r_{max} - 1, \max(0, y_{D'})) \end{aligned}$$

Or y_D et c^w_{max} sont deux variables de domaines, liées par un modulo. Cette contrainte n'existe pas. Il est possible de la simuler en utilisant un produit ($\mathrm{r} = y_D \bmod c^w_{max} \Leftrightarrow \exists \mathrm{q}, \quad y_D = \mathrm{q} \times c^w_{max} + \mathrm{r}$). La propagation serait alors totalement inefficace, et de ce fait, l'intérêt est dérisoire.

2.2.3 Conclusion

La prise en compte des fonctions d'accès affines (avec ou sans modulo) ne modifie pas la modélisation des dépendances. Seules les expressions formelles des coordonnées des sommets générateurs des polytopes de dépendance sont modifiées.

2.3 Les pré-traitements

Cette section décrit succinctement les différentes manipulations effectuées sur l'ensemble des nids de boucles (*i.e.*, l'application) permettant ainsi de retrouver un contexte compatible avec notre méthode de placement.

2.3.1 Accès distincts à un même tableau dans une TE

Les accès multiples à un même tableau peuvent être pris en compte :

1. en augmentant le modèle de dépendance ;
2. en ajoutant une opération de copie par référence ;
3. en ajoutant une boucle interne, i.e. un motif, pour regrouper par fermeture hyper-rectangulaire toutes les références.

L'idéal est de reconsidérer la modélisation des dépendances pour prendre en compte ces situations. Cependant, par rapport à ce que représente ce problème sur l'ensemble de l'étude, le gain d'une remise en question du modèle est trop faible.

Faire une copie du tableau ainsi accédé n'est pas une bonne solution non plus car cela fausse le calcul des dépendances et augmente la consommation mémoire.

Ce dont nous avons besoin sont les informations d'accès à un instant donné. C'est-à-dire la taille du motif utilisé par la TE. C'est pourquoi la solution retenue est la troisième. Du point de vue de l'algorithme concerné, cette méthode change sa sémantique. Cependant, ce n'est pas important car l'information qui nous intéresse n'est pas la nature du calcul, mais plutôt quels sont ces besoins en termes de données, de puissance de calcul, de durée.

2.3.2 Conditionnement des tâches élémentaires

De manière générale, la prise en compte de paramètres variables dans la description de l'application à placer ne rentre pas dans le cadre de cette thèse. C'est pourquoi nous avons choisi une instance du mode MFR où chaque paramètre a une valeur fixe.

2.3.2.1 TE conditionnée par un paramètre constant lors d'un pointage Nous considérons autant d'instance de l'application que le paramètre a de valeurs possibles. Pour le mode MFR, nous avons choisi une situation de pire cas.

2.3.2.2 TE conditionnée par un paramètre variable Dans notre application MFR, le paramètre variable est une donnée. Cela signifie que, les résultats des tests ne pouvant être connus à l'avance, il n'est possible de savoir quelles sont les itérations qui donneront lieu à un calcul ou quel calcul sera effectué. Cette incertitude concerne directement les trois modèles suivants (les autres sont touchés par effet de bord) :

la latence : selon le résultat du test, tel ou tel calcul est exécuté (ou pas selon que le test présente une alternative) ;

la mémoire : selon le calcul effectué, les besoins en mémoire sont différents (les fonctions d'accès aux données d'un tableau sont différentes, l'ensemble des tableaux utilisés est différent, le nombre de données produites,...) ;

les dépendances : découlent des deux décisions précédentes. Une fois qu'elles sont prises, nous retrouvons un contexte statique déterministe.

Pour évaluer la latence de l'application, nous avons trois possibilités.

La première consiste à déterminer exactement le nombre de calculs effectués et ainsi obtenir une durée. Il est évident que ce n'est pas possible puisqu'il faut savoir quelles sont les données qui engendreront un calcul et celles qui en engendreront un autre. Cette information n'est disponible qu'à l'exécution. Par conséquent, il faut approximer les durées d'exécution des calculs pour estimer la latence de l'application placée.

Dans cette optique, la deuxième solution consiste à faire une évaluation moyenne de la durée d'exécution de chaque tâche pour ainsi obtenir une latence moyenne de l'ensemble.

La troisième est de considérer le pire cas, c'est-à-dire celui où toutes les données initient le calcul le plus long (en temps).

2.3.3 Bornes de boucles paramétrées

Le problème des bornes de boucles paramétrées est le caractère dynamique que cela donne à la tâche. En effet, les modèles posés bénéficient de l'aspect statique du traitement

de signal systématique. Si l'on considère par exemple le partitionnement, il n'est pas possible de déterminer un découpage de l'espace d'itération dans un contexte de programmation par contrainte s'il n'est pas fixe. On a deux types de paramétrage. Le premier est constant sur un pointage, mais varie d'un pointage à l'autre, *i.e.*, à chaque instance d'un mode MFR. Le deuxième, paramétrage varie pendant la durée d'un pointage (en fonction du groupe de rafales par exemple).

2.3.3.1 Paramètre constant Il s'agit d'un paramètre dont la valeur est décidée avant le début d'un pointage, parmi plusieurs possibles. En première approximation, on affecte au paramètre une valeur au pire cas parmi celles possibles. Bien entendu, l'idéal serait de prendre en compte toutes les valeurs possibles simultanément.

2.3.3.2 Paramètre variable Nous avons opté pour la réécriture de la tâche, de façon à rendre constante les bornes de boucles. L'inconvénient de cette solution est la perte de la sémantique du calcul car bien souvent la réécriture ne permet pas d'obtenir un algorithme équivalent. Contrairement aux accès multiples à un tableau, cette perte correspond également à une modification dans les accès mémoire, les temps de calcul, de communications et modifie les dépendances. C'est pourquoi, la réécriture doit faire en sorte de sur-approximer les volumes mémoire utiles et les dépendances afin d'assurer la validité de la solution donnée. En effet, si le système trouve une solution pour un problème plus contraint, au sens où les ressources machines sont plus sollicitées, il est raisonnable de penser qu'il existe une solution pour le problème réel.

2.4 Conclusion

Nous savons désormais prendre en compte des applications dont le GFD est quelconque, où les fonctions d'accès aux données des tableaux sont affines (cycliques ou non). Le traitement dynamique (conditionnement, paramétrage,...) est rendu statique par combinaison de considérations de cas moyen et pire cas. Ceci est principalement dû à notre approche parce que l'utilisation de la PPC en domaines finis suppose que les variables manipulées sont instanciées par des valeurs et non des ensembles.

Admettons que l'on remplace toutes les entrées dynamiques par des variables de domaines représentant toutes les valeurs possibles de ces paramètres (on ne cherchera donc pas à les instancier au cours de la recherche de solution). Il faudrait les contraindre de façon à ce que les différentes propagations ne réduisent pas leur domaines (ce qui n'est pas possible). Enfin, les solutions obtenues seraient sans garantie, car rien ne permettrait de dire s'il existe ou non des valeurs pour ces variables telles que les solutions obtenues soient incohérentes. Par conséquent, l'approximation par des pires cas ou des cas moyens reste ce qu'il y a de plus sûr aujourd'hui, en terme de garantie de résultat. Il faut cependant garder à l'esprit que si la recherche de solution n'aboutit pas, cela ne signifie pas qu'il n'en existe pas pour autant,mais seulement que les approximations empêchent d'en trouver une.

Chapitre 4

Le domaine architectural

1 Définition de l'architecture cible

Notre méthode de placement prend en considération l'application de TSS à placer, mais également certains points de l'architecture multi-processeurs de la machine cible. Nous présentons ces derniers dans cette section. Cependant, pour des raisons de confidentialité, nous ne ferons ici qu'un survol de cette machine, permettant malgré tout d'en comprendre globalement son architecture. C'est pourquoi nous ne donnerons aucun détail sur l'architecture et aucune valeur numérique de capacité mémoire, de débit du réseau de communication ou de durée d'un cycle. Les valeurs utilisées pour les expérimentations permettent d'obtenir des résultats cohérents mais ne sont pas représentatives des valeurs réelles.

1.1 Structure générale

La machine cible a une architecture multi-processeurs à mémoire distribuée. Un processeur (ou nœud de calcul) est constitué de trois éléments : une *unité de calcul*, des *mémoires* et des *DMA*s pour les échanges de données. Ce nœud de calcul est présenté dans la section suivante.

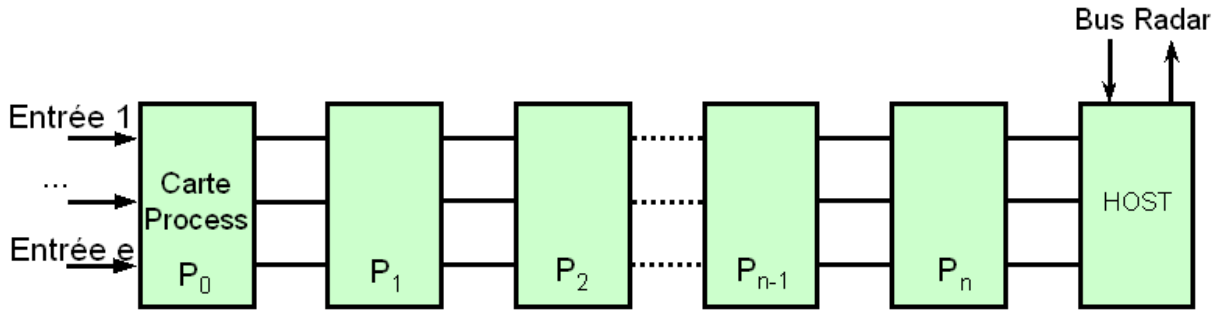
Des connections liant les nœuds de calcul permettent des échanges des données des uns vers les autres (fig. 4.1). Les communications sont gérées par ailleurs. Cette gestion permet de configurer ces connections selon le protocole désiré. Nous ne pouvons détailler davantage le mode de communication ainsi que l'architecture globale pour des raisons de confidentialité.

1.2 Structure d'un nœud

Un nœud de calcul est l'ensemble *Unité de calcul + mémoire + DMA* (fig. 4.2). La vision des interconnexions de ces trois données, proposée ici, reste simplifiée par rapport à la réalité. Cependant elle fait apparaître les aspects les plus importants pour le placement.

Un nœud de calcul est organisé autour d'un processeur. Les données entrent et sortent du nœud en transitant par une mémoire d'entrée/sortie. Les calculs sont faits par le processeur dans le bloc UC. Il communique exclusivement avec la mémoire interne. Les échanges sont

FIG. 4.1 – Représentation générale de la machine cible modélisée

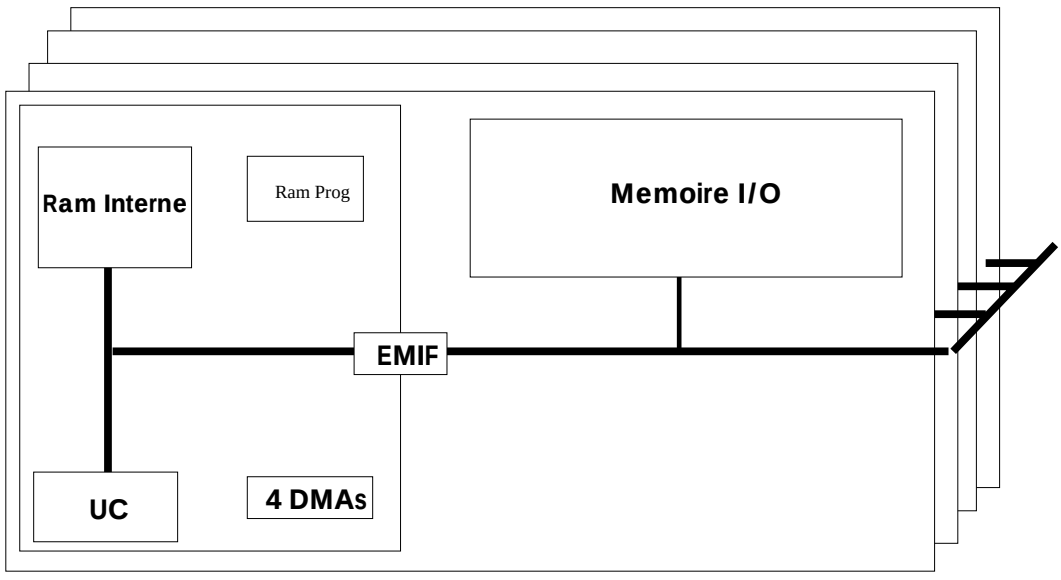


rapides grâce à une large bande passante. En contrepartie, cette Ram Interne est de petite capacité. Elle est complétée d'une mémoire plus lente mais de plus grande capacité. Cette dernière est connectée au bus liant le processeur au reste du réseau.

L'utilisation de données stockées dans les mémoires autres que la Ram Interne n'est pas directe (pour des raisons de performance, cette possibilité d'utilisation est interdite) et passe par l'EMIF (External Memory InterFace). Les communications entre mémoires et Ram Interne sont simultanées avec les communications entre l'UC et la Ram Interne.

La mémoire Ram prog contient les instructions du programme à exécuter. Elle peut éventuellement être agrémentée d'une autre mémoire externe au processeur. Elles ne sont pas prises en compte, bien que dans la réalité, leur communication par l'EMIF puissent rendre critique l'utilisation du bus.

FIG. 4.2 – Structure d'un nœud : unité de calcul + mémoires + DMA

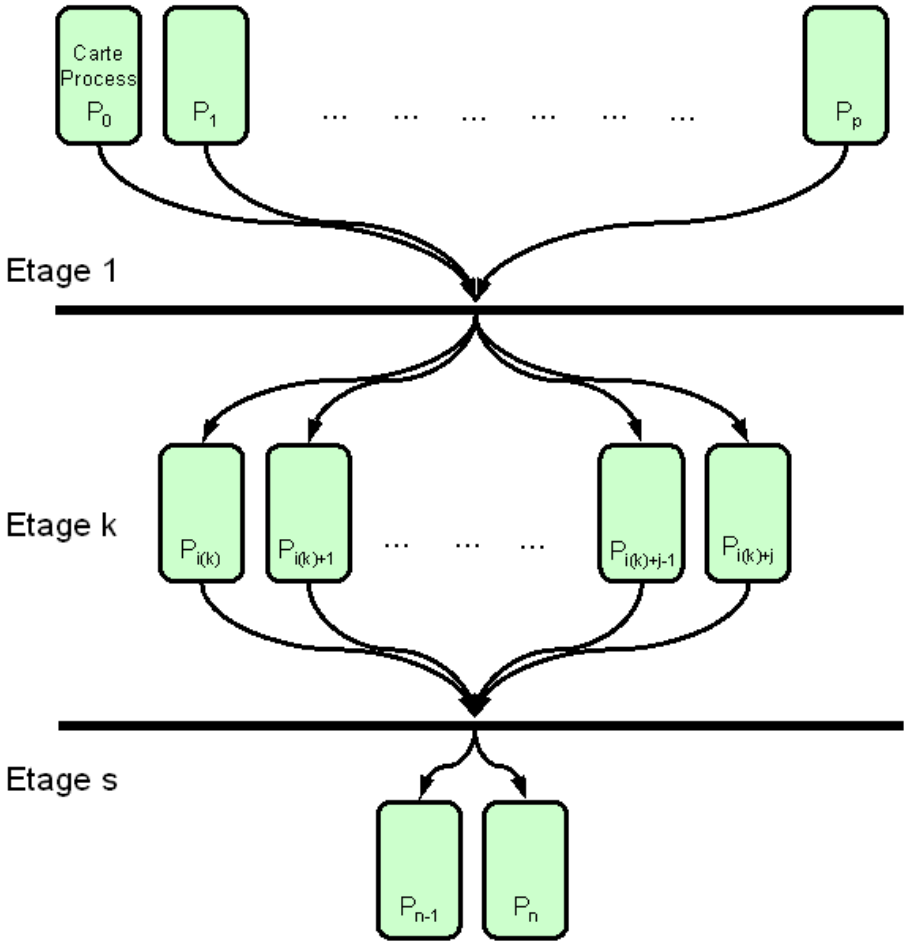


1.3 Prise en compte du mode M-SPMD

La considération du mode de programmation M-SPMD consiste en l'organisation virtuelle de la machine en pipeline de processeurs. Le support de communication ainsi que le protocole permettent de regrouper les processeurs en grappes. À chaque grappe correspond un étage du pipeline.

Les $p+1$ premiers processeurs constituent le premier étage de pipeline, les $j+1$ suivants le deuxième, ainsi de suite (fig. 4.3).

FIG. 4.3 – Structure fonctionnelle pour le mode de programmation M-SPMD



1.4 Conclusion

Même si l'on se rapproche petit à petit d'une certaine réalité, notre représentation de l'architecture cible reste simplifiée par rapport à l'architecture réelle : nous n'avons pris en considération ni les différents types de mémoire, ni leur hiérarchie (problème pris en compte par le projet RNRT PROMPT [7] évoqué dans le chapitre précédent).

Il est à noter que bien que la description de l'architecture cible soit très peu détaillée ici,

les travaux présentés dans ce document reposent fortement sur une architecture existante d'une entité THALES. Cette collaboration étroite a permis de rester dans un contexte simplifié mais proche de la réalité et significatif.

2 Le mode multi-SPMD

Les machines M-SPMD peuvent traiter des calculs de manière pipelinée. Elles sont constituées d'une suite de machines définissant chacune un étage de ce pipeline. Ces dernières sont également parallèles, mais de mode SPMD ou SIMD.

De telles machines sont utilisées en traitement d'images et en traitement du signal, en particulier en traitement du signal radar.

Cette section a pour objectif de voir comment une machine cible M-SPMD peut être modélisée par un ensemble de contraintes. La section 2.1 présente les différentes hypothèses considérées, ainsi qu'un exemple illustrant l'intérêt du M-SPMD par rapport au SPMD. La section 2.2 regroupe les différents modèles de la fonction placement pour lesquelles l'interaction avec la prise en compte d'une machine M-SPMD est faible. Les sections 2.3 et 2.4 présentent respectivement les modèles de *latence-débit* et *machine* dans un contexte M-SPMD. La section 2.5 n'est autre que la conclusion.

2.1 Pourquoi utiliser le mode multi-SPMD ?

Bien que la latence soit un critère prépondérant en traitement radar, avoir une efficacité et un débit les plus grands possibles est également un point important. Après avoir donné le cadre hypothétique, nous verrons dans un exemple que le M-SPMD prend mieux en compte ces deux paramètres que le SPMD. Les gains effectués sont cependant obtenus au détriment de la latence puisqu'elle ne peut plus être calculée précisément entre deux dates correspondant à des calculs, mais entre deux synchronisations. En effet, le fonctionnement des différents étages de pipeline les uns par rapport aux autres est asynchrone. Ainsi les seules dates permettant des mesures de temps sont les points de rendez-vous : les synchronisations. Pour les mêmes raisons, les durées des différentes allocations mémoire sont sur-évaluées, ce qui inflige un sur-coût quant aux volumes utilisés.

2.1.1 Définition et hypothèses du modèle M-SPMD

Le MULTI-SPMD permet d'exploiter le pipeline de tâches. Ce pipeline potentiel apparaît grâce au parallélisme sous-jacent à l'application que l'on veut placer, ou découle du partitionnement de ses différents nids de boucles. Il s'agit alors d'un parallélisme *temporel*.

Le pipeline est constitué en assignant les processeurs aux tâches et non aux données. Un fonctionnement en *barillet* où chaque processeur traite une itération temporelle différente ne rentre pas dans le cadre M-SPMD parce qu'une tâche est affectée à un étage de pipeline, ni dans le cadre SPMD parce que des processeurs exécutent des tâches différentes simultanément. Ce fonctionnement peut être approximé en réalignant temporellement les diverses exécution de manière à ce que tous les processeurs exécutent les mêmes tâches aux mêmes instants, au détriment de la mémoire et de la latence.

L'intérêt du modèle M-SPMD est l'amélioration de l'efficacité d'utilisation des processeurs et du débit par rapport à une machine SPMD.

La modélisation de notre machine M-SPMD est soumise aux hypothèses suivantes :

1. Les processeurs utilisés sont banalisés, *i.e.*, ils sont identiques.
2. L'ensemble des processeurs est partitionné de manière quelconque. Il n'y a pas de

contrainte sur le cardinal d'une partition, si ce n'est que le nombre total de processeurs est fixé.

3. Un processeur n'est utilisé que dans un et un seul étage de pipeline.
4. Les communications sont contraintes de diverses manières :
   (a) Il n'y a pas de communication dans un même étage de pipeline,
   (b) Les communications ne sont autorisées qu'entre étages de pipeline contigus,
   (c) Les communications sont monodirectionnelles et s'effectuent de l'étage de pipeline s à l'étage de pipeline $(s+k), k > 0$.
5. Un nid de boucle s'exécute sur un unique étage de pipeline.
6. La machine est synchronisée globalement. Les données produites dans un étage de pipeline ne sont disponibles dans le suivant qu'après une synchronisation globale. [1]
7. La i^e occurrence du s^e étage de pipeline est postérieure à la i^e occurrence du $(s-1)^e$ étage de pipeline (GFD connexe). Le temps d'exécution de n périodes d'un pipeline de S étages est $(n+S-1) \times \max_S(\mathrm{T}^{\mathrm{s}}_{Exec})$ où $\mathrm{T}^{\mathrm{s}}_{Exec}$ est la durée d'exécution de l'étage s. Parmi les latences d'une application, la valeur minimale de la latence maximale est $S \times \max_s(\mathrm{T}^{\mathrm{s}}_{Exec})$.
8. Les ressources sont non partageables dans un étage de pipeline, aussi bien les ressources de communications que les ressources de calculs.
9. Il n'y a pas de conflit de ressources : 1 canal émetteur / n canaux récepteurs.
10. Les communications sont non-préemptives.
11. Les bandes passantes entre deux étages de pipeline sont définies en fonction du nombre de processeurs actifs de l'étage émetteur. Si $\frac{1}{3}$ des processeurs de l'étage s sont utilisés, alors $\frac{1}{3}$ de la bande passante est à disposition.

Remarquons que la conjonction des hypothèses 4b et 4c impose $k = 1$.

2.1.2 Exemple SPMD / M-SPMD

Le but de cet exemple est de montrer l'importance de l'efficacité et du débit pour justifier une architecture M-SPMD par rapport à une architecture SPMD.

2.1.2.1 Description de l'exemple Considérons une application à deux nids de boucles, NB_1 et NB_2, illustrée par le pseudo-code suivant :

```
CC NB1
do t = 0, ...
  do v = 0, 15
    X1[t,v] = F1(X0[t,v])
  end do
end do
```

```
CC NB2
do t = 0, ...
  X2[t] = F2(X1[t,0..15])
end do
```

Soit T_1 (resp. T_2) la durée d'exécution d'une itération de F_1 (resp. F_2) dans le nid de boucles NB_1 (resp. NB_2). NB_1 a un parallélisme potentiel de 16, alors que NB_2 n'est pas parallélisable (si ce n'est dans la dimension temporelle qui nous obligerait à faire intervenir les entrées/sorties). Nous appellerons p le nombre de processeurs utilisés pour exécuter l'ensemble de l'application.

[1] Cette hypothèse est importante, car c'est elle qui permet de différencier le M-SPMD du MIMD.

2.1.2.2 D'un point de vue SPMD Dans un cadre SPMD, le nombre de processeurs utilisés pour exécuter l'ensemble de l'application est le nombre de processeurs maximum utilisés pour chaque nid de boucles partitionné.

La durée d'exécution T_{exec} est la somme des durées :

$$\forall p \geqslant 1, \quad T_{execSPMD}(p) = \left\lceil \frac{16}{p} \right\rceil T_1 + T_2$$

ce qui, de plus correspond à la latence $T_{latSPMD}$, ainsi qu'à la durée de la période (voir partie gauche de la figure 4.4). Ainsi, l'efficacité est :

$$\forall p \geqslant 1, \quad E_{SPMD}(p) = \frac{16.T_1 + T_2}{p\left(\left\lceil \frac{16}{p} \right\rceil T_1 + T_2\right)}$$

On remarque que l'efficacité n'est maximale que lorsque $p = 1$.

Théorème 4.2-1 : Soit une application à deux nids de boucles, NB_1 et NB_2. Soit T_1 (resp. T_2) la durée d'exécution d'une itération de la fonction F_1 (resp. F_2) du nid de boucles NB_1 (resp. NB_2). NB_1 a un parallélisme potentiel de p_{pot}, alors que NB_2 n'est pas parallélisable. Soit p le nombre de processeurs utilisés pour exécuter cette application. L'efficacité, sur une machine SPMD, $E_{SPMD}(p) = \frac{p_{pot}.T_1+T_2}{p\left(\left\lceil \frac{p_{pot}}{p} \right\rceil T_1+T_2\right)}$, avec $T_1 > 0$ et $T_2 > 0$, est maximale si et seulement si $p = 1$.

Preuve : L'efficacité est maximale lorsqu'elle vaut 1.

Il est évident que $(E_{SPMD}(p) = 1) \Leftarrow (p = 1)$. Montrons $(E_{SPMD}(p) = 1) \Rightarrow (p = 1)$:

$$\begin{array}{lrcl}
\text{On a :} & E_{SPMD}(p) & = & 1 \\
\Leftrightarrow & p_{pot}.T_1 + T_2 & = & p.\left(\left\lceil \frac{p_{pot}}{p} \right\rceil T_1 + T_2\right) \\
\Leftrightarrow & p_{pot}.T_1 + T_2 & = & p.\left\lceil \frac{p_{pot}}{p} \right\rceil T_1 + p.T_2 \\
\Leftrightarrow & \left(p_{pot} - p.\left\lceil \frac{p_{pot}}{p} \right\rceil\right).T_1 + (1-p).T_2 & = & 0
\end{array}$$

Or $(1-p) \leqslant 0$, et par définition $p.\left\lceil \frac{p_{pot}}{p} \right\rceil \geqslant p_{pot}$ donc $\left(p_{pot} - p.\left\lceil \frac{p_{pot}}{p} \right\rceil\right) \leqslant 0$. T_1 et T_2 étant strictement positifs, alors $\left(p_{pot} - p.\left\lceil \frac{p_{pot}}{p} \right\rceil\right).T_1 + (1-p).T_2 = 0$ si et seulement si $(1-p) = 0$ et $\left(p_{pot} - p.\left\lceil \frac{p_{pot}}{p} \right\rceil\right) = 0$. C'est-à-dire lorsque $p = 1$.

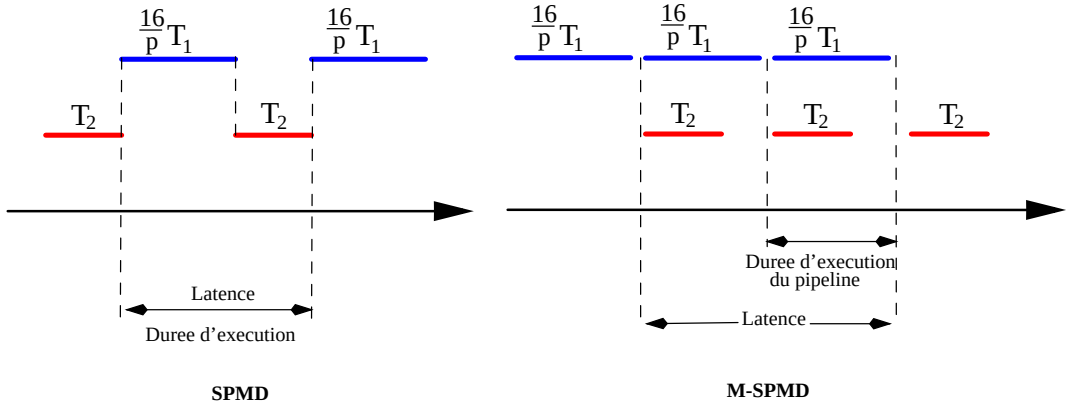
2.1.2.3 D'un point de vue M-SPMD Dans un cadre M-SPMD, le nombre de processeurs utilisés pour exécuter l'ensemble de l'application est la somme des nombres de processeurs maximum utilisés par étage de pipeline.

C'est pourquoi, il est nécessaire d'avoir strictement plus d'un processeur pour pouvoir pipeliner les deux nids de boucles, $p > 1$. Dans ce contexte, la durée d'exécution $T_{execM-SPMD}(p)$ est la durée maximale des étages de pipeline, soit

$$\forall p \geqslant 2, \quad T_{execM-SPMD}(p) = \max\left(\left\lceil \frac{16}{p-1} \right\rceil T_1, T_2\right)$$

La latence $T_{latM-SPMD}(p)$ vaut, dans notre exemple, 2 fois $T_{execM-SPMD}$, à cause de la synchronisation globale des étages de pipeline (hypothèse 6). La partie droite de la figure 4.4 permet, intuitivement, de comprendre pourquoi.

FIG. 4.4 – Exemple à deux nids de boucles : SPMD Vs M-SPMD



L'efficacité en M-SPMD E_{M-SPMD} est obtenue en calculant le rapport suivant :

$$\forall p \geqslant 2, \quad E_{M-SPMD}(p) = \frac{16.T_1 + T_2}{p.\max\left(\left\lceil \frac{16}{p-1} \right\rceil T_1, T_2\right)}$$

On constate que l'efficacité peut-être maximale lorsqu'il y a équilibre de charge (*i.e.*, $T_2 = \left\lceil \frac{16}{p-1} \right\rceil T_1$). Il suffit que $p-1$ divise 16 (dans notre exemple).

Théorème 4.2-2 : Soit une application à deux nids de boucles, NB_1 et NB_2. Soit T_1 (resp. T_2) la durée d'exécution d'une itération d'une fonction F_1 (resp. F_2) du nid de boucles NB_1 (resp. NB_2). NB_1 a un parallélisme potentiel de p_{pot}, alors que NB_2 n'est pas parallélisable. Soit p le nombre de processeurs utilisés pour exécuter cette application ($p > 1$). L'efficacité, sur une machine M-SPMD, $E_{M-SPMD}(p) = \frac{p_{pot}.T_1+T_2}{p.\max\left(\left\lceil \frac{p_{pot}}{p-1} \right\rceil T_1, T_2\right)}$ est maximale si et seulement si $(p-1)$ divise p_{pot} et qu'il y a équilibre de charge.

Preuve : Comme précédemment, l'efficacité est maximale lorsqu'elle vaut 1. Son calcul faisant intervenir un max de deux termes, montrons le théorème pour les deux valeurs possibles du max.

- $\max\left(\left\lceil \frac{p_{pot}}{p-1} \right\rceil T_1, T_2\right) = \left\lceil \frac{p_{pot}}{p-1} \right\rceil T_1 \quad (1)$

$$\begin{array}{rrcll}
 & E_{SPMD}(p) & = & 1 & \\
\Leftrightarrow & p_{pot}.T_1 + T_2 & = & p.\left\lceil \frac{p_{pot}}{p-1} \right\rceil .T_1 & \\
\Leftrightarrow & p.\left\lceil \frac{p_{pot}}{p-1} \right\rceil .T_1 - p_{pot}.T_1 & = & T_2 & (2) \\
(1) \text{ et } (2) \Rightarrow & p.\left\lceil \frac{p_{pot}}{p-1} \right\rceil .T_1 - p_{pot}.T_1 & \leqslant & \left\lceil \frac{p_{pot}}{p-1} \right\rceil .T_1 & \\
\Leftrightarrow & \left\lceil \frac{p_{pot}}{p-1} \right\rceil .(p-1) & \leqslant & p_{pot} & \text{car } T_1 > 0
\end{array}$$

Or par définition $\left\lceil \frac{p_{pot}}{p-1} \right\rceil .(p-1) \geqslant p_{pot}$, donc $\left\lceil \frac{p_{pot}}{p-1} \right\rceil .(p-1) = p_{pot}$, d'où :

$$\boxed{(p-1) \text{ divise } p_{pot}}$$

Par conséquent, (2) devient

$$\boxed{T_2 = \frac{p_{pot}}{p-1}.T1}$$

Il y a donc équilibre des charges de calcul.

- $\max\left(\left\lceil \frac{p_{pot}}{p-1} \right\rceil T_1, T_2\right) = T_2$ (1)

$$\begin{array}{rrcll} & E_{SPMD}(p) & = & 1 & \\ \Leftrightarrow & p_{pot}.T_1 + T_2 & = & p.T_2 & \\ \Leftrightarrow & \frac{p_{pot}}{p-1}.T_1 & = & T_2 & (2) \end{array}$$

or $T_2 \in \mathbb{N}^\star$, donc $(p-1)$ divise $p_{pot}.T_1$.

$$\begin{array}{rrcll} (1) \text{ et } (2) \Rightarrow & \frac{p_{pot}}{p-1}.T_1 & \geqslant & \left\lceil \frac{p_{pot}}{p-1} \right\rceil .T_1 & \\ \Leftrightarrow & \frac{p_{pot}}{p-1} & \geqslant & \left\lceil \frac{p_{pot}}{p-1} \right\rceil & \text{car } T_1 > 0 \end{array}$$

Or par définition $\left\lceil \frac{p_{pot}}{p-1} \right\rceil \geqslant \frac{p_{pot}}{p-1}$, donc $\left\lceil \frac{p_{pot}}{p-1} \right\rceil = \frac{p_{pot}}{p-1}$,
d'où

$$\boxed{(p-1) \text{ divise } p_{pot}}$$

De plus, (2) signifie alors directement qu'il y a équilibre de charge de calcul.

2.1.2.4 Comparaison La comparaison des deux latences nous amène à dire que, en supposant que le même nombre de processeurs soit utilisé, nous avons $\forall p \geqslant 2, \quad T_{latM-SPMD}(p) \geqslant T_{latSPMD}(p)$. En effet,

$$T_{latM-SPMD}(p) = 2 \times T_{execM-SPMD}(p) \quad \geqslant \left\lceil \tfrac{16}{p-1} \right\rceil T_1 + T_2 \quad \geqslant \left\lceil \frac{16}{p} \right\rceil T_1 + T_2 = T_{latSPMD}(p)$$

C'est-à-dire que, dans cet exemple, le critère de latence n'est pas le bon critère à optimiser puisqu'alors cela conduirait à un pipeline d'un seul étage, *i.e.*, un modèle SPMD. Et malheureusement, l'efficacité maximale est alors atteinte pour un code s'exécutant sur un seul processeur (théorème 1), au détriment de la mémoire et du débit. Mais une optimisation selon le critère d'efficacité avantage le modèle M-SPMD, puisqu'elle est maximale à équilibre de charge. On a alors :

$$E_{M-SPMD}(p) = \frac{p_{pot}.T_1 + T_2}{p.\max\left(\left\lceil \frac{p_{pot}}{p-1} \right\rceil T_1, T_2\right)} > \frac{p_{pot}.T_1 + T_2}{p\left(\left\lceil \frac{p_{pot}}{p} \right\rceil T_1 + T_2\right)} = E_{SPMD}(p)$$

2.1.3 Variables de placement M-SPMD

Chaque nid de boucles nb est associé à un et un seul étage de pipeline (hypothèse 5). Il existe donc une application surjective entre les deux ensembles :

$$\begin{array}{rrcl} \text{Stage} : & \text{NB} & \longrightarrow & STAGE = [1, \text{NbMax}] \\ & nb & \longmapsto & s \end{array} \tag{4.1}$$

NBS désignera le nombre d'étages effectifs du pipeline. On a NBS $\leqslant$ NbMax.

Le modèle M-SPMD englobe strictement le modèle SPMD, puisqu'il suffit de mettre tous les nids de boucles dans le même étage de pipeline, à condition que l'hypothèse 4a soit levée. Aussi, si une solution SPMD est meilleure pour un critère donné que les solutions M-SPMD, alors elle sera choisie.

2.2 Interactions avec les modèles de la fonction placement

Comme nous l'avons vu dans le chapitre 1, la fonction placement est découpée en sept modèles : partitionnement, dépendances, interphase (communications), ordonnancement, mémoire, machine et temps-réel. Nous montrons dans cette section que les cinq premiers de ces modèles ne sont pas ou peu touchés par la prise en compte du modèle architectural M-SPMD.

2.2.1 Partitionnement

Nous avons envisagé de faire apparaître la notion d'étage de pipeline explicitement au niveau du partitionnement. Un étage de pipeline aurait alors calculé un superbloc correspondant à plusieurs valeurs de c. Les avantages de cette explicitation n'ont pas semblé suffisants pour y procéder et le partitionnement choisi dans [32] a été conservé.

2.2.1.1 Ajout d'une dimension L'ajout d'une dimension au partitionnement permettrait de disposer d'une dimension supplémentaire pour le modèle d'ordonnancement et de mettre en évidence la période d'une application M-SPMD. L'ensemble des ordonnancements possibles serait alors strictement plus riche. L'espace des solutions serait naturellement accru ainsi que le temps de résolution. L'obtention de solutions supplémentaires n'étant pas notre motivation première, il nous a donc semblé inutile de compliquer la résolution. Nous avons préféré utiliser une période globale implicite α déductible de l'ordonnancement (sec. 2.2.2 ordonnancement).

Par ailleurs, il s'est naturellement posé le problème de cohérence sémantique et mathématique entre les paramètres matriciels/vectoriels du partitionnement et le paramètre scalaire de l'étage de pipeline. S'il est toujours possible de trouver une fonction permettant de passer d'un espace multi-dimensionnel à un espace mono-dimensionnel, la cohérence sémantique est plus problématique. En effet, le partitionnement, de part sa fonction, est lié à chacune des boucles de l'espace d'itérations, alors que l'attribution d'un étage de pipeline concerne le nid de boucles dans sa globalité (hyp. 5). Les seules relations entre le partitionnement d'un nid de boucles et son attribution à un étage de pipeline apparaissent dans la détection des communications (chap. 5). Car les partitionnements choisis lors de la recherche de solution influencent la présence ou l'absence de communication.

2.2.1.2 Partitionnement indépendant du mode M-SPMD Dans un étage de pipeline, la définition du partitionnement 3D présentée dans [32] reste identique, puisqu'il fonctionne en SPMD. Nous avons toujours, pour chaque nid de boucles *nb* la contrainte 2.6 (p. 44).

Ce partitionnement garantit que tous les calculs sont distribués, qu'aucun n'est dupliqué, et que tous les processeurs de la partition sont actifs ([32, Chap. 4]).

Il existe cependant un lien avec le M-SPMD. Le déterminant de P^{nb} correspond au nombre de processeurs utilisés par une partition. Comme ce partitionnement est local à un étage du pipeline, la contrainte de ressources de calcul doit prendre en compte le nombre de processeur de l'étage sur lequel se trouve le nid de boucles nb. Cette contrainte est donnée dans la section 2.4.

2.2.2 Ordonnancement

La façon de modéliser l'ordonnancement n'est pas modifiée. Il est toujours représenté par un fonction affine du paramètre vectoriel cyclique du partitionnement :

$$\forall nb \in \mathrm{NB}, \forall c^{nb} \prec c^{nb}_{max}, \quad \mathrm{d}\left(c^{nb}\right) = \alpha^{nb}.c^{nb} + \beta^{nb}$$

Deux synchronisations globales successives sont séparées par une durée événementielle que nous appellerons α. Cette durée définit la période d'apparition des synchronisations globales.

Définition 4.2-1 : Période de synchronisation α

La période de synchronisation des étages du pipeline, notée α, est telle que :

$$\alpha = \mathrm{ppcm}_{nb}\left(\alpha^{nb}_0\right) \tag{4.2}$$

où α^{nb}_0 désigne la période de calcul d'une itération de la boucle la plus externe du nid de boucles nb.

De cette définition, et par définition des composantes α^{nb} et c^{nb}, nous déduisons le nombre de blocs de calcul d'un nid de boucles nb (de profondeur ld) sur une période α, noté $\mathrm{nbc}^{nb}(\alpha)$, donné par :

$$\mathrm{nbc}^{nb}(\alpha) = \frac{\alpha}{\alpha^{nb}_0} \times \prod_{i=1}^{ld^{nb}-1} c^{nb}_{max,i} \tag{4.3}$$

Les dépendances sont préservées, *i.e.*, deux blocs de calcul dépendants ne sont pas exécutés en même temps. C'est-à-dire que l'on reprend les contraintes 2.10 et 2.1.2 en y ajoutant l'aspect hiérarchique du pipeline :

Définition 4.2-2 :

Dans un étage de pipeline, deux blocs de calculs dépendants ne s'exécutent pas en même temps :

$$\alpha^{nb} = N_s.\gamma^{nb} \tag{4.4}$$

$$\beta^{nb} = N_s.\delta^{nb} + k^{nb}_s \tag{4.5}$$

avec N_s le nombre de nid de boucles dans l'étage de pipeline s et $k^{nb}_s < N_s$ le numéro attribué au nid de boucles nb (de profondeur ld) dans l'étage de pipeline s, $\alpha^{nb}_{ld-1} \geqslant 1$ et $\forall n \in [1, ld-1] \quad \alpha^{nb}_{n-1} \geqslant \alpha^{nb}_n c^{nb}_n$.

La date logique de la première exécution d'un bloc de calcul du nid de boucles nb, β^{nb}, est au moins égale au nombre d'événements produits jusqu'à la synchronisation globale précédant le lancement de l'étage de pipeline de nb. C'est-à-dire :

$$\beta^{nb} \geqslant (\text{Stage}(nb) - 1) \times \alpha \tag{4.6}$$

2.2.3 Dépendances

Les dépendances sont liées à l'application, au modèle de partitionnement et à la fonction d'ordonnancement. Ces deux derniers modèles restant inchangés, les dépendances ne sont pas modifiées par la prise en compte d'une nouvelle machine cible.

Cependant, les hypothèses 4b et 4c permettent de dire que si un nid de boucle nb^2 dépend d'un nid de boucle nb^1, il s'exécute dans un étage de pipeline égal ou supérieur :

✓
$$\forall nb^1, \forall nb^2, \quad \left(\exists c^{nb^1}, c^{nb^2} \quad \mathcal{D}\left(c^{nb^1}, c^{nb^2}\right)\right) \Rightarrow \left(0 \leqslant \text{Stage}(nb^2) - \text{Stage}(nb^1) \leqslant 1\right) \tag{4.7}$$

Deux blocs de calculs dépendants s'exécutant dans deux étages différents doivent être séparés par au moins une synchronisation globale (hypothèse 6).

✓
$$\begin{gathered}\forall nb_1, nb_2 \in \text{NB}, \ \text{Stage}(nb^1) < \text{Stage}(nb^2), \\ \forall c^{nb^1}, \forall c^{nb^2} \\ \left(\mathcal{D}\left(c^{nb^1}, c^{nb^2}\right)\right) \Rightarrow \left(\exists \lambda > 0 \text{ t.q. } \mathrm{d}^{nb^1}\left(c^{nb^1}\right) < \lambda.\alpha \leqslant \mathrm{d}^{nb^2}\left(c^{nb^2}\right)\right)\end{gathered} \tag{4.8}$$

Remarque : Il est possible de supprimer le quantificateur existentiel de la contrainte (4.8). En effet :

$$(4.8) \Leftrightarrow \left(\left\lfloor \frac{\mathrm{d}^{nb^1}\left(c^{nb^1}\right)}{\alpha} \right\rfloor < \left\lfloor \frac{\mathrm{d}^{nb^2}\left(c^{nb^2}\right)}{\alpha} \right\rfloor\right) \tag{4.9}$$

$$\text{ou encore} \Leftrightarrow \left(\left\lfloor \frac{\mathrm{d}^{nb^2}\left(c^{nb^2}\right)}{\alpha} \right\rfloor - \left\lfloor \frac{\mathrm{d}^{nb^1}\left(c^{nb^1}\right)}{\alpha} \right\rfloor \geqslant 1\right) \tag{4.10}$$

2.2.4 Mémoire

Trois types de zones de mémoire doivent être allouées par processeur par nid de boucles :

1. les zones de réception correspondant à un terme droit produit dans un étage de pipeline précédent,
2. les zones d'émission correspondant à un terme gauche utilisé dans un étage de pipeline suivant,
3. les zones internes à un processeur traitant deux nids de boucles successifs dans un même étage de pipeline.

L'utilisation des deux premières zones est synchronisée par les périodes globales.

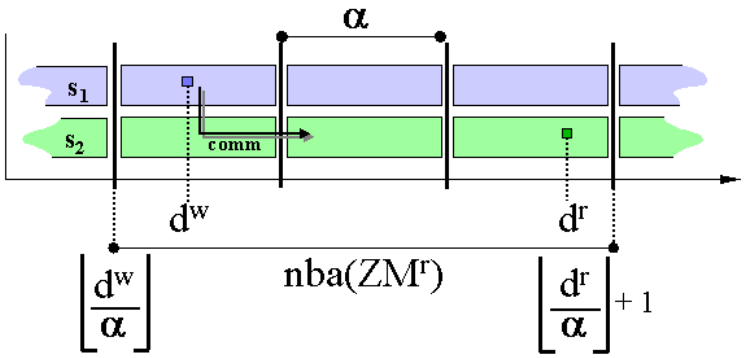
2.2.4.1 Zones de réception : entrées des nids de boucles Le volume mémoire dépend de la durée de vie des données. Il est sensible aux valeurs de l'offset β. L'implication (4.8) va se traduire par une augmentation de β^{nb^2} puisque les $\alpha_0^{nb^2}$ sont déjà contraints en dimension infinie par les dépendances et la taille finie des buffers. Cette augmentation correspond à un allongement de la durée de vie de la variable portant la dépendance. Le volume mémoire nécessaire en entrée va donc croître.

Comme les seules synchronisations sont globales, on ne saura qu'une zone peut être réutilisée en réception qu'à la fin de la période où elle est utilisée en entrée. Et il faudra que les transferts aient eu lieu dans la période de production. Il faut donc allouer toutes les zones nécessaires sur le nombre de périodes séparant la production de la consommation (fig. 4.5). Si l'on note d^r la date logique de consommation, d^w la date logique de production d'une zone mémoire ZM^r, alors

$$\mathrm{nba}\left(\mathrm{ZM}^r\right) = \left\lfloor \frac{\mathrm{d}^r}{\alpha} \right\rfloor - \left\lfloor \frac{\mathrm{d}^w}{\alpha} \right\rfloor + 1 \tag{4.11}$$

est le nombre de périodes globales séparant la production de la consommation.

FIG. 4.5 – Durée d'allocation d'une zone mémoire de réception



En l'absence de recouvrement, les données nécessaires, pour un bloc d'un nid de boucles r, ont un volume de $\det(L^r) \times \det(M^r_{in})$ (où M^r_{in} est la matrice diagonale constituée des m_{max} associés au tableau in). En présence de recouvrement, ce volume est inférieur mais se posent alors des problèmes d'adressage lors du retour au début de la zone d'entrée. Quoiqu'il en soit, le volume à allouer est borné supérieurement par le volume sans recouvrement $\mathcal{V}^{\mathrm{rcv}}_{r,in}$ (rcv pour *receive*) :

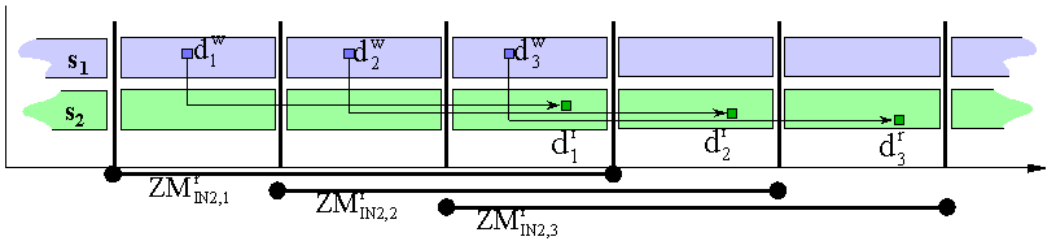
$$\mathcal{V}^{\mathrm{rcv}}_{r,in} = \mathrm{nba}\left(\mathrm{ZM}^r\right) \times \mathrm{nbc}^r\left(\alpha\right) \times \det(L^r) \times \det(M^r_{in}) \tag{4.12}$$

L'ensemble des zones de réception d'un nid de boucles est obtenu en sommant par rapport à tous les arguments du nid de boucles provenant d'un étage de pipeline précédent :

$$\mathcal{V}_r^{\text{rcv}} = \sum_{in} \mathcal{V}_{r,in}^{\text{rcv}} \tag{4.13}$$

$$= \text{nbc}^r(\alpha) \times \det(L^r) \times \left(\sum_{in} \text{nba}\left(\text{ZM}_{\text{in}}^{\text{r}}\right) \times \det(M_{in}^r) \right) \tag{4.14}$$

FIG. 4.6 – Allocation cumulée selon la durée de vie des données



2.2.4.2 Zones d'émission : sorties des nids de boucles Par ailleurs, la présence d'une unique synchronisation globale ne permet pas forcément de garantir que les buffers d'émissions seront vidés avant leur prochain remplissage. Les choix faits pour l'utilisation des buffers d'émission ont un impact sur l'estimation du temps d'exécution d'une période (cf. section 2.3) et vice-versa. Nous pouvons faire trois suppositions :

1. Les communications liées à un bloc ont lieu au plus tard pendant l'exécution du bloc suivant. Il faut disposer de deux buffers d'émission et générer une attente de fin de communication après chaque bloc de calcul sauf le premier d'une période (cf. fig. 4.7 cas 1). On peut sur-approximer le temps d'exécution d'une période en supposant que le parallélisme entre calcul et communication n'existe que pour une même tâche w.

   ✓ $$\mathcal{V}_{w,out}^{\text{snd}} = 2 \times \det(L^w) \times \det(M_{out}^w) \tag{4.15}$$

2. Les communications sont gérées de manière implicite par un exécutif, le nombre de buffers nécessaires est égal au nombre de blocs du nid de boucles considérés pendant une période (cf. fig. 4.7 cas 2). Il suffit de générer une attente globale en fin de période. Il est difficile de trouver une formule explicite précise d'approximation inférieure ou supérieure du temps d'exécution parce que les dates de début au plus tôt des communications dépendent des dates de fin de calcul.

   Le volume nécessaire est alors :

   $$\mathcal{V}_{w,out}^{\text{snd}} = \text{nbc}^w(\alpha) \times \det(L^w) \times \det(M_{out}^w) \tag{4.16}$$

3. On impose l'exécution immédiate des communications ou plutôt, on empêche le calcul d'un bloc jusqu'à ce que la communication associée au bloc précédent soit terminée. Il n'y aurait alors plus besoin que d'un unique buffer de sortie (cf. fig. 4.7 cas 3). Il pourrait y avoir du parallélisme entre les communications d'une tâche et les calculs de toutes les autres mais nous n'avons pas trouvé de formule explicite pour des bornes inférieures ou supérieures précises d'une période.

   $$\mathcal{V}_{w,out}^{\text{snd}} = \det(L) \times \det(M_{out}) \tag{4.17}$$

En pratique, l'utilisation d'un exécutif gérant les entrées-sorties et les communications semble être la solution la plus vraisemblable. On peut coupler cette utilisation avec une évaluation pessimiste des temps d'exécution fondée sur du parallélisme intra-tâche entre communications et calculs pour disposer d'une formule utilisable (cf.section 2.3.2.2).

Dans le modèle M-SPMD, le calcul du volume mémoire fait intervenir deux composantes importantes : les buffers d'émission et les buffers de réception. L'absence de synchronisations fines nécessite l'utilisation de larges buffers couvrant toute une période en émission et $\mathrm{nba}(r) \times \alpha$ périodes en réception. Cela parce que les synchronisations globales ont lieu une fois les communications terminées.

2.2.4.3 Zone interne : données d'un nid de boucles non communiquées Les données internes à un étage ne sont pas communiquées puisque les communications entre processeurs d'un même étage sont prohibées (hypothèse 4a).

Malgré tout nous pouvons assimiler les entrées internes aux entrées des zones de réceptions. La durée de vie de ces données est majorée par le nombre de blocs de calcul sur la plus grande distance de dépendances posée sur elles. On obtient la sur-approximation suivante :

$$\mathcal{V}_{r,in}^{\mathrm{input}} = \mathrm{nbc}\left(\max_{r,w}(\mathrm{d}^r(in) - \mathrm{d}^w(in))\right) \times \det(L^r) \times \det(M^r) \tag{4.18}$$

Pour le volume de données produites internes, nous avons deux possibilités. La première est de considérer que les données produites par un bloc de calcul d'un nid de boucles sont écrites directement dans les zones des entrées internes des nids de boucles consommateurs. Auquel cas on a alors :

$$\mathcal{V}_{w,out}^{\mathrm{output}} = 0 \tag{4.19}$$

La deuxième possibilité est de considérer l'existence de ce buffer. Les implications de cette considération sont la prise en compte d'un temps de recopie dans les zones des entrées internes des nids de boucles consommateurs (dans la mesure où la recopie est interne au processeur, nous pouvons négliger sa durée), et le fait que cela duplique ces données. Ce qui sur-évalue le volume nécessaire. Ce volume interne de données produites est obtenu par :

$$\mathcal{V}_{w,out}^{\mathrm{output}} = \det(L^w) \times \det(M^w) \tag{4.20}$$

Le volume mémoire interne d'un nid de boucles est la somme, pour chaque tableau utilisé en interne du pipeline, des volumes internes d'entrée et de sortie :

$$\mathcal{V}_{nb}^{\mathrm{int}} = \sum_{in}\left(\mathcal{V}_{nb,in}^{\mathrm{input}}\right) + \mathcal{V}_{nb,out}^{\mathrm{output}} \tag{4.21}$$

2.2.4.4 Volume global par étage Les tableaux ne partagent jamais d'espace mémoire. Ce n'est pas très gênant pour le modèle M-SPMD parce que chaque étage de pipeline a une

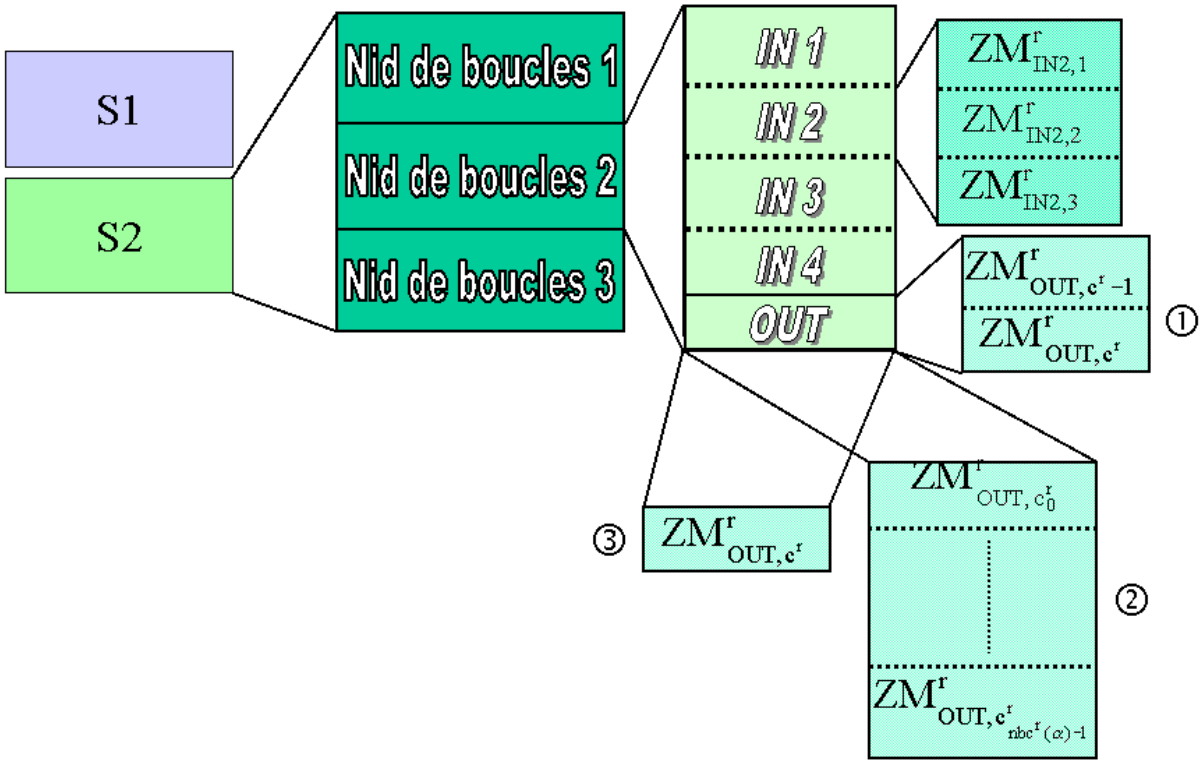
longueur de nids de boucles plus limitée que dans les modèles SPMD et SIMD. Les possibilités de réutilisation d'espace sont donc moins fréquentes. Il faut ajouter les volumes en entrée et en sortie par processeur pour tous les nids de boucles de l'étage, ainsi que les volumes des données internes.

$$\forall s, \quad \mathcal{V}_s = \sum_{\{nb|\ \mathrm{Stage}(nb)=s\}} \left(\mathcal{V}_{nb}^{\mathrm{rcv}} + \mathcal{V}_{nb}^{\mathrm{snd}} + \mathcal{V}_{nb}^{\mathrm{int}}\right) \tag{4.22}$$

2.2.4.5 Contrainte de volume La contrainte de volume doit être respectée par chaque processeurs de chaque étage.

$$\forall s \quad \mathcal{V}_s \leqslant \mathcal{V}_{max} \tag{4.23}$$

FIG. 4.7 – Répartition de la mémoire



La mémoire d'un étage de pipeline est segmentée en zones attribuées à chaque nid de boucles de cet étage. La zone mémoire réservée à un nid de boucles est également partagée, principalement en deux : une pour la réception et l'autre pour l'émission. La première est délimitée pour chaque tableau consommé. Selon la durée de vie des données cette zone sera plus ou moins importante. La zone mémoire d'émission est utilisée pour le tableau produit. Selon la supposition faite (suppositions 1, 2 ou 3 p.94) le calcul de son volume sera différent.

2.2.5 Entrées-sorties

En l'absence de synchronisations fines, les entrées-sorties doivent être cadencées par les synchronisations globales. Elles peuvent donc être effectuées par un exécutif. Toutes les données lues doivent être disponibles avant la synchronisation précédente et l'espace correspondant ne peut pas être réutilisé avant la synchronisation suivante. Ce qui correspond à :

$$2 \times \mathrm{nbc}^r(\alpha) \times \det(L^r) \times \det(M^r_{in}) \tag{4.24}$$

Les zones contenant des données écrites doivent être allouées avant la synchronisation précédant les calculs. Elles ne peuvent être considérées comme libérées qu'après la fin de la période suivante. Ce qui correspond à :

$$2 \times \mathrm{nbc}^w(\alpha) \times \det(L^w) \times \det(M^w_{in}) \tag{4.25}$$

Ceci revient à considérer que les mécanismes d'entrée et de sortie forment deux étages de pipeline supplémentaires et que ces volumes correspondent aux zones de réception et d'émission des nids de boucles associés.

2.3 Signal temps réel - latence - débit

Une des macro contraintes à respecter est le temps imparti au calculateur pour effectuer sa détection. Nous allons voir dans cette section comment introduire des durées dans notre modélisation.

2.3.1 La latence

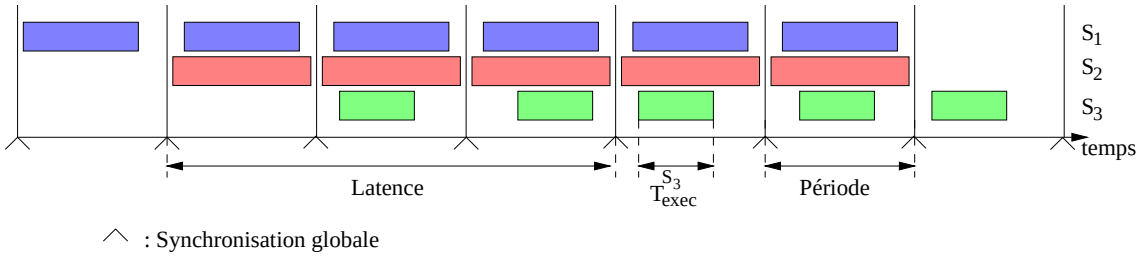
Les seuls repères temporels statiques en M-SPMD sont les points de synchronisations. En effet, les blocs de calculs appartenant à deux étages de pipelines différents s'exécutent de façon asynchrone. Donc les seuls événements pour lesquels les dates physiques sont prédictibles sont les points de rendez-vous, *i.e.*, les points de synchronisations globales.

Définition 4.2-3 :
On appelle $\mathrm{T}^{\mathrm{s}}_{exec}$ la durée physique de l'étage de pipeline s. La date physique du $k^{\text{ème}}$ point de synchronisation SP_k est donnée par :

$$\mathrm{SP}_k = k.\max_s(\mathrm{T}^{\mathrm{s}}_{exec}) \qquad \text{(hyp. 7)} \tag{4.26}$$

Dans un étage de pipeline, les calculs commencent au plus tôt à une synchronisation et les communications des derniers calculs se terminent au plus tard avant la suivante (hyp. 6). Par conséquent, des garanties sur la latence ne peuvent être fournies que si elle est calculée entre deux synchronisations (cf. figure 4.8).

FIG. 4.8 – Évaluation de la latence entre deux points de synchronisation



Théorème 4.2-3 : Soit α la durée événementielle séparant deux synchronisations successives (section 2.2.2, eq. 4.2), soit Δ la longueur événementielle du chemin critique du GFD partitionné, soit $\mathrm{T}^{\mathrm{s}}_{exec}$ est la durée physique de l'étage de pipeline s, alors la latence est telle que :

$$\left\lceil \frac{\Delta}{\alpha} \right\rceil . \max_s \left(\mathrm{T}^{\mathrm{s}}_{exec}\right) \leqslant \mathrm{lat}_{mspmd} \leqslant \left(\left\lceil \frac{\Delta}{\alpha} \right\rceil + 1 \right) . \max_s \left(\mathrm{T}^{\mathrm{s}}_{exec}\right) \quad (4.27)$$

Preuve : Soit $\mathrm{SP}_{k_{deb}}$ et $\mathrm{SP}_{k_{fin}}$ les dates physiques des synchronisations déterminant la latence. Alors

$$\begin{aligned} \mathrm{lat}_{mspmd} &= \mathrm{SP}_{k_{fin}} - \mathrm{SP}_{k_{deb}} \\ &= (k_{fin} - k_{deb}) . \max_s \left(\mathrm{T}^{\mathrm{s}}_{exec}\right) \end{aligned} \quad (4.28)$$

Appelons k la date logique de la première exécution du premier bloc de calculs à partir duquel le calcul de la latence commence. Alors $k_{deb} = \lfloor \frac{k}{\alpha} \rfloor$ et $k_{fin} = \lceil \frac{k+\Delta}{\alpha} \rceil$. (4.28) devient :

$$\mathrm{lat}_{mspmd} = \left(\left\lceil \frac{k+\Delta}{\alpha} \right\rceil - \left\lfloor \frac{k}{\alpha} \right\rfloor \right) . \max_s \left(\mathrm{T}^{\mathrm{s}}_{exec}\right)$$

Or, par définition des opérateurs d'arrondis, on a

$$\begin{aligned} \left\lfloor \frac{k}{\alpha} \right\rfloor + \left\lceil \frac{\Delta}{\alpha} \right\rceil &\leqslant \left\lceil \tfrac{k+\Delta}{\alpha} \right\rceil \leqslant & \left\lceil \frac{k}{\alpha} \right\rceil + \left\lceil \frac{\Delta}{\alpha} \right\rceil \\ \left\lfloor \frac{k}{\alpha} \right\rfloor + \left\lceil \frac{\Delta}{\alpha} \right\rceil - \left\lfloor \frac{k}{\alpha} \right\rfloor &\leqslant \left\lceil \tfrac{k+\Delta}{\alpha} \right\rceil - \left\lfloor \tfrac{k}{\alpha} \right\rfloor \leqslant & \left\lceil \frac{k}{\alpha} \right\rceil + \left\lceil \frac{\Delta}{\alpha} \right\rceil - \left\lfloor \frac{k}{\alpha} \right\rfloor \\ \left\lceil \frac{\Delta}{\alpha} \right\rceil &\leqslant \left\lceil \tfrac{k+\Delta}{\alpha} \right\rceil - \left\lfloor \tfrac{k}{\alpha} \right\rfloor \leqslant & \left\lceil \frac{\Delta}{\alpha} \right\rceil + 1 \end{aligned}$$

D'où le résultat.

Il reste encore à exprimer la durée physique $\mathrm{T}^{\mathrm{s}}_{exec}$ d'un étage de pipeline s.

2.3.2 La durée physique d'un étage de pipeline

Les ressources sont non partageables dans un étage de pipeline, aussi bien les ressources de calculs que les ressources de communications (hyp. 8). Cela signifie que les communications de blocs de calculs différents ne peuvent être simultanées (idem pour les calculs eux-mêmes).

Nous noterons $T_{cal}(k)$ et $T_{comm}(k)$ les durées physiques respectivement d'un bloc de calculs c^k et d'un bloc de communications d'un nid de boucles k.

Définition 4.2-4 :

La date d'exécution $t_k(c)$ d'un bloc de calcul c d'un nid de boucles k est supérieure à toutes les dates de fin de tous les blocs de calculs de tous les nids de boucles qui ont une date logique inférieure à celle de k.

$\forall k, k' \in \{nb \in \text{NB} |\ \text{Stage}(nb) = s\}$
$\forall c, c'$ t.q. $d_k(c) > d_{k'}(c')$

$$t_k(c) \geqslant t_{k'}(c') + T_{cal}(k') \tag{4.29}$$

Définition 4.2-5 : Communication par bloc

Une communication ne peut commencer avant que le bloc de calculs auquel elle est associée soit terminé. Soit $b_k(c)$ l'instant de début d'une communication :

$$\forall k \in \{nb \in \text{NB} |\ \text{Stage}(nb) = s\}, \quad \forall c \prec c^k_{max}, \quad b_k(c) \geqslant t_k(c) + T_{cal}(k) \tag{4.30}$$

Définition 4.2-6 : Politique FIFO

Une communication associée à un bloc de calcul k ne commence que si toutes les communications de tous les blocs de calculs de tous les nids de boucles ayant une date logique inférieure à celle de k sont terminées.

$$\forall k, k' \in \{nb \in \text{NB} |\ \text{Stage}(nb) = s\}, \forall c, c' \text{ t.q. } d_k(c) > d_{k'}(c') \quad b_k(c) \geqslant e_{k'}(c') \tag{4.31}$$

La date de fin d'une communication est déduite de sa date de début par :

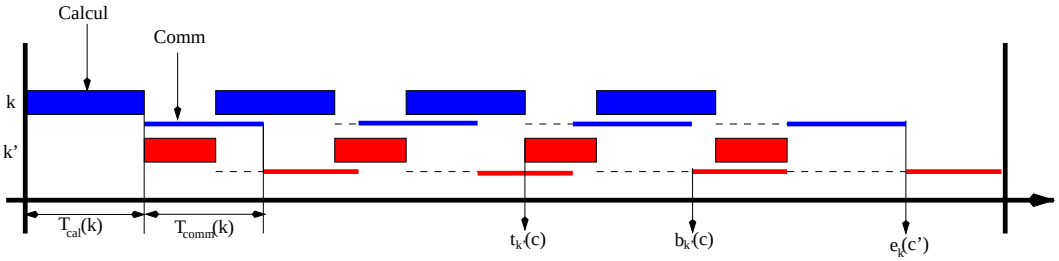
$$e_k(c) = b_k(c) + T_{comm}(k) \tag{4.32}$$

La période de l'étage de pipeline est supérieure à la durée séparant le lancement d'un calcul quelconque et la terminaison d'une communication quelconque :

$\forall d, \quad \forall k, c \quad \forall k', c'$ t.q. $d \leqslant d_{k'}(c') \leqslant d_k(c) < d + \alpha$

$$\text{T}^{\text{s}}_{Exec} \geqslant e_k(c) - t_{k'}(c') \tag{4.33}$$

FIG. 4.9 – Chronogramme en temps physique d'un étage de pipeline entre deux synchronisations globales successives : ordonnancement au plus tôt des calculs et des communications

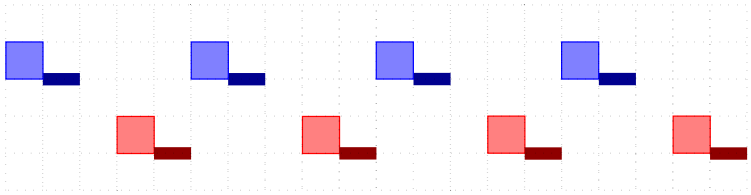


La figure 4.9 donne un exemple de chronogramme possible satisfaisant les contraintes (4.29) (4.30) (4.31). Pour calculer la durée physique de l'étage de pipeline, il faut alors

prendre en compte toutes les combinaisons possibles de recouvrement de calculs par des communications (ou l'inverse). Ce qui n'est pas possible à moins d'être exhaustif sur les dates et les durées de toutes les tâches (calcul et communications). C'est pourquoi nous choisissons d'allouer les ressources de communications d'un nid de boucles en même temps que les ressources de calcul du même nid de boucles. Dans la suite, nous appellerons ϵ^k la fonction d'ordonnancement des communications du nid de boucles k.

2.3.2.1 Les communications consécutivement aux calculs : $\epsilon^k(c^k) = \mathrm{d}^k(c^k)+1$ La figure 4.10 montre, sur une échelle de temps logique, une succession possible des événements calcul/communication dans le cas où les communications sont placées à l'instant logique suivant les calculs.

FIG. 4.10 – Ordonnancement des communications tel que $\epsilon(\mathrm{c}) = \mathrm{d}(\mathrm{c})+1$



Échelle de temps logique.

Si sur une échelle de temps événementielle, les calculs et les communications sont disjoints les uns des autres, il peut en être tout autrement sur une échelle de temps physique. La figure 4.11 montre trois chronogrammes dérivables de l'ordonnancement logique de la figure 4.10.

L'inconvénient d'ordonnancer les communications de cette façon est que l'on ne peut pas prévoir le recouvrement calcul/communication, à moins de vérifier pour tous les couples possibles (calcul_k, $\text{communication}_{k'}$) s'ils s'exécutent en même temps ou non (cf. les trois chronogrammes de la figure 4.11).

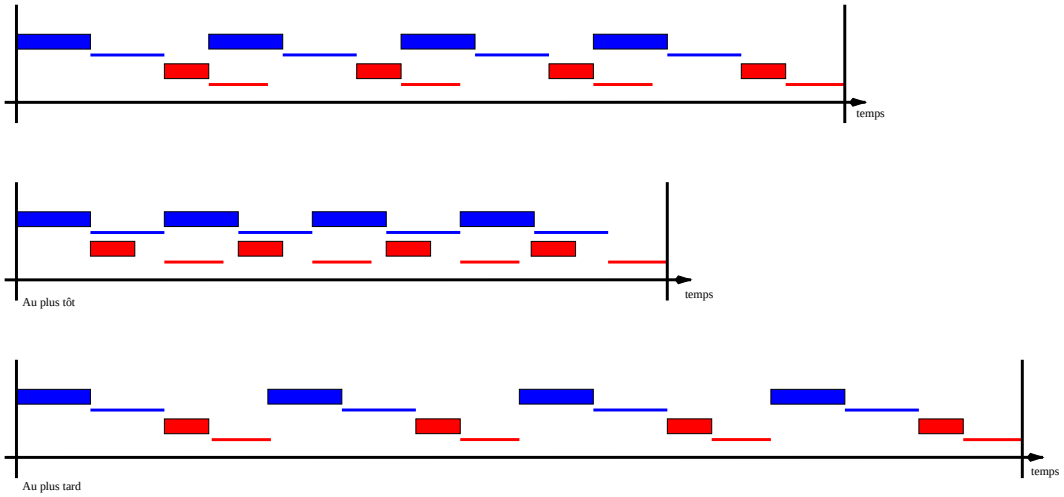
De ce fait, nous ne pouvons que définir un intervalle dans lequel la durée T^{s}_{exec} se trouve. Les deux situations extrêmes sont :

1. lorsqu'aucun calcul et aucune communication ne sont recouverts, à chaque instant une seule ressource est utilisée : pas de recouvrement (borne supérieure) ;
2. lorsque tous les calculs et toutes les communications sont recouverts, c'est-à-dire lorsque toutes les ressources sont tout le temps utilisées : parfait recouvrement (borne inférieure).

La durées physique T^{s}_{exec} d'un étage de pipeline s est bornée par :

$$
\begin{aligned}
T^{\mathrm{s}}_{exec} &\geqslant \max \left[\begin{array}{l} \sum_{k\in\{nb|\ \mathrm{Stage}(nb)=s\}} \left(\mathrm{nbc}^k(\alpha) \times T_{comm}(k)\right), \\ \sum_{k\in\{nb|\ \mathrm{Stage}(nb)=s\}} \left(\mathrm{nbc}^k(\alpha) \times T_{cal}(k)\right) \end{array} \right] \\
T^{\mathrm{s}}_{exec} &\leqslant \sum_{k\in\{nb|\ \mathrm{Stage}(nb)=s\}} \left[\mathrm{nbc}^k(\alpha) \times (T_{comm}(k) + T_{cal}(k))\right]
\end{aligned}
\tag{4.34}
$$

FIG. 4.11 – Trois exemples de chronogrammes en temps physique d'un étage de pipeline : ϵ(c) = d(c)+1

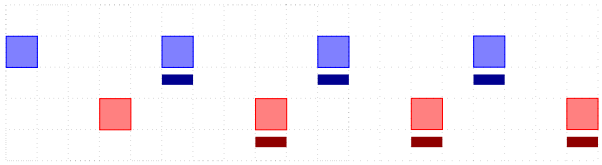


L'échelle de temps est la même pour les trois chronogrammes. Notons tout de même que le chronogramme *au plus tard* est tel qu'à chaque instant s'exécute un calcul ou une communication : il n'y a pas de temps mort.

Pour garantir un résultat sur la latence, seule la borne supérieure de cet intervalle est exploitable. Malheureusement, elle ne permet pas de prendre en compte le recouvrement calculs/communications.

2.3.2.2 Communication i et calcul (i+1) simultanés : $\epsilon^k(c^k) = \mathrm{d}^k(c^k+1)$ Une façon de savoir statiquement quel calcul recouvre quelle communication (ou l'inverse) est de décaler les communications d'un cycle logique par rapport au calcul dont elles dépendent. C'est-à-dire $\epsilon(c^k) = d(c^k+1)$: la communication associée au bloc de calcul c^k s'effectue en même temps que le bloc de calcul c^k+1. On obtient alors un ordonnancement événementiel ressemblant à celui de la figure 4.12. Les contraintes (4.29) (4.30) (4.31) sont toujours respectées.

FIG. 4.12 – Ordonnancement des communications tel que ϵ(c) = d(c+1)



Échelle de temps logique.

Il est alors plus facile de calculer la durée $\mathrm{T}^{\mathrm{s}}_{Exec}$ de l'étage de pipeline s. En effet, entre deux synchronisations globales successives, un étage de pipeline se découpe en trois parties : le prologue, le régime permanent et l'épilogue.

Le prologue est le premier bloc de calcul d'un nid de boucle dans son étage de pipeline (non associé à une communication). Par opposition à l'épilogue qui est la dernière communication d'un nid de boucle dans son étage de pipeline. Le régime permanent est ce qui se passe entre les deux. La figure 4.13 illustre ces propos. Comme deux blocs de calculs de deux nids de boucles différents ne peuvent s'exécuter en même temps (tout comme les communications), nous obtenons la contrainte suivante :

$$\mathrm{T}^{\mathrm{s}}_{Exec} = \sum_{k\in\{nb|\ \mathrm{Stage}(nb)=s\}} (T_{cal}(k) + Permanent_k + T_{comm}(k)) \quad (4.35)$$

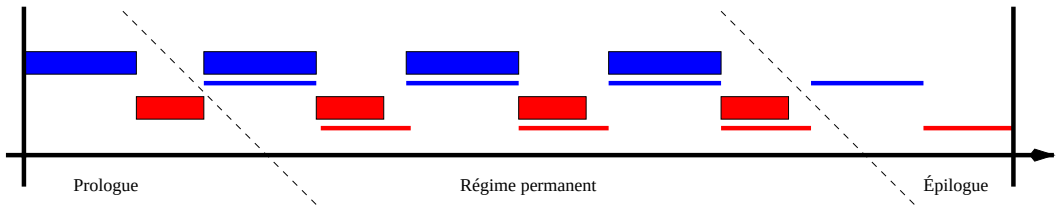
$$Permanent_k = \left(\mathrm{nbc}^k(\alpha) - 1\right) \times \max(T_{cal}(k), T_{comm}(k)) \quad (4.36)$$

$Permanent_k$ est la durée physique du régime permanent pour la tâche k.

Ce qui est équivalent à

✓ $$\mathrm{T}^{\mathrm{s}}_{Exec} = \sum_{k\in\{nb|\ \mathrm{Stage}(nb)=s\}} \left(\mathrm{nbc}^k(\alpha) \times \max(T_{cal}(k), T_{comm}(k)) + \min(T_{cal}(k), T_{comm}(k))\right) \quad (4.37)$$

FIG. 4.13 – Chronogramme en temps physique d'un étage de pipeline : $\epsilon(c) = d(c+1)$



2.3.3 Les durées physiques élémentaires

2.3.3.1 Durée d'un bloc de calculs La durée $T_{cal}(k)$ d'un bloc de calculs d'un nid de boucles k est le produit de la durée élémentaire d'une itération, $T^k_{cal_{elem}}$, et de la taille du bloc de calcul (*i.e.*, le nombre d'itérations dans un bloc), $\det(L^k)$:

✓ $$T_{cal}(k) = \det(L^k).T^k_{cal_{elem}} \quad (4.38)$$

2.3.3.2 Durée d'un bloc de communications Pour calculer la durée d'une communication, il faut connaître le volume de données à transmettre. Ce volume est constant quel que soit le mode de transmission choisi (cf. p. 2.2.4.2) et est donné par :

$$\mathcal{V}^k_{\mathrm{ecr}} = \det(L) \times \prod(m_{max}) \quad (4.39)$$

Durée de la communication :

$$T_{comm}(k) \leqslant \left\lceil \frac{\mathcal{V}^k_{\text{ecr}} \times \text{sizeOfData}^k_{\text{ecr}} \times \text{NbPEMax}_s}{\text{BP}_{s,s+1}} \right\rceil + \text{offsetComm} \tag{4.40}$$

où $\text{sizeOfData}^k_{\text{ecr}}$ désigne la taille en octets d'une donnée stockée dans le tableau ecr du nid de boucles k, offsetComm est le délai d'initialisation d'une communication et $\text{BP}_{s,s+1}$ est la bande passante de l'étage s à l'étage $s+1$. C'est-à-dire que la durée de la communication est évaluée comme étant la somme de la durée effective du transfert de données avec la durée d'initialisation de ce transfert. Selon le volume de données à communiquer, ce temps d'initialisation peut-être prépondérant.

2.3.4 Débit

Comme on l'a montré sur un exemple dans la section 2.1.2, le fonctionnement M-SPMD est plutôt destiné à optimiser le débit et l'efficacité que la latence. Si υ^e représente la période d'acquisition des données d'une tâche e, alors la durée physique d'une période est obtenue par :

$$\max_s \left(\text{T}^{\text{s}}_{exec}\right) = \upsilon^e \frac{\alpha}{\alpha^e_0} \tag{4.41}$$

Comme le débit en entrée est souvent fixé par les spécifications de l'application, il faut essayer d'optimiser l'efficacité de l'utilisation des processeurs en équilibrant les puissances disponibles dans chaque étage.

2.3.5 Conclusion

Ces contraintes se traduisent directement en PPC. Cependant, étant donnée leur complexité, leur potentiel de propagation est quasi-nul. En effet, les contraintes non linéaires et n-aires ne permettent généralement pas de déduire grand chose tant que les variables qui les composent sont encore nombreuses à ne pas être instanciées. Il faut pour cela passer par une modélisation beaucoup plus *contrainte* que mathématique (par l'utilisation des contraintes globales).

Elles permettent malgré tout de fournir une première solution (respectant donc, par nature de la PPC, toutes les contraintes posées) pour le placement sur machine M-SPMD sous contraintes de temps et de débit, car elles donnent des minorants pour les coûts des différentes solutions pendant la recherche.

2.4 Modèle machine M-SPMD

Une machine M-SPMD est constituée de processeurs élémentaires homogènes, partitionnés en une suite ordonnée d'étages logiques de pipeline. Suivant les hypothèses, les communications ont lieu par bus. Il y a un bus entre chaque étage.

On suppose que les entrées s'effectuent par un bus entrant dans le premier étage de pipeline et que les sorties s'effectuent par un bus sortant du dernier étage de pipeline.

2.4.1 Processeurs élémentaires

Les processeurs élémentaires sont supposés homogènes. Ils ont tous le même capacité de mémoire disponible.

Le coût d'un processeur n'est pas supposé dépendre du volume de mémoire utilisé. Il suffit que la solution de placement proposée tienne dans le volume disponible. L'efficacité d'une solution ne fait donc pas intervenir de coût mémoire.

Différentes formules de temps de calcul d'un bloc peuvent être utilisées.

Si le temps de calcul est proportionnel à la taille du bloc, $\det(L)$, le temps de calcul est minimisé en choisissant $L = Id$. Cela reste vrai si le temps de communication est aussi affine.

Le temps de calcul peut aussi être affine par rapport à la taille du bloc. Dans la mesure où les bornes de boucles sont exactement divisibles par les coefficients diagonaux de la matrice $L.P$ et dans la mesure où le temps de communication est aussi affine par rapport au volume et s'il n'y a pas recouvrement entre les calculs et les communications, il faut alors choisir L aussi grand que possible.

Le surcoût à l'origine peut modéliser un surcoût de contrôle, un surcoût de transfert de code et/ou bien un surcoût de pipelinage des itérations d'un bloc.

La prise en compte d'une hiérarchie mémoire n'est pas faite dans ce travail.

2.4.2 Étage de pipeline

Le partitionnement est contraint par les ressources de la machine parallèle cible. En SIMD / SPMD, cette contrainte est la suivante : $\max_{nb \in \mathrm{NB}}(\det(P^{nb})) \leqslant \mathrm{NbPEMax}$. C'est-à-dire que le nombre de processeurs utilisés par chaque nid de boucle, *i.e.*, $\det\left(P^{nb}\right)$, doit être inférieur ou égal au nombre de processeurs dont dispose la machine, NbPEMax.

La considération des étages de pipeline rend local à chacun le partitionnement des nids de boucles. Aussi, $\det(P^{nb})$ représente le nombre de processeurs de l'étage de pipeline s utilisé par le nid de boucle nb. Puisqu'un processeur n'appartient pas à deux étages différents (hyp. 3), si $\mathrm{NbPEMax}_s$ est le nombre de processeurs attribués à l'étage de pipeline s, on a la contrainte de ressource suivante :

✓

$$\forall \mathrm{s}, \forall nb \in \{t |\ \mathrm{Stage}(t) = \mathrm{s}\}, \quad \det(P^{nb}) \ \leqslant \ \mathrm{NbPEMax}_s \tag{4.42}$$

Ces contraintes permettent de ne pas dépasser la quantité de ressources de calcul disponibles sur chaque étage du pipeline. Sachant que la quantité totale de ressources de calcul est bornée également :

$$\sum_{\mathrm{s}=1}^{\mathrm{NBS}} \mathrm{NbPEMax}_s \ \leqslant \ \mathrm{NbPEMax} \tag{4.43}$$

2.4.3 Communications

Chaque étage communique uniquement avec le suivant (hypothèses 4b et 4c).

Les bandes passantes sont définies indépendamment pour chaque étage, sauf pour le dernier. Les bandes passantes sont exprimées en octets par seconde.

La bande passante disponible par processeur dépend du nombre de processeurs affectés à l´étage émetteur. Suivant les possibilités du réseau d'interconnexion, on utilise soit le nombre de processeurs affectés à l'étage, soit le nombre de processeurs utilisés par la tâche émettrice (hypothèse 11).

La communication peut être ralentie par les émetteurs ou les récepteurs dans la mesure où le réseau d'interconnexion ne supporte pas la diffusion.

La durée d'une communication dépend donc du volume émis, du nombre d'émetteurs, du volume reçu et du nombre de récepteurs.

On considère que le volume émis est égal au volume calculé bien que certaines données calculées puissent ne jamais être utilisées et bien que les mêmes données puissent être émises plusieurs fois par un émetteur vers plusieurs récepteurs.

Par exemple, suite à une FFT, on peut souhaiter ne traiter qu'une plage de fréquence bien que le motif propre à la FFT produise 256 valeurs. On sur-approxime donc le volume émis.

Si on calcule une valeur moyenne dans un étage de pipeline et qu'on veuille l'utiliser comme seuil dans l'étage suivant, il faut la diffuser. On sera donc limité par la bande passante du côté des récepteurs.

Le volume émis par un étage pour un tableau est donc inférieur à

$$p^w p^r \det(L^w) \det(M^w_{in})$$

parce que la même donnée n'est pas envoyée deux fois au même processeur.

Le volume reçu par un étage pour un tableau est inférieur à

$$p^r \det(L^r) \det(M^r_{in})$$

parce qu'il peut y avoir du recouvrement entre motifs en partie droite mais pas de réceptions multiples contrairement à ce qui se passe en émission.

Comme toutes les données émises sont reçues et vice-versa, ces deux volumes sont égaux.

Il ne peut pas y avoir d'héritage de données sur un processeur contrairement aux cas traités par [32] parce qu'un nid de boucles n'est traité que sur un étage et parce que les sous-ensembles de processeurs associés à chaque étage sont disjoints.

Pour raffiner l'approximation des volumes échangés, il faudrait être capable de détecter les patterns de communications. Par exemple, en cas de bijection entre les émetteurs et les récepteurs, la contrainte sur les volumes émis peut être durcie en

$$p^w \det(L^w) \det(M^w_{in})$$

.

2.4.4 Entrées-sorties

Les entrées-sorties sont assimilées à des communications entre étages de pipeline. Les débits sont spécifiés comme pour les autres bus. On suppose que ces débits sont compatibles avec les mécanismes d'entrées et les mécanismes de sorties et donc qu'on peut ignorer ces derniers. Il ne reste donc que les contraintes entrantes des communications pour les processeurs utilisant des entrées et les contraintes sortantes des communications pour les processeurs produisant des résultats.

Sous ces hypothèses, les sorties produites dans des étages de pipeline intermédiaires devraient être propagées jusqu'au dernier étage de pipeline. On ignore ce phénomène en ne créant pas de tâches de copies supplémentaires et en ne prenant en compte ni les communications ni les stockages intermédiaires nécessaires.

2.5 Conclusion

Nous avons montré que le modèle SPMD reste un cas particulier de notre M-SPMD[2]. Aussi, une solution SPMD est également une solution M-SPMD dans laquelle il n'y a qu'un seul étage de pipeline. Et si cette solution est meilleure, alors elle sera retenue.

Les modèles MIMD et M-SPMD ne sont pas les mêmes. Le second est moins permissif que le premier. Une première étape pour passer du M-SPMD au MIMD serait de partitionner de manière fixe l'ensemble des processeurs (pour simplifier l'allocation) mais d'autoriser les exécutions parallèles entre partitions.

[2]Dans la mesure où l'hypothèse 4a et la contrainte sur les communications intra étage de pipeline sont relâchées

Chapitre 5

Vers une modélisation plus fine

Le chapitre précédent présentant notre modèle de machine M-SPMD fait intervenir plusieurs notions sur lesquelles nous avons effectuées des approximations. Ces dernières sont principalement dues au fait qu'il n'est pas possible, dans un contexte d'utilisation de la PPC pour le placement, d'utiliser des définitions en extension. Ce chapitre propose donc quelques affinements des modèles correspondant à ces notions approximées.

Tout d'abord, nous évaluons plus finement le volume mémoire nécessaire à l'exécution d'un bloc de calcul (sec. 1). C'est le volume mémoire élémentaire servant de base au calcul de volume nécessaire à l'ensemble. Il est important d'en avoir une expression plus proche de la réalité pour que les approximations effectuées par ailleurs ne rendent pas l'ensemble grossier.

Un autre point important amélioré est le modèle des dépendances. Nous avons montré dans le chapitre 2 (sec. 3 page 46) qu'il n'est pas possible d'écrire en extension l'ensemble des contraintes décrivant les dépendances entre blocs de calculs. Or la définition donnée du polyèdre de dépendances en introduit de fausses qui éliminent des ordonnancements pourtant valides. La section 2 de ce chapitre présente différentes méthodes pour tenter de les éliminer.

Les deux sections suivantes découlent directement des travaux effectués sur le modèle M-SPMD. La première des deux (page 122) concerne la modélisation des entrée-sorties. Elle permet une meilleure caractérisation de la mémoire, de la latence et du débit. La deuxième (page 126) présente la détection des communications entre processeurs à partir du partitionnement des nids de boucles. Cette détection permet l'attribution d'un étage de pipeline à un nid de boucles. Cependant cette section est indépendante du modèle de programmation choisi (SPMD, SIMD, M-SPMD) puisque l'absence d'un lien de communications entre deux processeurs est conditionnée par la présence sur l'un des deux des données utiles au calcul dont le second a la charge.

1 Volume mémoire élémentaire utilisé par un bloc de calcul

La méthode exposée ci-après fait suite aux travaux décrits dans [32] sur la définition d'un modèle capacitif de la mémoire. Plus précisément, nous définissons un modèle de mémoire locale pour une itération c d'une tâche. Nous commençons par décrire ce que nous appelons *le volume de données de référence*, puis nous l'exploitons pour donner l'expression du volume total de mémoire nécessaire à l'exécution d'un nid de boucle partitionné. Ce dernier est, dans certains cas, une sur-approximation de la quantité réellement nécessaire.

1.1 Volume de données de référence

Ce volume correspond, pour un tableau donné, au nombre maximal d'éléments distincts utilisés par un bloc de calcul, en lecture ou en écriture. Nous allons déterminer ce volume en étudiant l'injectivité de la fonction d'accès.

1.1.1 Définition du problème.

Considérons une tâche $\mathcal{T}$. Nous noterons n_e la profondeur de son nid de boucles externe, n_i^A la profondeur du nid de boucles interne (définissant le motif d'un tableau A accédé par $\mathcal{T}$). Le vecteur d'itérations externe sera $\mathbf{i}$ et le vecteur d'itérations interne sera $\mathbf{m}$. Les matrices de pavage et d'ajustage du tableau A sont :

$$\Omega_{pav}^A = \begin{pmatrix} \omega_{11}^{pav} & \cdots & \omega_{1n_e}^{pav} \\ \vdots & \ddots & \vdots \\ \omega_{n1}^{pav} & \cdots & \omega_{nn_e}^{pav} \end{pmatrix} \quad \Omega_{fit}^A = \begin{pmatrix} \omega_{11}^{fit} & \cdots & \omega_{1n_i}^{fit} \\ \vdots & \ddots & \vdots \\ \omega_{n1}^{fit} & \cdots & \omega_{nn_i}^{fit} \end{pmatrix}$$

Elles définissent la fonction d'accès à ces données. Si φ est le vecteur d'accès, alors

$$\varphi = \left(\Omega_{pav}^A.\mathbf{i} + \Omega_{fit}^A.\mathbf{m} + \mathcal{K}\right) \bmod \Phi$$

avec $\mathcal{K}$ le vecteur de translation et Φ le vecteur de taille de chaque dimension du tableau accédé par cette fonction. Pour simplifier les calculs nous éliminons le modulo, tout en gardant à l'esprit qu'il s'agit d'une sur-approximation. Comme nous nous intéressons à un calcul de volume, nous pouvons encore simplifier en prenant $\mathcal{K} = \vec{0}$, puisque la translation conserve les volumes. La fonction d'accès est réduite en :

$$\varphi = \Omega_{pav}^A.\mathbf{i} + \Omega_{pav}^A.\mathbf{m} \tag{5.1}$$

L'introduction du partitionnement 3D (chap. 2 sec. 3.1) change (5.1) en :

$$\varphi = \Omega_{pav}^A.\left(L.P.\mathbf{c} + L.\mathbf{p} + \mathbf{l}\right) + \Omega_{fit}^A.\mathbf{m}$$

La relation liant une donnée d'un tableau avec le vecteur d'accès φ permettant d'accéder à celle-ci est une bijection : à une donnée correspond un unique vecteur d'accès, à un vecteur

d'accès correspond une unique donnée. Aussi, évaluer le volume de données nécessaire à un calcul revient à évaluer le nombre de vecteurs d'accès φ distincts générés pour ce calcul.

Par conséquent, le volume de données de référence du tableau A nécessaire à l'exécution de la tâche $\mathcal{T}$ à l'instant logique $\mathbf{c}$ sur la partition $\mathbf{p}$ est donné par :

$$\begin{aligned} \mathcal{V}^A_{ref}(\mathbf{c}, \mathbf{p}) &= card\left(\{\varphi \mid \varphi \in \Phi, \exists \mathbf{l} \in \mathfrak{D}_l, \exists \mathbf{m} \in \mathfrak{D}_m, \varphi = \Omega^A_{pav}.(L.P.\mathbf{c} + L.\mathbf{p} + \mathbf{l}) + \Omega^A_{pav}.\mathbf{m}\}\right) \\ \text{avec } \mathfrak{D}_l &= [0, L_{11} - 1] \times \cdots \times [0, L_{n_e n_e} - 1] \\ \mathfrak{D}_m &= [0, \mathbf{m}_{max_1} - 1] \times \cdots \times [0, \mathbf{m}_{max_{n_i}} - 1] \end{aligned}$$

Le problème est de calculer ce cardinal.

1.1.2 De l'injectivité de la fonction d'accès

Comme à chaque instant tous les processeurs exécutent le même calcul, la quantité de données utiles à une tâche à l'instant logique 0 sur la partition 0 est la même que celle utilisée par la même tâche sur une autre partition à un autre moment[1]. D'où $\forall(\mathbf{c}, \mathbf{p}), \mathcal{V}_{ref}(\mathbf{c}, \mathbf{p}) \leqslant \mathcal{V}_{ref}(0, 0)$, c'est-à-dire :

$$\mathcal{V}_{ref}(0, 0) = card\left(\{\varphi \mid \varphi \in \Phi, \exists \mathbf{l} \in \mathfrak{D}_l, \exists \mathbf{m} \in \mathfrak{D}_m, \varphi = \Omega^A_{pav}.\mathbf{l} + \Omega^A_{fit}.\mathbf{m}\}\right)$$

L'espace d'itération $\mathfrak{D}_l \times \mathfrak{D}_m$ définit un hyper-parallélogramme. Comme à chaque itération on effectue un calcul, l'idée intuitive pour calculer $\mathcal{V}_{ref}(0, 0)$ est de calculer le volume de cet hyper-parallélogramme, à savoir

$$\prod_{k=1}^{n_e} L_{kk} . \prod_{j=1}^{n_i} \mathbf{m}_{max_j} \tag{5.2}$$

Ce calcul ne convient pas car il ne tient pas compte des éventuels doublons, i.e. des références multiples à une même donnée.

Considérons, par exemple, deux boucles imbriquées, dont les indices sont respectivement i et m, et un tableau T à une dimension. Supposons que la fonction d'accès aux données de T soit $i + m$ et que le partitionnement donne $P = (1)$, $L = (i_{max})$, alors si $i \in [0, 2]$ et $m \in [0, 1]$ la formule précédente calcule $(2 + 1) \times (1 + 1) = 6$ données, ce qui est faux puisque l'on n'accède qu'à 4 données ($T[0]$, $T[1]$, $T[2]$, $T[3]$). L'erreur est de supposer qu'à chaque itération d'un calcul élémentaire on utilise de nouvelles données. Ce n'est vrai que si Ω^A est injective. Du fait de l'assignation unique, la fonction d'accès du membre gauche l'est toujours. Ce n'est pas le cas pour le membre droit (utilisation d'une fenêtre glissante par exemple).

À quelles conditions Ω^A n'est-elle pas injective ? Par définition de l'injectivité, on a :

$$(\Omega^A \text{ n'est pas injective}) \Leftrightarrow \left(\begin{array}{c} \exists \varphi \in \Phi, \exists \left((\mathbf{l}_a, \mathbf{m}_a), (\mathbf{l}_b, \mathbf{m}_b)\right) \in (\mathfrak{D}_l \times \mathfrak{D}_m)^2, (\mathbf{l}_a, \mathbf{m}_a) \neq (\mathbf{l}_b, \mathbf{m}_b), \\ \Omega^A . \begin{pmatrix} \mathbf{l}_a \\ \mathbf{m}_a \end{pmatrix} = \Omega^A . \begin{pmatrix} \mathbf{l}_b \\ \mathbf{m}_b \end{pmatrix} = \varphi \end{array} \right)$$

[1] Dans le cadre SPMD, la dernière partition est susceptible de faire moins de calculs que les autres. Aussi, ce volume majore le volume mémoire nécessaire à la dernière partition.

Soit à résoudre :

$$\Omega^A.\begin{pmatrix}\mathbf{l}_a-\mathbf{l}_b\\ \mathbf{m}_a-\mathbf{m}_b\end{pmatrix}=0 \qquad ((\mathbf{l}_a,\,\mathbf{m}_a),\,(\mathbf{l}_b,\,\mathbf{m}_b))\in(\mathfrak{D}_l\times\mathfrak{D}_m)^2 \tag{5.3}$$

Ce qui donne le système d'équations suivant à résoudre (les termes ω_{j,k_j} désignent les éléments de la diagonale de la matrice correspondante, à la permutation près) :

$$\left\{\begin{array}{c}\omega^{pav}_{1,k_1}\times\left(\mathbf{l}_{a_{k_1}}-\mathbf{l}_{b_{k_1}}\right)+\omega^{fit}_{1,k'_1}\times\left(\mathbf{m}_{a_{k'_1}}-\mathbf{m}_{b_{k'_1}}\right)=0\\ \omega^{pav}_{2,k_2}\times\left(\mathbf{l}_{a_{k_2}}-\mathbf{l}_{b_{k_2}}\right)+\omega^{fit}_{2,k'_2}\times\left(\mathbf{m}_{a_{k'_2}}-\mathbf{m}_{b_{k'_1}}\right)=0\\ \vdots\\ \omega^{pav}_{n,k_n}\times\left(\mathbf{l}_{a_{k_n}}-\mathbf{l}_{b_{k_n}}\right)+\omega^{fit}_{n,k'_n}\times\left(\mathbf{m}_{a_{k'_n}}-\mathbf{m}_{b_{k'_1}}\right)=0\\ \text{avec }\forall j\in[1,\,n]\;k_j\in[1,\,n_e]\;\text{ et }\;k'_j\in[1,\,n_i]\end{array}\right. \tag{5.4}$$

Ces équations sont linéairement indépendantes puisque Ω^A_{pav} et Ω^A_{fit} sont diagonales à une permutation près (sec. 1 p. 4), toutes de la forme $a.x+b.y=0$. On peut, par conséquent, réduire le problème à l'étude d'une seule de ces équations. Considérons alors la j^{ieme} telle que $\left(\mathbf{l}_{a_{k_j}},\,\mathbf{m}_{a_{k'_j}}\right)$ et $\left(\mathbf{l}_{b_{k_j}},\,\mathbf{m}_{b_{k'_j}}\right)$ sont distincts. On a donc :

$$\omega^{pav}_{j,k_j}.\left(\mathbf{l}_{a_{k_j}}-\mathbf{l}_{b_{k_j}}\right)+\omega^{fit}_{j,k'_j}.\left(\mathbf{m}_{a_{k'_j}}-\mathbf{m}_{b_{k'_j}}\right)=0 \tag{5.5}$$

Or,

- le cas $\left(\left(\mathbf{l}_{a_{k_j}}-\mathbf{l}_{b_{k_j}}=0\right)\wedge\left(\mathbf{m}_{a_{k'_j}}-\mathbf{m}_{b_{k'_j}}=0\right)\right)$ n'est pas possible, puisque les couples $\left(\mathbf{l}_{a_{k_j}},\,\mathbf{m}_{a_{k'_j}}\right)$ et $\left(\mathbf{l}_{b_{k_j}},\,\mathbf{m}_{b_{k'_j}}\right)$ sont distincts ;
- $\left(\omega^{fit}_{j,k'_j}=0\wedge\omega^{pav}_{j,k_j}=0\right)$ signifie (puisqu'il y a autant d'équations que le tableau a de dimensions) que la j^{ieme} dimension du tableau A n'est référencée que par une unique valeur qui est 0. Le nombre d'accès distincts sur cette dimension vaut alors 1.

Nous ne considérons désormais que le cas où $\omega^{pav}_{j,k_j}\neq 0$ et $\omega^{fit}_{j,k'_j}\neq 0$. Puisque nous avons supposé qu'il n'y avait pas d'accès modulo, si un seul de ces deux termes est nul on a alors :

- si $\omega^{pav}_{j,k_j}=0$, le nombre d'accès distincts sur la dimension j vaut $\mathbf{m}_{max_{k'_j}}$;
- si $\omega^{fit}_{j,k'_j}=0$, le nombre d'accès distincts sur la dimension j vaut $L_{k_jk_j}$.

1.1.3 Caractérisation des accès redondants à une donnée dans un tableau.

Soit la droite δ d'équation $y=-\frac{\omega^{pav}_{j,k_j}}{\omega^{fit}_{j,k'_j}}x$. Les points de δ qui nous intéressent sont les points $(x,\,y)\in\mathbb{Z}^2$. Une solution triviale est $x_s=\omega^{fit}_{j,k'_j}$ et $y_s=-\omega^{pav}_{j,k_j}$. Si x_s et y_s ne sont pas premier entre eux, considérons alors $x'_s=\dfrac{\omega^{fit}_{j,k'_j}}{\mathrm{pgcd}\left(\omega^{pav}_{j,k_j},\omega^{fit}_{j,k'_j}\right)}$ et $y'_s=-\dfrac{\omega^{pav}_{j,k_j}}{\mathrm{pgcd}\left(\omega^{pav}_{j,k_j},\omega^{fit}_{j,k'_j}\right)}$. Le point de coordonnées (x'_s,y'_s) est le plus proche de l'origine et générateur des autres. En effet, en multipliant ses coordonnées par k quelconque de $\mathbb{Z}^*$ jusqu'à sortir du domaine, on trouve tous les autres points. Deux cas de figures se présentent alors à nous (fig. 5.1 p.111) :

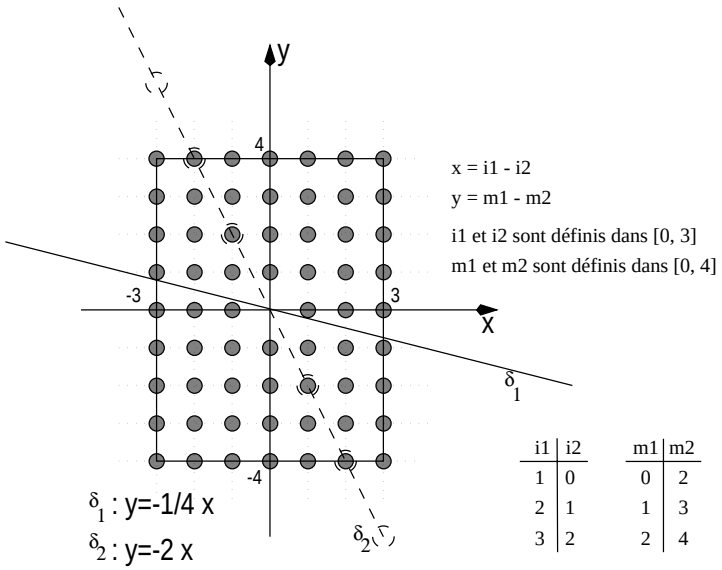
1. $(x'_s,\, y'_s) \notin \left[-L_{k_jk_j}+1,\, L_{k_jk_j}-1\right] \times \left[-\mathbf{m}_{max_{k'_j}}+1,\, \mathbf{m}_{max_{k'_j}}-1\right]$
2. $(x'_s,\, y'_s) \in \left[-L_{k_jk_j}+1,\, L_{k_jk_j}-1\right] \times \left[-\mathbf{m}_{max_{k'_j}}+1,\, \mathbf{m}_{max_{k'_j}}-1\right]$

Le premier cas assure qu'il n'y a pas de solution entière dans
$\left[-L_{k_jk_j}+1,\, L_{k_jk_j}-1\right] \times \left[-\mathbf{m}_{max_{k'_j}}+1,\, \mathbf{m}_{max_{k'_j}}-1\right]$ et par conséquent le nombre d'accès distincts au tableau A sur la dimension j a pour valeur :

$$L_{k_jk_j} \times \mathbf{m}_{max_{k'_j}} \tag{5.6}$$

Le second cas indique qu'il existe des valeurs pour lesquelles deux itérations distinctes accèdent à la même donnée. Calculons le nombre de ces accès récurrents.

FIG. 5.1 – Exemple pour deux fonctions d'accès distinctes à un même tableau : $\Omega_1 = (1 \quad 4)$ et $\Omega_2 = (2 \quad 1)$.



Du point de coordonnées (1, -2), on déduit les accès redondants: (i1=1, m1= 0) et (i2 = 0, m2 = 2) accèdent à la même donnée référencée par la valeur 2. Ainsi, tous les couples que l'on peut faire tels que x=1 et y=-2 sont des accès redondants. Dans cet exemple il y en a 9.

On a :

$$\left((x_s,\, y_s) \in \left[-L_{k_jk_j}+1,\, L_{k_jk_j}-1\right] \times \left[-\mathbf{m}_{max_{k'_j}}+1,\, \mathbf{m}_{max_{k'_j}}-1\right]\right)$$
$$\Updownarrow$$
$$\left(\begin{array}{c} \exists \left(\left(\mathbf{l}_{a_{k_j}},\, \mathbf{m}_{a_{k'_j}}\right),\, \left(\mathbf{l}_{b_{k_j}},\, \mathbf{m}_{b_{k'_j}}\right)\right) \in \left(\left[0,\, L_{k_jk_j}-1\right] \times \left[0,\, \mathbf{m}_{max_{k'_j}}-1\right]\right)^2, \\ \omega^{pav}_{j,k_j}\left(\mathbf{l}_{a_{k_j}}-\mathbf{l}_{b_{k_j}}\right)+\omega^{fit}_{j,k'_j}\left(\mathbf{m}_{a_{k_j}}-\mathbf{m}_{b_{k_j}}\right)=0 \end{array}\right)$$

On cherche donc à calculer

$$\mathcal{C}\left(x'_s,\, y'_s\right) = card\left(\left\{ \begin{array}{c} \left((\mathbf{l}_a,\, \mathbf{m}_a)\,,\, (l_b,\, m_b)\right) \in \left(\left[0,\, L_{k_jk_j} - 1\right] \times \left[0,\, \mathbf{m}_{max_{k'_j}} - 1\right]\right)^2 \mid \\ \mathbf{l}_{a_{k_j}} - \mathbf{l}_{b_{k_j}} = x'_s,\, \mathbf{m}_{a_{k'_j}} - \mathbf{m}_{b_{k'_j}} = y'_s \end{array} \right\}\right)$$

Or on sait que $\forall d \in \mathbb{N}$, $card\left(\left\{(a,b) \in \left([0,\, X]\right)^2 \mid a - b = d\right\}\right) = \max\left((X - 0 + 1) - |d|\,,\, 0\right)$, d'où

$$\begin{aligned} \mathcal{C}\left(x'_s,\, y'_s\right) &= card\left(\left\{ \begin{array}{c} (\mathbf{l}_a,\, \mathbf{l}_b) \in \left(\left[0,\, L_{k_jk_j} - 1\right]\right)^2 \mid \\ \mathbf{l}_{a_{k_j}} - \mathbf{l}_{b_{k_j}} = x'_s \end{array} \right\}\right).card\left(\left\{ \begin{array}{c} (\mathbf{m}_a,\, \mathbf{m}_b) \in \left(\left[0,\, \mathbf{m}_{max_{k'_j}} - 1\right]\right)^2 \mid \\ \mathbf{m}_{a_{k'_j}} - \mathbf{m}_{b_{k'_j}} = y'_s \end{array} \right\}\right) \\ &= \left(L_{k_jk_j} - |x'_s|\right).\left(\mathbf{m}_{max_{k'_j}} - |y'_s|\right) \end{aligned}$$

Par conséquent le nombre d'accès distincts à la j^{ieme} dimension du tableau $\mathcal{A}$, noté $\mathcal{N}_j^{\mathcal{A}}$, a pour valeur :

$\forall j \in [1,\, n]$,

$$\begin{aligned} \mathcal{N}_j^{\mathcal{A}} &= L_{k_jk_j}.\mathbf{m}_{max_{k'_j}} - \left(L_{k_jk_j} - \frac{\omega_{j,k'_j}^{fit}}{\text{pgcd}\left(\omega_{j,k_j}^{pav},\, \omega_{j,k'_j}^{fit}\right)}\right).\left(\mathbf{m}_{max_{k'_j}} - \frac{\omega_{j,k_j}^{pav}}{\text{pgcd}\left(\omega_{j,k_j}^{pav},\, \omega_{j,k'_j}^{fit}\right)}\right) \\ &= \frac{\omega_{j,k'_j}^{fit}.\mathbf{m}_{max_{k'_j}}}{\text{pgcd}\left(\omega_{j,k_j}^{pav},\, \omega_{j,k'_j}^{fit}\right)} + \frac{\omega_{j,k_j}^{pav}.L_{k_jk_j}}{\text{pgcd}\left(\omega_{j,k_j}^{pav},\, \omega_{j,k'_j}^{fit}\right)} - \left(\frac{\omega_{j,k'_j}^{fit}.\omega_{j,k_j}^{pav}}{\text{pgcd}\left(\omega_{j,k_j}^{pav},\, \omega_{j,k'_j}^{fit}\right).\text{pgcd}\left(\omega_{j,k_j}^{pav},\, \omega_{j,k'_j}^{fit}\right)}\right) \\ &= \frac{\omega_{j,k_j}^{pav}.L_{k_jk_j}}{\text{pgcd}\left(\omega_{j,k_j}^{pav},\, \omega_{j,k'_j}^{fit}\right)} + \frac{\omega_{j,k'_j}^{fit}.\left(\mathbf{m}_{max_{k'_j}} - \frac{\omega_{j,k_j}^{pav}}{\text{pgcd}\left(\omega_{j,k_j}^{pav},\, \omega_{j,k'_j}^{fit}\right)}\right)}{\text{pgcd}\left(\omega_{j,k_j}^{pav},\, \omega_{j,k'_j}^{fit}\right)} \end{aligned} \tag{5.7}$$

Des équations (5.6) et (5.7), nous déduisons l'expression du volume de données de référence pour le tableau $\mathcal{A}$:

$$\mathcal{V}_{ref}^{\mathcal{A}}\left(c,\, p\right) \leqslant \prod_{j=1}^{n} \min\left(\mathcal{N}_j^{\mathcal{A}},\, L_{k_jk_j}.\mathbf{m}_{max_{k'_j}}\right) \tag{5.8}$$

1.2 Volume de données total nécessaire à l'exécution d'un bloc de calcul

À partir de ce volume de référence, nous allons déduire les volume de données nécessaire à l'exécution d'un bloc de calcul d'un nid de boucle. Il s'agit de la somme du volume de données en entrée avec le volume de données en sortie.

1.2.1 Volume de données consommées

Notre tâche $\mathcal{T}$ lit ses données dans N tableaux distincts. Par conséquent, le volume total de données en lecture est la somme des volumes de données accédées en lecture par $\mathcal{T}$ par tableau, i.e., :

$$\mathcal{V}_{lec}^{\mathcal{T}} = \sum_{s=1}^{N} \mathcal{V}_{ref}^{\mathcal{A}_s} \tag{5.9}$$

Ce volume n'est exact que si l'on accède à un tableau grâce à une et une seule fonction d'accès. En effet, dans le cas contraire, le volume de données consommées calculé est une sur-approximation du volume réel puisqu'alors ce type d'accès redondants n'est pas détecté par la méthode présentée ici.

1.2.2 Volume de données produites

Le volume de données en écriture $\mathcal{V}_{ecr}^{\mathcal{T}}$ est plus simple du fait de l'hypothèse de travailler sur des tableaux à assignation unique. Par conséquent, à une itération correspond une donnée. En d'autres mots, il n'y a pas de doublon, et le volume est donné par l'équation 5.2.

1.2.3 Volume de données total

On en déduit alors le volume total (consommé et produit) nécessaire à l'exécution d'un bloc de calcul comme étant la somme du volume de données consommées et du volume de données produites :

$$\mathcal{V}_{total}^{T} = \mathcal{V}_{lec}^{\mathcal{T}} + \mathcal{V}_{ecr}^{\mathcal{T}} \tag{5.10}$$

2 Élimination de la sur-approximation des dépendances

Notre polygone de dépendance n'est pas caractérisé par des inéquations, mais par les sommets issus de leur intersection (2.3.3 p. 46). Ces sommets ont des coordonnées entières ou rationnelles. Malheureusement plus souvent rationnelles. Comme nous ne travaillons qu'en discret, leur utilisation directe n'est pas possible. C'est pourquoi, une relaxation dans les entiers est nécessaire.

Cette relaxation consiste à considérer non pas le polygone exact mais la plus petite enveloppe convexe définie dans $\mathbb{N}^2$ le contenant. Ce sont sur ses sommets générateurs entiers que les contraintes de précédences sont posées. Comme ce sont des fonctions affines, cela garantit qu'elles sont respectées sur l'ensemble des points contenus par cette enveloppe convexe [20].

Or, du fait de cette sur-approximation, nous définissons ainsi plus de points de dépendance qu'il n'en existe réellement. L'effet sur l'ordonnancement des blocs de calcul déduit des contraintes de précédence est immédiat : certains ordonnancements valides sont considérés comme étant des non solutions.

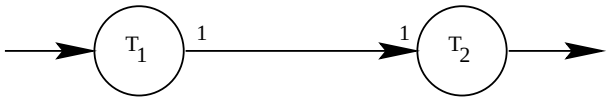
Nous allons voir au travers d'un exemple les différentes méthodes étudiées pour essayer d'éliminer, ou minimiser, cette sur-approximation.

2.1 Description du problème sur un exemple

2.1.1 Un exemple

Considérons une application dans laquelle ne figurent que deux tâches T_1 et T_2. T_1 ne produit qu'une donnée à la fois et T_2 n'en consomme également qu'une à la fois (comme le montre le GFD de la figure 5.2), et ce pour une durée infinie. On a comme fonction d'accès $\Omega^w_{pav} = (1)$, $\Omega^r_{pav} = (1)$ et $\Omega^w_{fit} = (1)$, $\Omega^r_{fit} = (1)$ avec un motif de taille 1.

FIG. 5.2 – Exemple de dépendance faisant apparaître l'approximation.



Supposons que chaque production de T_1 soit consommée immédiatement par T_2. Nous avons alors un ordonnancement du type $T_1T_2T_1T_2T_1...$ Ce qui correspond aux fonctions d'ordonnancement :

$$\begin{array}{lllllll} \alpha^{T_1} & = & (2) & \alpha^{T_2} & = & (2) \\ \beta^{T_1} & = & 0 & \beta^{T_2} & = & 1 \end{array}$$

Si nous représentons les dépendances dans un espace où l'axe des ordonnées représente les itérations donnant un accès en écriture, l'axe des abscisses représente les itérations donnant un accès en lecture, alors les dépendances sont triviales et représentées par la diagonale principale (cf. Fig. 5.3 partie de gauche).

2.1.2 Contraintes de précédences

Dans notre exemple, les sommets générateurs du polygone de dépendances sont au nombre de trois et sont confondus à l'origine. Ces points (A, B et H sur la figure 5.3 page 116) ont respectivement pour coordonnées $(0,0)$, $(x_B,0)$ et $(0,y_H)$ où (pour simplifier, on prend $\mathcal{K}^w$ et $\mathcal{K}^r$ nuls) :

$$x_B = \left\lceil \frac{\Omega^w_{pav}(L^w P^w - 1) + \Omega^w_{fit}(m^w_{max} - 1)}{\Omega^r_{pav} L^r P^r} \right\rceil$$

$$y_H = \left\lceil \frac{\Omega^r_{pav}(L^r P^r - 1) + \Omega^r_{fit}(m^r_{max} - 1)}{\Omega^w_{pav} L^w P^w} \right\rceil$$

Considérons la situation pour laquelle le partitionnement 3D fait que $L^w = P^w = L^r = P^r = 1$. Nous avons alors $A = B = H = (0,0)$. Les contraintes d'ordonnancement (ou de *précédence*) sont :

$$\forall (c^w_{T_1}, c^r_{T_2}), \mathcal{D}\left(c^w_{T_1}, c^r_{T_2}\right) \Rightarrow d^{T_1}(c^w_{T_1}) < d^{T_2}(c^r_{T_2}) \tag{5.11}$$

Les points A, B et H étant des sommets générateurs du polyèdre, on a :

$$\left.\begin{array}{rcl} d^{T_1}(y_A) & < & d^{T_2}(x_A) \\ d^{T_1}(y_B) & < & d^{T_2}(x_B) \\ d^{T_1}(y_H) & < & d^{T_2}(x_H) \end{array}\right\} \quad \Rightarrow \quad (5.11)$$

Par définition des fonctions d'ordonnancement, $d^T(c^T) = \alpha^T.c^T + \beta^T$. De plus, les hypothèses architecturales imposent $\alpha^T = N.\gamma^T$ et $\beta^T = N.\delta^T + k^T$, où N est le nombre de macro-tâches constituant l'application et k^T le numéro de la macro-tâche T. Dans notre exemple, $N = 2$, $k^{T_1} = 0$ et $k^{T_2} = 1$

De ce fait nous avons :

$$\begin{array}{rcl} d^{T_1}(y_A) & < & d^{T_2}(x_A) \\ & \Leftrightarrow & \\ \alpha^{T_1}.y_A + \beta^{T_1} & < & \alpha^{T_2}.x_A + \beta^{T_2} \\ & \Rightarrow & \\ \beta^{T_1} & < & \beta^{T_2} \\ & \Rightarrow & \\ 2.\delta^{T_1} & < & 2.\delta^{T_2} + 1 \end{array} \qquad \begin{array}{rcl} d^{T_1}(y_B) & < & d^{T_2}(x_B) \\ & \Leftrightarrow & \\ \alpha^{T_1}.y_B + \beta^{T_1} & < & \alpha^{T_2}.x_B + \beta^{T_2} \\ & \Rightarrow & \\ \beta^{T_1} & < & \beta^{T_2} \\ & \Leftrightarrow & \\ 2.\delta^{T_1} & < & 2.\delta^{T_2} + 1 \end{array}$$

$$\begin{array}{rcl} d^{T_1}(y_H) & < & d^{T_2}(x_H) \\ & \Leftrightarrow & \\ \alpha^{T_1}.y_H + \beta^{T_1} & < & \alpha^{T_2}.x_H + \beta^{T_2} \\ & \Rightarrow & \\ \beta^{T_1} & < & \beta^{T_2} \\ & \Rightarrow & \\ 2.\delta^{T_1} & < & 2.\delta^{T_2} + 1 \end{array}$$

Ce qui donne le système d'inéquations suivant :

$$\begin{cases} (A) & : & 2.\delta^{T_1} & < & 2.\delta^{T_2}+1 \\ (B) & : & 2.\delta^{T_1} & < & 2.\delta^{T_2}+1 \\ (H) & : & 2.\delta^{T_1} & < & 2.\delta^{T_2}+1 \end{cases} \tag{5.12}$$

Un ordonnancement au plus tôt possible est $\gamma^{T_1} = \gamma^{T_2} = (1)$ et $\delta^{T_1} = \delta^{T_2} = 0$. On obtient les fonctions d'ordonnancement suivantes :

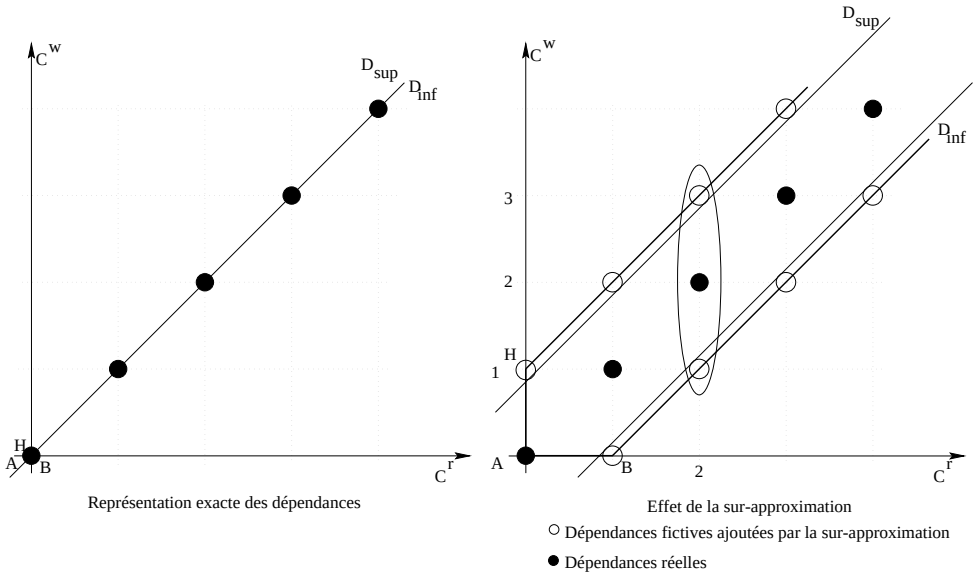
$$\begin{aligned} d^{T_1}(c^{T_1}) &= 2.c^{T_1} \\ d^{T_2}(c^{T_2}) &= 2.c^{T_2}+1 \end{aligned}$$

Nous retrouvons les résultats énoncés plus haut.

2.1.3 Effet de la sur-approximation

Imaginons que nous soyons soumis à des contraintes de débit élevé et que nous disposions de 8 processeurs. Ces 8 processeurs seraient alors utilisés. Quelles sont les conséquences de ce partitionnement, $P^w = P^r = 8$ et $L^w = L^r = 1$, sur notre polyèdre de dépendance ?

FIG. 5.3 – Effet de la sur-approximation



Le calcul des coordonnées des points B et H donne $B = (1, 0)$ et $H = (0, 1)$, ce qui a pour effet de créer des dépendances qui n'existent pas (fig. 5.3 partie de droite).

D'un point de vue graphique, la sur-approximation introduit des points de dépendances fictifs. L'impact sur l'ordonnancement est d'entraîner des décalages dans le temps qui n'ont pas lieu d'être. Dans notre exemple, la troisième itération de la tâche T_2, numérotée 2 sur l'axe des abscisses, doit attendre l'exécution des deuxième, troisième et quatrième itérations

de T_1, numérotées 1, 2 et 3 sur l'axe des ordonnées. Alors qu'en réalité, seule l'itération 2 de T_1 est nécessaire. De ce fait, l'ordonnancement n'est plus $d^{T_1}(c^{T_1}) = 2.c^{T_1}$ et $d^{T_2}(c^{T_2}) = 2.c^{T_2} + 1$ mais $d^{T_1}(c^{T_1}) = 2.c^{T_1}$ et $d^{T_2}(c^{T_2}) = 2.c^{T_2} + 3$.

Cela modifie le problème posé et invalide l'ordonnancement trivial $T_1T_2T_1T_2...$ trouvé précédemment, bien qu'il soit valide pour le partitionnement choisi.

En effet, le système (5.12) devient alors :

$$\begin{cases} (A) & : & 2.\delta^{T_1} & < & 2.\delta^{T_2} + 1 \\ (B) & : & 2.\delta^{T_1} & < & 2.\gamma^{T_2} + 2.\delta^{T_2} + 1 \\ (H) & : & 2.\gamma^{T_1} + 2.\delta^{T_1} & < & 2.\delta^{T_2} + 1 \end{cases} \tag{5.13}$$

La solution trouvée précédemment n'en est plus une, bien que pour le problème initial elle le soit toujours, puisqu'alors la troisième inéquation de (5.13) donne $2 < 1$. Les fonctions d'ordonnancement au plus tôt solutions du nouveau problème sont alors :

$$\begin{array}{lcl} d^{T_1}(c^{T_1}) & = & 2.c^{T_1} \\ d^{T_2}(c^{T_2}) & = & 2.c^{T_2} + 3 \end{array}$$

Elle est valide (puisque l'on a fait une sur-approximation) dans la mesure où il ne s'agit pas de donner LA solution exacte résolvant le problème, mais de garantir la causalité des calculs, ainsi que le respect des contraintes de temps réel et de consommation mémoire. La sur-approximation des dépendances implique cependant une sur-consommation mémoire et une latence plus importante. Par conséquent, s'il existe une solution au problème modifié garantissant ces contraintes, alors il en existe une (meilleure) au problème réel.

Le problème posé par cette approximation apparaît lorsqu'il n'existe pas de solution avec la sur-approximation. Car le problème sur-contraint n'est pas le problème à résoudre. Plus nous pouvons gagner en précision sur la description du polytope de dépendance, et meilleurs sont les résultats finaux.

Plusieurs méthodes ont été étudiées afin de réduire le plus possible cette sur-approximation. La première a été de poser les contraintes de précédences sur les sommets générateurs rationnels du polygone. La deuxième a consisté à chercher une représentation, toujours en intention, exacte des dépendances. La troisième approche a été de faire une simplification entière des équations des droites D_{inf} et D_{sup}. Chacune de ces méthodes est exposée dans la suite.

2.2 Considération des sommets à coordonnées rationnelles

Par définition des droites D_{inf}, D_{sup} et de l'espace d'itération, les sommets générateurs sont définis dans $\mathbb{Q}^2$. Aussi, pourquoi ne pas essayer de résoudre ce sous-problème dans $\mathbb{Q}^2$?

Les coordonnées des sommets générateurs seraient alors de la forme $\begin{pmatrix} \frac{\eta_1}{\zeta_1} \\ \vdots \\ \frac{\eta_n}{\zeta_n} \end{pmatrix}$. La fonction

d'ordonnancement appliquée à un de ces sommets devient :

$$\begin{aligned} d(c) &= \alpha.c + \beta \\ &= (\textstyle\sum_{i=1}^{n} \alpha_i c_i) + \beta \\ &= \left(\textstyle\sum_{i=1}^{n} \alpha_i \frac{\eta_i}{\zeta_i}\right) + \beta \\ &= \frac{\left(\sum_{i=1}^{n} \alpha_i \eta_i \frac{\prod_{j=1}^{n} \zeta_j}{\zeta_i}\right) + \beta \prod_{j=1}^{n} \zeta_j}{\prod_{j=1}^{n} \zeta_j} \end{aligned}$$

Les contraintes d'ordonnancement liées aux dépendances sont des relations d'ordre. Ainsi, pour un hyper-sommet $M = (c^r, c^w)$, la relation $d(c^w) < d(c^r)$ devient :

$$\left(\left(\sum_{i=1}^{n} \alpha_i^w \eta_i^w \frac{\prod_{j=1}^{n} \zeta_j^w}{\zeta_i^w}\right) + \beta^w \prod_{j=1}^{n} \zeta_j^w\right) \prod_{j'=1}^{m} \zeta_{j'}^r < \left(\left(\sum_{i'=1}^{m} \alpha_{i'}^r \eta_{i'}^r \frac{\prod_{j'=1}^{m} \zeta_{j'}^r}{\zeta_{i'}^r}\right) + \beta^r \prod_{j'=1}^{m} \zeta_{j'}^r\right) \prod_{j=1}^{n} \zeta_j^w$$

Cette approche n'est pas bonne pour deux raisons majeures :

1. d'un point de vue des performances de propagation d'une telle relation ;
2. d'un point de vue de la qualité de la solution trouvée.

Du point de vue de la programmation par contraintes, il est évident qu'une telle relation va propager très peu d'information à cause de sa complexité. Ce point a été vérifié lors des essais : le temps de résolution du petit problème de la section 2.1.1), très éloigné en terme de complexité de l'application de référence, dépasse très largement le raisonnable. En imposant le partitionnement, l'outil fournit un résultat. Or il se trouve que cette solution est identique à celle trouvée en considérant la sur-approximation initiale.

Cela s'explique si l'on revient au sens porté par le paramètre c du partitionnement. Ce paramètre représente une unité de temps logique. Cet axe temporel logique n'est pas continu mais *pointillé* : on transite de 1 à 2 sans passer l'ensemble des réels (a fortiori des rationnels) existant entre les deux. L'espace qui sépare deux points consécutifs de cet axe n'a pas de sens. Cela signifie que notre point d'intersection n'existe, ou n'a de sens, que si ses coordonnées sont entières. Dans le cas contraire, par rapport à notre axe de temps logique, il est confondu avec l'instant logique précédent. Autrement dit, le temps logique 0 peut être représenté par l'intervalle de $\mathbb{R}$ $[0, 1[$, 1 par $[1, 2[$, ainsi de suite.

2.3 Caractérisation exacte du polygone de dépendance

Le problème dû à la sur-approximation est éliminé si on élimine la sur-approximation. C'est pourquoi cette première réflexion porte sur la caractérisation du polygone exact[2], c'est-à-dire la recherche de la plus grande enveloppe convexe entière contenue dans ce polygone.

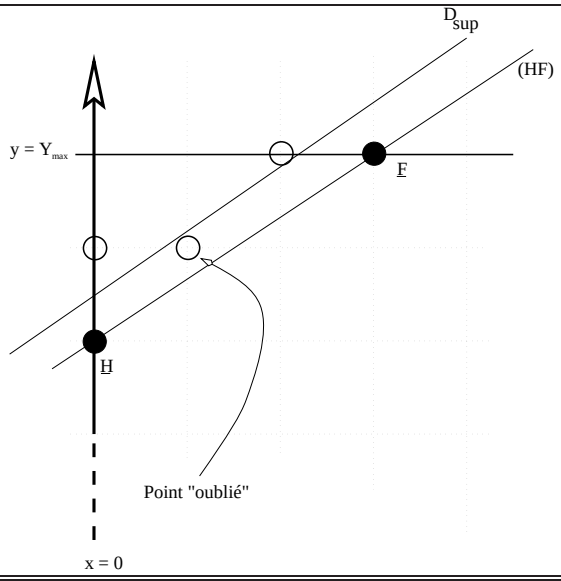
Il s'agit dans un premier temps d'inverser les arrondis des coordonnées définissant les points J, B, D, L, K, F, H, I.

Cela ne suffit pas, car, si l'on considère le voisinage de D_{sup} par exemple, l'espace délimité par les droites D_{sup}, (HF), $x = 0$ et $y = Y_{max}$ peut contenir des points entiers (fig. 5.4).

[2]Par rapport aux différentes approximations faites par ailleurs.

Or ces points **sont** des points de dépendances. Par conséquent, toute «solution» qui en est déduite est une solution fausse. Il faut donc, dans un deuxième temps, prendre en compte suffisamment de points (nouveaux sommets générateurs) de façon à englober tous les points à coordonnées entières caractérisant les dépendances, mais sans en ajouter de fictives.

FIG. 5.4 – De la nécessité d'ajouter des sommets générateurs pour décrire exactement les dépendances.



Se contenter d'inverser les arrondis des coordonnées des sommets générateurs, pour éliminer la sur-approximation, peut donner des résultats faux. Car la dépendance portée par le point «oublié» ne sera alors pas prise en compte. Il faut ajouter des points pour décrire exactement le polygone.

Malheureusement, tous les paramètres étant variables, il est très difficile de prévoir le nombre nécessaire de points à ajouter. De plus, il reste le problème d'en exprimer formellement leurs coordonnées. Nous retrouvons là le problème des coupes de Gomory [28] pour lequel les seules solutions sont algorithmiques (l'algorithme de Chernikova[15], par exemple, est le plus utilisé).

Cependant, Harvey donne un algorithme dans [35] permettant de calculer, dans un espace de dimension 2, les sommets générateurs de la plus grande enveloppe convexe entière d'un polygone convexe (potentiellement non borné). Ce polygone est caractérisé par l'ensemble des inéquations linéaires le délimitant. De plus il montre que la complexité de son algorithme est polynomiale et optimale.

Malgré tout, il ne s'agit pas de la solution miracle pour les mêmes raisons évoquées plus haut. En effet, Harvey est dans un contexte où, pour une inéquation de la forme $a.x+b.y \leqslant c$, les seules inconnues sont x et y. Ce qui n'est pas notre cas, puisque tous les coefficients (x, y, a, b, c) sont inconnus[3].

Nous ne pouvons donc pas décrire exactement le polygone des dépendances. Voyons s'il est possible de réduire, plutôt que d'éliminer, cette sur-approximation.

[3]Tous ces coefficients dépendent principalement du partitionnement 3D et donc évoluent avec lui tout le long de la résolution.

2.4 Simplification entière des inégalités

Une deuxième solution est de forcer les droites D_{sup} et D_{inf} à passer par des points à coordonnées entières et plus précisément, par les points entiers les plus proches à l'intérieur du domaine de dépendance. Les équations des droites D_{inf} et D_{sup} doivent donc avoir des solutions dans $\mathbb{Z}$.

Théorème 5.2-1 : Toute équation d'inconnues (x, y), de la forme $-a.x + b.y = c$, telle que a et b sont des entiers naturels strictement positifs et c un entier quelconque, a des solutions dans $\mathbb{Z}^2$ si et seulement si $\text{pgcd}(a, b)$ divise c, *i.e.* :

$$\forall (a,b) \in (\mathbb{N}^*)^2, \quad \forall c \in \mathbb{Z},$$
$$\left(\exists (x_0, y_0) \in \mathbb{Z}^2,\ -a.x_0 + b.y_0 = c\right) \Leftrightarrow (\text{pgcd}(a,b) \mid c)$$

Preuve :

$(\Leftarrow)$ $(\text{pgcd}(a,b)|c) \Rightarrow \left(\exists (x_0, y_0) \in \mathbb{Z}^2, -a.x_0 + b.y_0 = c\right)$

$\exists k \in \mathbb{Z}, c = k.gcd(a,b)$
Or on sait que
$\exists (u,v) \in \mathbb{Z}^2, \quad -a.u + b.v = \text{pgcd}(a,b)$
$\Leftrightarrow \qquad\qquad -a.k.u + b.k.v = k.\text{pgcd}(a,b) = c$
Il suffit de prendre $x_0 = k.u$ et $y_0 = k.v$

$(\Rightarrow)$ $\left(\exists (x_0, y_0) \in \mathbb{Z}^2, -a.x_0 + b.y_0 = c\right) \Rightarrow (\text{pgcd}(a,b)|c)$

On a $-a.x_0 + b.y_0 = c$
$\text{pgcd}(a,b).(-a'.x_0 + b'.y_0) = \text{pgcd}(a,b).\frac{c}{\text{pgcd}(a,b)}$,
avec $a = a'.\text{pgcd}(a,b), b = b'.\text{pgcd}(a,b)$
soit $-a'.x_0 + b'.y_0 = \frac{c}{\text{pgcd}(a,b)}$

La multiplication et l'addition étant des lois de composition interne dans $\mathbb{Z}$, on a $-a'.x_0 + b'.y_0 \in \mathbb{Z}$. Par conséquent, x_0 et y_0 étant solutions, $\text{pgcd}(a,b)$ divise c. □

Remarque : Si $\text{pgcd}(a,b) = 1$ alors il existe une solution (x_0, y_0) telle que x_0 et y_0 sont proportionnelles à c.

Preuve :

$$\text{pgcd}(a,b) = \text{pgcd}(-a,b) = 1 \quad \Leftrightarrow \quad \exists (u,v) \in \mathbb{Z}^2, -a.u + b.v = 1$$
$$\Leftrightarrow \quad -a.c.u + b.c.v = c$$

Il suffit de prendre $x_0 = c.u$ et $y_0 = c.v$. □

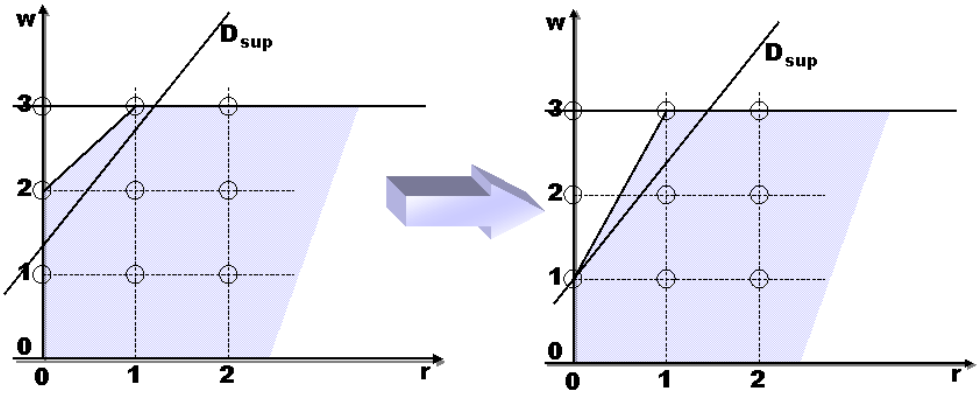
Les équations des droites D_{inf} et D_{sup} sont toutes les deux de la forme $-a.x + b.y = c$ (avec $(a,b) \in \mathbb{N}^{*^2}$ et $c \in \mathbb{Z}$). Si ces équations n'ont pas de solutions dans $\mathbb{Z}^2$ (*i.e.*, les droites ne passent pas par des points à coordonnées entières), alors le polygone de dépendance ainsi délimité sera sur-approximé. Or le théorème 1 dit qu'il existe des solutions dans $\mathbb{Z}^2$ si et seulement si $\text{pgcd}(a,b)$ divise c. Or le plus grand (resp. petit) entier inférieur (resp. supérieur) à c divisible par $\text{pgcd}(a,b)$ est $\text{pgcd}(a,b) \times \left\lfloor \frac{c}{\text{pgcd}(a,b)} \right\rfloor$ (resp. $\text{pgcd}(a,b) \times \left\lceil \frac{c}{\text{pgcd}(a,b)} \right\rceil$).

Ainsi on obtient les nouvelles équations de droite suivantes :

$$D'_{inf} \quad : \quad -a.x + b.y = \text{pgcd}(a,b) \times \left\lceil \frac{c}{\text{pgcd}(a,b)} \right\rceil \tag{5.14}$$

$$D'_{sup} \quad : \quad -a.x + b.y = \text{pgcd}(a,b) \times \left\lfloor \frac{c'}{\text{pgcd}(a,b)} \right\rfloor \tag{5.15}$$

FIG. 5.5 – Élimination partielle de la sur-approximation



Cependant, cette solution ne permet d'éliminer la sur-approximation que dans certains cas. En effet, les points qui nous intéressent sont les intersections des droites d'équations $x = cst$ et $y = cst$, D_{inf} et D_{sup}. Aussi si ces points n'ont pas de coordonnées entières, l'approximation n'est pas contournable.

Dans notre exemple à deux tâches, les droites sont ramenées à la diagonale principale, et nous nous retrouvons alors dans le même cas de figure qu'initialement.

2.5 Conclusion

L'élimination d'approximations dans notre modélisation des dépendances n'est pas possible. La seule solution serait de revenir à un modèle en extension, qui n'est pas viable et qui constitue la principale motivation de notre représentation en intention des dépendances.

Malgré tout, une solution visant à réduire ces approximations est envisagée. En simplifiant les équations des droites définissant notre polygone de dépendances et en les contraignant à passer par des points à coordonnées entières, nous obtenons une élimination, totale ou partielle selon les cas, de la sur-approximation.

Ce problème est un point crucial du placement, agissant sur divers autres problèmes (partitionnement, ordonnancement, génération de code, mémoire, etc). Cette section n'apporte qu'un élément de réponse, mais d'autres voies restent sans doute à explorer.

Ces réflexions ont tout de même permis l'élaboration d'une première version d'une contrainte *PGCD-PPCM* de deux entiers (détaillée en section 2 du chapitre 5).

3 Les entrées/sorties

Ce que nous appelons *entrées/sorties* sont les échanges de données que l'application à placer peut avoir avec l'extérieur (*i.e.*, tout autre système ou application, par exemple le radar). On désigne par *entrée* ce qui *vient* de l'extérieur, et par *sortie* ce qui *va* vers l'extérieur.

Notre modèle d'entrée-sortie M-SPMD s'appuie sur le modèle d'entrée-sortie SIMD/SPMD [4]. Ce dernier peut être exploité de deux manières différentes :

1. microscopiquement, en utilisant les dates physiques exactes de chaque événement,
2. macroscopiquement, en vérifiant seulement que les débits sont tenus et en garantissant que les dépendances sont respectées en les décalant d'une période globale et en les faisant effectuer par un exécutif.

La première approche suppose, pour être effective, que les temps d'exécution de toutes les tâches soient parfaitement connus statiquement ainsi que le comportement du processeur.

La deuxième approche nécessite plus de mémoire pour tamponner les entrées et les sorties sur une ou plusieurs périodes globales alors que les données pouvaient être traitées au vol dans la première approche. Elle ne perturbe pas le débit mais elle augmente la latence des sorties par rapport aux entrées. Elle conduit a priori à une génération de code plus simple puisqu'elle peut naturellement s'appuyer sur les fonctionnalités fournies par un exécutif.

Dans un cadre M-SPMD, sans synchronisation fine entre étapes et avec une architecture physique plaçant les entrées en tête de chaîne et les sorties en queue, on est naturellement conduit à choisir la deuxième approche. Par ailleurs, la méthode de résolution utilisée ne permet pas non plus de choisir la première méthode, le nombre de contraintes à poser étant potentiellement infini (puisqu'il faudrait les poser pour tous les blocs de calculs).

3.1 Tâches fictives d'entrée-sortie

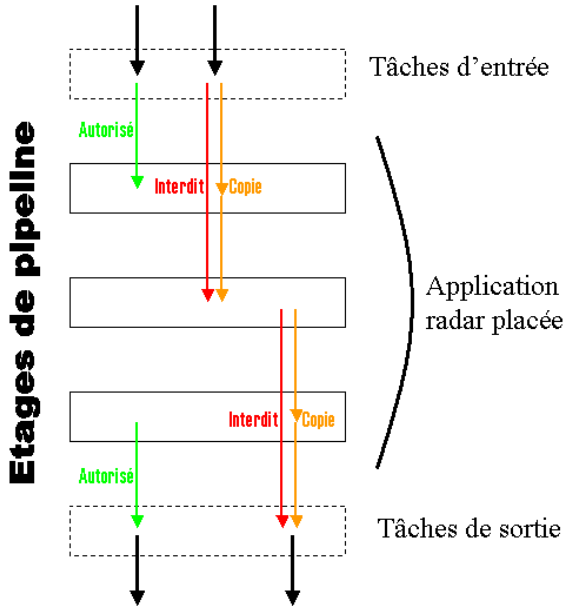
Comme dans [4], des tâches fictives sont associées aux tableaux d'entrée et de sortie. Ces tâches sont respectivement regroupées dans le premier et le dernier étage de pipeline (fig. 5.6). On doit donc supposer que chaque étage de pipeline peut communiquer avec tous ses prédécesseurs et/ou avec tous ses successeurs pour effectuer des lectures tardives ou des écritures prématurées dans la chaîne de traitement.

Ceci est contraire à l'hypothèse M-SPMD 4b (page 86). Puisqu'il n'est pas possible d'ignorer cette hypothèse, il faut ajouter des tâches de copie dont le nombre dépend du placement. En effet, comme l'attribution d'un étage de pipeline à un nid de boucles dépend de son partitionnement, la nécessité d'une tâche de copie en dépend également. De plus l'ajout de ces tâches nuit à l'évaluation de la mémoire, ainsi qu'à celle de la latence. Par ailleurs leur regroupement nécessite également que l'on ajoute deux étages supplémentaires à ceux prévu initialement dans le pipeline.

On impose donc une communication avant toute utilisation d'un tableau en entrée et une communication après toute production d'un tableau en sortie.

On suppose que les temps CPU associés aux entrées-sorties dans [4] sont systématiquement disponibles sur le matériel cible. Les autres temps CPU liés aux entrées et aux sorties sur les étages de calcul ne sont plus que des temps de communication et traités en tant que tels.

FIG. 5.6 – Regroupement des tâches fictives d'entrées/sorties



3.2 Période physique d'une application

Comme dans [4], la période physique Υ se dérive de la période logique α en utilisant n'importe laquelle des périodes d'entrée ou de sortie. Elle est définie par :

$$\Upsilon = \max_{s} \left(\mathrm{T}^{\mathrm{s}}_{exec}\right) = \upsilon^{e} \frac{\alpha}{\alpha^{e}_{0}} \tag{5.16}$$

ou e représente n'importe quel nid de boucles d'entrée ou de sortie et ou υ^e représente la période physique d'acquisition ou d'émission correspondante. Ceci montre que toutes les périodes d'entrée-sortie doivent présenter des rapports rationnels entre elles.

Il faut préciser la supposition suivante : une acquisition correspond au remplissage complet d'un buffer d'entrée. C'est-à-dire que l'on peut faire correspondre la période logique d'une tâche d'entrée e (*i.e.*, α^e_0) avec sa période physique d'acquisition.

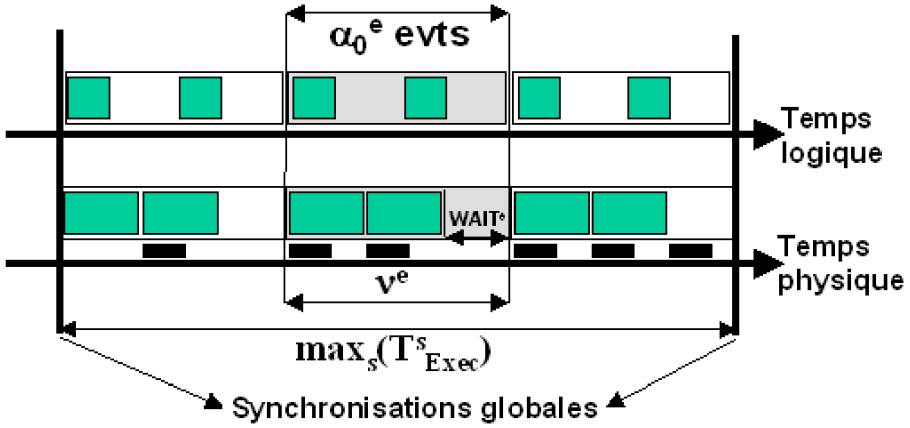
Cependant, cela suppose également que la définition de la durée physique séparant deux synchronisations globales correspond à un multiple commun des durées d'acquisitions. Or ce n'est pas le cas. Il faut introduire un délais d'attente WAIT^e tel que :

$$\mathrm{nbc}^{e}\left(\alpha^{e}_{0}\right) \times \max\left(T_{cal}(e), T_{comm}(e)\right) + \mathrm{WAIT}^{e} = \upsilon^{e} \tag{5.17}$$

Ce délais d'attente doit être supérieur au temps d'une communication de façon à absorber la dernière communication avant synchronisation globale :

$$\mathrm{WAIT}^{e} \geqslant T_{comm}(e) \tag{5.18}$$

FIG. 5.7 – Correspondance période logique / période physique d'acquisition



3.3 Exploitation des périodes physiques/minimisation de $\det(P)$

Cette période physique doit aussi être respectée par les calculs au niveau de chaque étage de pipeline et a fortiori au niveau de chaque nid de boucles. On utilise cette condition redondante pour minimiser les déterminants des matrices P associés aux tâches les plus gourmandes en calculs. On commence par placer toutes les tâches sur un processeur ($\forall P, P = Id$) et sans effectuer de blocage ($\forall L, L = Id$). On obtient une période minimale pour chaque tâche. Si cette période est supérieure à celle que l'on veut atteindre, il faut la réduire en augmentant le nombre de processeurs. On suppose qu'il n'y a pas d'accélération super-linéaire et qu'elle est donc bornée par le déterminant de P.

Par rapport à la condition $L = Id$, il faut utiliser la composante linéaire de la vitesse de calcul. On obtient donc bien une borne supérieure de la vitesse d'exécution indépendante de la valeur de L, dans la mesure où le temps d'exécution est une fonction affine à coefficients positifs de $\det(L)$.

Si on ne dispose que d'un objectif de latence, on sait que la latence est supérieure à une période et au moins égale à NBS fois la période.

Au-delà de la réduction du domaine de $\det(P)$, l'objectif est d'utiliser un des théorèmes de changement d'axe (ou corner-turn) [58] pour placer des tâches dans des étages différents quand on peut dériver $det(P) > 1$.

L'utilisation de ce théorème sur des tâches non contiguës (ne communiquant pas de variables) peut être valide en utilisant la fermeture des dépendances introduites dans[4] au lieu du polyèdre de dépendance habituel.

3.4 Conclusion

Les entrées-sorties permettent de raccorder de manière absolue le temps logique et le temps physique.

Les relations entre périodes d'entrée-sortie et temps de calcul permettent de minimiser

le nombre de processeurs nécessaires pour un nid de boucle et de forcer le placement de certaines tâches dans des étages de pipeline différents.

4 La détection des communications

Un des critères de changement d'étage de pipeline dans un modèle machine M-SPMD est la présence nécessaire d'une communication entre deux nids de boucles (chap. 5 sec. 2). Il est donc utile de pouvoir détecter ce besoin de communication pour accélérer la répartition des tâches entre étages de pipeline. Cette détection est aussi utile dans le cas SIMD ou SPMD pour affiner le calcul des coûts de communication. Cette section est donc indépendante du modèle M-SPMD.

Il faut noter que deux tâches en dépendance, pouvant ne pas communiquer avant la prise en compte du modèle M-SPMD, peuvent être néanmoins placées dans des étages de pipeline différents et, par conséquent, devoir utiliser le réseau de communication de la machine. On distingue donc le prédicat *COMM* qui indique que deux nids de boucles dépendants placés dans des étages de pipeline différents communiquent, du prédicat *MUSTCOMM* qui indiquent que deux nids de boucles partitionnés nécessitent une communication indépendamment de la sélection des étages de pipeline.

Il existe un canal de communication entre deux nids de boucles dépendants l'un de l'autre si et seulement s'ils sont dans des étages de pipeline distants de 1 :

$$\forall nb^1, nb^2 \in \mathrm{NB}, \exists c^{nb^1}, c^{nb^2}$$
$$\left(COMM(nb^1, nb^2)\right) \Longleftrightarrow \left(\ \mathrm{Stage}(nb^2) = \ \mathrm{Stage}(nb^1) + 1\right) \wedge \mathcal{D}\left(c^{nb^1}, c^{nb^2}\right)_{nb^1, nb^2} \quad (5.19)$$

Ce n'est qu'une reformulation des hypothèses 4a et 4b de la section 2 du chapitre 5.

Si deux nids de boucles partitionnés doivent communiquer indépendamment de leurs placements en étages, alors il existera un canal de communication entre eux :

$$\forall nb^1, nb^2 \in \mathrm{NB}$$
$$\left(MUSTCOMM(nb^1, nb^2)\right) \Longrightarrow \left(COMM(nb^1, nb^2)\right) \quad (5.20)$$

Il s'agit ici d'analyser les conditions de communication, *MUSTCOMM* et de non communication $\neg MUSTCOMM$, aussi appelées *parfait alignement des processeurs* dans le chapitre 6 de [32].

Dans [32], le problème de l'existence d'une communication n'est pas résolu puisqu'une hypothèse était de considérer la présence systématique d'une communication (voir chapitre 6 de [32]).

Précisons que cette section repose intégralement sur [58].

4.1 Exemple

De nombreux problèmes peuvent être mis en évidence sur une toute petite application. Elle ne contient que trois nids de boucles (voir le programme de la figure 5.8).

Elle se déroule indéfiniment après son déclenchement.

Pour les professionnels du traitement du signal, il y a un *changement d'axe* entre NB1 et NB3, mais NB2 peut être mis dans le premier ou le second étage de pipeline, avec NB1, ou

FIG. 5.8 – Application avec corner-turn déplaçable

```
#[NB1] FFT 1D complexe sur un vecteur de capteurs
DO t = 0, ...
  DO c = 0, 99
    FREQ[t, c, 0:255] = FFT(INPUT[256*t:256*t+255, c])

#[NB2] calcul d'énergie
DO t = 0, ...
  DO c = 0, 99
    DO f = 0, 255
      ENERG[t, c, f] = ENERGIE(FREQ[t, c, f])

#[NB3] sommation sur les capteurs pour chaque fréquence
DO t = 0,...
  DO f = 0, 255
    ENERG_F[t, f] = SUM(ENERG[t, 0:99, f])
```

NB3. Suivant le choix fait, le volume des communications peut être différent. Ici, le calcul d'énergie peut permettre de réduire d'un facteur 2 le volume de données à transférer.

Ce *changement d'axe* concerne donc des nids de boucles qui ne sont pas contigus dans le graphe de dépendances des tâches.

Par ailleurs le partitionnement permet de supprimer le besoin de communication. Il est possible de n'allouer qu'un processeur pour les trois tâches NB1, NB2 et NB3. Il est aussi possible d'utiliser l'axe temporel pour effectuer la parallélisation. Ces solutions mathématiquement valides sont éliminées d'emblée par les spécialistes. La première ne permettrait pas de tenir le débit et la seconde allongerait excessivement la latence. Ces solutions ne peuvent être éliminées rapidement que si des contraintes redondantes sont définies. Par exemple, le débit ne peut pas être supérieur à l'inverse du temps de traitement d'un bloc d'une tâche quelconque et la latence est au moins égale à ce temps. Le problème de l'introduction de l'expérience du spécialiste sous la forme de contraintes redondantes n'est pas l'objet de cette section. Il faut juste se souvenir qu'un problème peut être simple pour le spécialiste et plus compliqué pour un outil (et vice-versa) parce qu'ils ne disposent pas des mêmes connaissances :

- une FFT représente (toujours) une part importante d'un calcul et il faut (toujours) la paralléliser ;
- on n'utilise jamais l'axe temporel pour paralléliser une application parce que, par expérience, on est (presque) sûr d'être limité par la cadence des entrées et de ne pouvoir offrir une latence acceptable.

Les spécialistes du traitement du signal font aussi des hypothèses sur l'ordre des boucles et sur l'ordre des dimensions des tableaux. Chaque boucle et chaque dimension correspondent à une grandeur physique particulière propre à l'application et donc à tous les nids de boucles et à tous les tableaux. On s'aperçoit alors que les schémas, les nids de boucles et les dimensions

des tableaux suivent tous un ordre global, celui qui a été choisi pour les dimensions physiques. Le spécialiste sait que les indices c1, c2 et c3 correspondent nécessairement à la dimension de sémantique *capteur*. Le modèle mathématique des spécifications ne prend pas cela en compte et la résolution doit donc envisager toutes les permutations possibles sur les nids de boucles.

La spécification par ARRAY-OL ne permet pas de distinguer les tâches de réduction, comme la sommation sur les capteurs, des autres tâches. Les tâches sont atomiques. La communication liée au *changement d'axe* pourrait cependant être considérablement réduite si des réductions partielles étaient effectuées avec le partitionnement choisi pour NB1 et NB2.

Enfin, le partitionnement de la tâche NB2, qui ne pose intrinsèquement aucun problème puisque c'est une tâche point-à-point (sans motif), peut nécessiter des communications avec NB1 ou NB3. Il suffit de placer NB2 sur une grille 2-D non-triviale de processeurs pour ne plus pouvoir se passer de communication ni avec NB1 ni avec NB3 et se retrouver dans l'obligation d'ajouter un troisième étage de pipeline (hypothèse 4a). On peut aussi créer des problèmes en ne traitant qu'une partie des capteurs ou des fréquences. Même si les motifs ont des tailles compatibles, il y a systématiquement des intersections dues aux décalages à l'origine.

En conclusion, le problème de modélisation mathématique des communications est plus complexe qu'on ne le pense a priori. Le partitionnement automatique ne prend pas en compte dans ses choix le savoir que peut avoir un expert. De ce fait, beaucoup de possibilités offertes (mathématiquement) à un outil sont éliminées par l'expérience qu'un expert peut avoir.

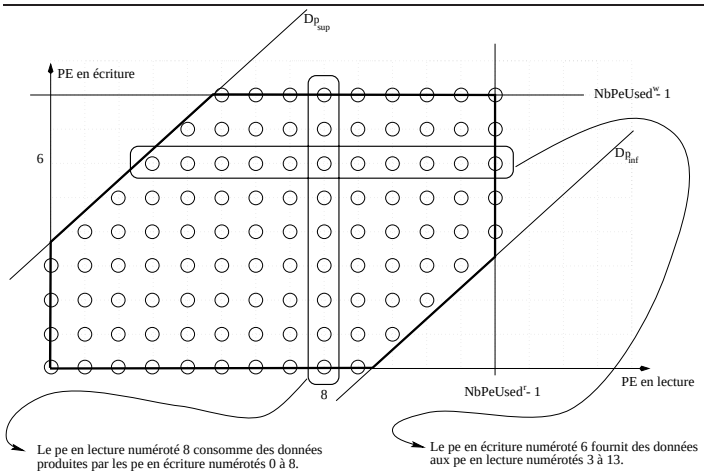
4.2 Détection par projection sur l'axe processeur

S'il existe un lien de communication entre deux blocs de calcul alors il existe nécessairement une dépendance entre les deux ; pouvons-nous reprendre l'idée du polygone de dépendance pour l'adapter aux communications ? C'est-à-dire modifier notre point de vue d'un axe temporel (le paramètre c du partitionnement) vers un axe processeur (le paramètre p du partitionnement) ? Son interprétation serait similaire à celle des dépendances, comme le montre le figure 5.9.a. Ainsi, une absence de communication serait traduite par la diagonale principale sur toutes les dimensions (fig. 5.9.b).

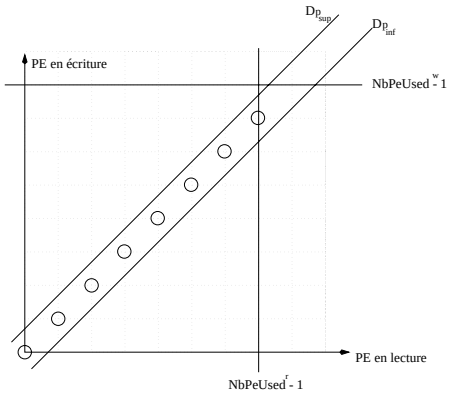
Malheureusement du fait des différentes projections à effectuer, il n'est généralement pas possible de calculer un polyèdre de communication précis [58]. La composante p du partitionnement n'étant pas de poids fort, il y a des *espaces* séparant deux itérations s'effectuant sur le même processeur, à des cycles différents. Ces espaces correspondent aux itérations calculées sur des processeurs différents. Les projections effectuées pour isoler les variables p perdent ces informations. Si l'on considère un ensemble d'itérations calculées sur un processeur, $\{2, 4, 6, 8\}$ par exemple, ces projections le transforme en l'intervalle $[2, 8]$. Ce qui signifie faussement que ce processeur participe aux itérations $\{3, 5, 7\}$. De manière générale, cela se traduit par le fait que tous les processeurs communiquent les uns avec les autres et que la seule condition pour qu'il n'y ait pas de communication est de n'en utiliser qu'un seul.

Nous pouvons observer le même type d'imprécision sur le polyèdre de dépendance, à la nuance près que le paramètre cyclique c est la composante de poids fort du partitionnement. Mais la différence la plus importante est la suivante : les cycles de calcul étant exécutés les uns à la suite des autres, dans l'ordre croissant, s'il existe un cycle c^r en lecture sur des données produites en partie par un cycle c_1^w et en partie par un autre cycle c_6^w, alors il est

FIG. 5.9 – Représentation des dépendances physiques : Échange de données entre processeurs dépendants



a. Cas général.



b. Cas sans échange entre processeurs.

nécessaire pour c^r d'attendre que les cycles c_1^w à c_6^w se soient exécutés. C'est-à-dire que le regroupement conduisant à une impasse dans le cas des communications (*tous les processeurs communiquent*) est négligeable (et même nécessaire pour les durées de vie) dans le cas des dépendances.

Sous certaines conditions, la projection peut être suffisamment précise. Il faut pouvoir diviser l'équation de communication, à savoir $\exists c^w, p^w, l^w, m^w, c^r, p^r, l^r, m^r$

$$\Omega^w_{pav}\left(L^w P^w c^w + L^w p^w + l^w\right) + \Omega^w_{fit}.m^w + \mathcal{K}^w \quad = \quad \Omega^r_{pav}\left(L^r P^r c^r + L^r p^r + l^r\right) + \Omega^r_{fit}.m^r + \mathcal{K}^r \quad (5.21)$$

par le coefficient de c, à la condition que le pavage recouvre l'ajustage, soit lorsque $\Omega_{pav} \geqslant \Omega_{fit} m_{max}$.

Ces coefficients sont : $\Omega^w_{pav} L^w P^w$ et $\Omega^r_{pav} L^r P^r$.

S'ils sont égaux, on peut alors simplifier l'équation en éliminant les c puis trouver des conditions sur les p en projetant les m et les l.

Cette condition est utilisée par l'expert.

4.3 Normalisation des accès

Les matrices d'accès, Ω_{pav} et Ω_{fit}, sont des permutations de matrices diagonales, carrées ou non. Dans le cadre du calcul du prédicat $MUSTCOMM$, il est possible d'ajouter des boucles fictives à w dont le nombre d'itérations est systématiquement égal à 1 de manière à ce que les dimensions des vecteurs i^w et m^w soient égales à la dimension du tableau référencé. Les matrices Ω^w_{pav} et Ω^w_{fit} sont complétées par des 0 et des 1. On applique ensuite une permutation sur les dimensions de i^w et une autre permutation sur les dimension de m^w pour obtenir deux nouvelles matrices Ω^w_{pav} et Ω^w_{fit} diagonales sans avoir modifié l'ensemble des prédicats élémentaires de $MUSTCOMM$.

Il est possible de faire exactement la même chose avec les boucles externes et internes de r

bien que la dimension de i^r puisse être supérieure à celle du tableau lu ou plus généralement bien que la matrice de pavage Ω^r_{pav} puisse contenir une colonne nulle : les mêmes éléments sont relus plusieurs fois par des itérations différentes. Ce serait le cas, par exemple, d'un tableau de coefficients. On peut néanmoins utiliser deux permutations pour diagonaliser Ω^r_{fit} qui ne doit pas avoir de colonnes nulles et pour transformer Ω^r_{pav} en une matrice diagonale pouvant contenir des coefficients nuls complétée par une matrice nulle pour absorber les dimensions d'itération n'intervenant pas dans la fonction d'accès.

FIG. 5.10 – Application avec corner-turn déplaçable après normalisation

```
DO t = 0, ...
  DO c = 0, 99
    DO f = 0, 0
      FREQ[t, c, f+0:255] = FFT(INPUT[256*t:256*t+255,c,f])

DO t = 0, ...
  DO c = 0, 99
    DO f = 0, 255
      ENERG[t, c, f] = ENERGIE(FREQ[t, c, f])

DO t = 0, ...
  DO c = 0, 0
    DO f = 0, 255
      ENERG_F[t, c, f] = SUM(ENERG[t, c+0:99, f])
```

Les matrices de pavage et d'ajustage sont transformées de la manière suivante pour la définition du tableau FREQ :

$$\Omega^w_{pav} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \longrightarrow \Omega^w_{pav} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{5.22}$$

$$\Omega^w_{fit} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \longrightarrow \Omega^w_{fit} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.23}$$

Elles ne sont pas modifiées pour son utilisation puisque déjà complètes :

$$\Omega^r_{pav} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \Omega^r_{fit} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.24}$$

Les matrices d'accès aux tableaux ENERG et ENERG_F sont traitées de la même manière.

Cet exemple ne montre pas de permutation de dimensions parce que les dimensions physiques sont toujours traitées dans le même ordre. Il est facile d'imaginer un échange de

boucles dans le nid de définition de l'énergie pour comprendre pourquoi une permutation des dimensions peut aussi être utile.

La normalisation des contraintes de communication consiste donc à prendre en considération d'abord toutes les dimensions du tableau créant la dépendance puis toutes les dimensions j du nid de boucles de lecture, r. Si une dimension j est utilisé pour parcourir une dimension k du tableau, i.e. si $\Omega^r_{pav_{k,j}} \neq 0$, on ne la garde pas. Dans le cas contraire, on l'ajoute à la base commune en complétant le tableau et le nid de boucles w par une dimension supplémentaire de borne 1. On ne change donc ni les volumes des tableaux ni les nombres d'itérations. Les indices de motifs sont aussi complétés de la même manière pour avoir les mêmes dimensions. Les matrices Ω^w_{pav}, Ω^w_{fit}, Ω^r_{pav} et Ω^r_{fit} sont aussi complétées et permutées pour être toutes diagonales. Les matrices de domaines, $\mathfrak{D}^r$, M^r, $\mathfrak{D}^w$ et M^w, sont pareillement permutées et complétées.

Cette normalisation des nids de boucles et des matrices d'accès permet d'éviter l'introduction de fonctions de permutation faisant correspondre (ou pas) à une dimension de tableau un indice de boucle particulier et réciproquement. L'équation de dépendance nécessite deux fonctions de ce genre pour être écrite sous forme scalaire.

4.4 Condition de réutilisation des données

Une donnée est réutilisée si :

$$\begin{aligned} \exists i^r \in \mathfrak{D}^r,\ \exists m^r \in M^r \\ \exists i'^r \neq i^r \in \mathfrak{D}^r,\ \exists m'^r \in M^r \\ \Omega^r_{pav} i^r + \Omega^r_{fit} m^r + \mathcal{K}^r = \Omega^r_{pav} i'^r + \Omega^r_{fit} m'^r + \mathcal{K}^r \end{aligned} \tag{5.25}$$

ce qui peut être réécrit :

$$\begin{aligned} \exists x \neq 0 \in \mathfrak{D}^r,\ \exists y \in M^r \\ \Omega^r_{pav} x = \Omega^r_{fit} y \end{aligned} \tag{5.26}$$

en limitant les domaines de $x = i^r - i'^r$ et de $y = m'^r - m^r$ aux entiers positifs moyennant une éventuelle multiplication par moins un de l'équation. Les matrices et les vecteurs sont réduits à la dimension k et ces équations sont donc scalaires.

L'ensemble des solutions est :

$$\left\{(x,y) | \exists \lambda \in \mathbb{N}\ (x,y) = \lambda({\Omega^r_{fit}}', {\Omega^r_{pav}}') < (\mathfrak{D}^r, M^r)\right\} \tag{5.27}$$

où ${\Omega^r_{fit}}'$ et ${\Omega^r_{pav}}'$ sont les nombres premiers entre eux obtenus par les divisions de Ω^r_{fit} et Ω^r_{pav} par leur plus grand commun diviseur.

Cet ensemble est vide si ${\Omega^r_{fit}}' \geqslant \mathfrak{D}^r$, ce qui voudrait dire que le coefficient d'ajustage est trop grand par rapport au nombre d'itérations. Par exemple, on pourrait souhaiter mettre dans un motif toutes les fréquences parmi 256 qui ont le même modulo f0 par rapport à 32 :

... = f(FREQ[t, c, f0+32*m])

Cet ensemble est vide si ${\Omega^r_{pav}}' \geqslant M^r$ ce qui voudrait dire que le coefficient de pavage est trop grand par rapport au nombre de motifs. Par exemple, on pourrait utiliser 10 capteurs tous les 16 capteurs :

... = INPUT[t, 16*c+m, f]

Cet ensemble est vide par définition s'il n'y a qu'une itération, i.e. si $\mathfrak{D}^r = 1$. Comme Ω^r_{fit} est plus grand ou égal à 1 par définition, cette condition est impliquée par ${\Omega^r_{fit}}' \geqslant \mathfrak{D}^r$.

Cet ensemble est vide si le motif est réduit à un point, $M^r = 1$ parce que y est alors forcé à zéro ce qui est impossible avec λ strictement positif, tout comme Ω^r_{fit} et Ω^r_{pav}. Comme Ω^r_{pav} est plus grand ou égal à 1 par définition, cette condition est impliquée par ${\Omega^r_{pav}}' \geqslant M^r$.

Lemme 5.4-1 : Une donnée est réutilisée par un nid de boucle r si et seulement si ${\Omega^r_{fit}}' < \mathfrak{D}^r$ et si ${\Omega^r_{pav}}' < M^r$.

Preuve : voir ci-dessus le nombre de solutions du prédicat de réutilisation.

Théorème 5.4-1 : si l'ajustement est trivial, i.e. $\Omega^r_{fit} = 1$, une donnée est réutilisée si le coefficient de pavage est inférieur à la taille du motif, i.e. $\Omega^r_{pav} < M^r$.

Preuve : cas particulier du Lemme précédent.

Si les données lues sont réutilisées par des itérations différentes d'un nid de boucles r dans une dimension k (après normalisation), alors il doit y avoir une communication entre w et r si cette dimension est parallélisée dans w ou dans r.

4.5 Projection des processeurs logiques sur les processeurs physiques

Le partitionnement attribue à chaque nid de boucles r un ensemble de processeurs logiques définis par la matrice diagonale P^r. Ces processeurs logiques doivent être associés à des processeurs physiques (cf. [32, chapitre 6]). Comme la topologie n'a pas d'impact sur les communications, on peut considérer que les processeurs banalisés sont numérotés de 0 à $PEMax - 1$. On suppose que la fonction de placement des processeurs logiques est une application injective : un processeur logique ne s'exécute qu'une fois et un processeur physique n'exécute qu'un processeur logique. Ceci permet d'évaluer les temps d'exécution en ne considérant que les processeurs logiques, *i.e.*le placement des processeurs n'introduit pas de nouveaux conflits de ressource.

Cette fonction de placement des processeurs est appelée $\mathcal{L}$ dans [32] et est définie par :

$$\begin{array}{rccl} \mathcal{L} & : & \{p | 0 \leqslant q < P\} & \longrightarrow [0..PEMax - 1] \\ & & p & \mapsto \mathcal{L}(p) \end{array} \tag{5.28}$$

$$\mathcal{L}(p) = \sum_i p_i \prod_{j<i} P_{j,j} \tag{5.29}$$

Cette projection des processeurs logiques vers les processeurs physiques n'est pas satisfaisante, pratiquement ou mathématiquement. En effet, elle est conduite indépendamment des autres nids de boucles. Elle ne permet pas de dire si deux processeurs virtuels utilisés pour le partitionnement de deux nids de boucles sont projetés sur le même processeur physique.

Elle conduit à la condition

$$P^w \neq P^r \Rightarrow MUSTCOMM(w,r)$$

Autrement dit il faudrait avoir $P^n = P^m$ pour tous les nids de boucles n et m appartenant au même étage de pipeline M-SPMD.

Les égalités de matrices P sont compliquées par la normalisation. Il faut prendre en compte au moins des permutations pour écrire ces égalités puisque la normalisation varie suivant les paires de références considérées.

Dans un contexte M-SPMD, il vaudrait mieux utiliser la condition $P^w \geqslant P^r$ à l'intérieur d'un étage et projeter les processeurs logiques en utilisant un offset et la matrice P la plus grande de l'étage pour chaque «chaîne» passant par cette étage.

4.6 Changements d'axe : les *corner-turns*

La détection des corner-turns semble très facile aux praticiens parce qu'ils savent que les tâches concernées ne peuvent pas être exécutées séquentiellement (débordement en latence ou insuffisance en débit) ou parallèlement dans la dimension temporelle (débordement en mémoire et en latence).

Il faut donc utiliser une condition de latence provenant des contraintes d'entrée-sortie pour prouver que $P^w > Id$ et/ou $P^r > Id$. Cela doit venir des α^t et du nombre d'exécutions de la tâche t pendant une période α. On fixe $P = Id$ et $L = Id$ et on prend la composante vectorielle du temps d'exécution (s'il est affine en L sinon on prend simplement le temps d'exécution). On suppose qu'il n'y a pas d'accélération super-linéaire et qu'elle est inférieure à $\det(P)$. On peut donc minimiser $\det(P)$ si la tâche nécessite beaucoup de calculs et si la contrainte temps-réel d'entrée-sortie est assez tendue.

Voici trois théorèmes du corner-turn utilisant soit $P^w = P^r$ soit $P^w \geqslant P^r$ soit aussi le fait que toutes les itérations de w produisent au moins une donnée utile pour r. Ces théorèmes permettent d'utiliser le fait que les déterminants sont plus grands que 1 et que les spécifications sont incompatibles dans certaines dimensions pour montrer que les deux tâches doivent être placées dans des étages différents.

Théorème 5.4-2 : Théorème 1 du corner-turn

Si après normalisation la matrices P^w n'est pas plus grande que la matrice P^r, alors il y a forcément communication entre les tâches w et r quel que soit le placement des processeurs logiques sur les processeurs physiques :

$$\neg(P^w \geqslant P^r) \implies MUSTCOMM(w,r) \tag{5.30}$$

Preuve : Soit k une dimension telle $P^w_{k,k} \neq P^r_{k,k}$. Tout processeur $(0,0,...,p^r_k,0,...0)$ utilise une donnée produite par un processeur $(x_1, x_2, ..., p^w_k, x_{k+1}, ...x_n)$ où :

$$x_i = \frac{\mathcal{K}^r_i}{\Omega^w_{pav_{i,i}} L^w_{i,i}} \bmod P^w_{i,i} \tag{5.31}$$

Les x_i sont nuls quand les décalages $\mathcal{K}^r_i$ sont nuls mais ce n'est pas toujours le cas.

Les équations $i \neq k$ sont trivialement vérifiées par définition des x_i parce que les données consommées sont forcément produites et les équations k sont vérifiées par construction. S'il n'y a pas communication, il faut que chaque processeur consommateur ait un producteur différent et donc que $P^w_{k,k} \geqslant P^r_{k,k}$.

On a donc : $P^w \geqslant P^r$ puisqu'il y a égalité dans les autres cas.

Théorème 5.4-3 : Théorème 2 du corner-turn

Si après normalisation les matrices P^w et P^r ne sont pas égales dans leur $n-1$ premières dimensions et si les processeurs logiques sont placés en utilisant la forme linéaire $\mathcal{L}()$, alors il y a forcément communication entre les tâches w et r.

$$\left(\exists k \in [1..n[\;\; P^w_{k,k} \neq P^r_{k,k}\right) \vee P^w_{n,n} < P^r_{n,n} \quad \Rightarrow \quad MUSTCOMM(w,r) \tag{5.32}$$

Preuve : Il faut maintenant utiliser la fonction de linéarisation pour montrer que l'identité des processeurs physiques implique :

$$\forall i \in [1..k[P^w_{i,i} = P^r_{i,i}$$

Les processeurs physiques lecteurs sont :

$$\left\{ q^r | \exists x \in [0, P^r_{k,k}[q^r = x \prod_{j<k} P^r_{j,j} \right\} \tag{5.33}$$

tandis que les processeurs physiques écrivains correspondant sont à prendre parmi :

$$\left\{ q^w | \exists y \in [0, P^w_{k,k}[q^w = y \prod_{j<k} P^w_{j,j} + \sum_{i \neq k} x_i \prod_{j<i} P^w_{j,j} \right\} \tag{5.34}$$

Comme tous les lecteurs sont utilisés, on sait donc que le processeur logique 0 est utilisé et qu'il est placé sur le processeur physique 0 par la fonction de linéarisation.

On a donc :

$$\exists y \in [0, P^w_{k,k}[0 = y \prod_{j<k} P^w_{j,j} + \sum_{i \neq k} x_i \prod_{j<i} P^w_{j,j}$$

Comme tous les termes sont positifs et que $P^w_{j,j} > 0$, il faut donc :

$$\begin{aligned} \forall i \neq k \; x_i \prod_{j<i} P^w_{j,j} &= 0 \\ \iff \forall i \neq k \; x_i &= 0 \\ \iff \forall i \neq k \; \frac{\mathcal{K}^r_i}{\Omega^w_{pa\,v_{i,i}} L^w_{i,i}} \bmod P^w_{i,i} &= 0 \end{aligned} \tag{5.35}$$

en substituant x_i par sa définition 5.31. Notons que les contraintes supplémentaires (5.35) pourraient être aussi utilisées pour détecter facilement une communication.

L'ensemble des producteurs potentiels se simplifie donc de (5.34) en :

$$\left\{ q^w | \exists y \in [0, P^w_{k,k}[q^w = y \prod_{j<k} P^w_{j,j} \right\} \tag{5.36}$$

Comme $P^w_{j,j} \geqslant P^r_{j,j} > 0$ pour tout j, on a donc

$$\prod_{j<k} P^w_{j,j} \geqslant \prod_{j<k} P^r_{j,j} \tag{5.37}$$

En l'absence de communications, les processeurs lecteurs (equation 5.33) ne peuvent toucher au plus que :

$$n = \frac{P^r_{k,k} - 1) \prod_{j<k} P^r_{j,j}}{\prod_{j<k} P^w_{j,j}} \tag{5.38}$$

lecteurs. Ce nombre de lecteur n est donc inférieur ou égal au nombre de producteur. Pour qu'il n'y ait pas de communication, il faut donc :

$$\forall k \in dim(P^r) \quad \prod_{j<k} P^r_{j,j} = \prod_{j<k} P^w_{j,j} \tag{5.39}$$

Il faut donc que les deux matrices P^w et P^r soient égales, sauf sur la dernière dimension où P^w doit être supérieure à P^r (équation 5.30).

Théorème 5.4-4 : Théorème 3 du corner-turn

Si pour toute dimension $k \in [1, n[$, $P^w_{k,k} \leqslant 1$ ou $P^r_{k,k} \leqslant 1$, il faut effectuer un changement d'axe (i.e. une communication) si la tâche r ou la tâche w doit être effectivement exécutée en parallèle sur au moins une de ces dimensions.

$$\left.\begin{array}{l} \left(\forall k \in [1, n[\quad P^r_{k,k} \leqslant 1 \vee P^w_{k,k} \leqslant 1\right) \\ \wedge \left(\left(\prod_{k/P^r_{k,k} \neq 0} (P^r_{k,k}) > 1 \right) \vee \left(\prod_{k/P^w_{k,k} \neq 0} (P^w_{k,k}) > 1 \right) \right) \end{array}\right\} \iff MUSTCOMM(w, r) \tag{5.40}$$

Preuve :

Pour qu'il n'y ait pas communication à coup sûr, i.e. $\neg MUSTCOMM(w, r)$, il faut, après normalisation, que $P^w = P^r$ sur les $n-1$ premières dimensions, car (Théorème 2 du corner-turn) :

$$P^w \neq P^r \implies MUSTCOMM(w, r)$$

donc $P^w_{k,k} = P^r_{k,k}$ pour tout $k \in [1, n[$:

$$P^w_{k,k} \leqslant 1 \wedge P^w_{k,k} = P^r_{k,k} \Longrightarrow P^w_{k,k} = P^r_{k,k} = 1$$
$$P^r_{k,k} \leqslant 1 \wedge P^w_{k,k} = P^r_{k,k} \Longrightarrow P^w_{k,k} = P^r_{k,k} = 1$$

C'est-à-dire aucun parallélisme, ce qui est contraire à la deuxième hypothèse. Il doit donc y avoir communication.

4.7 Conclusion

Il est difficile de prédire une présence ou une absence de flot de données entre processeurs automatiquement. À cela trois raisons majeures. Tout d'abord, même si l'on sait associer les processeurs logiques entre eux, la fonction de projection de [32] vers les processeurs physiques ne garantit pas la conservation de ces associations. En effet, elle est définie pour chaque partitionnement indépendamment les uns des autres. Considérons par exemple deux nids de

boucles w et r, de profondeurs différentes, tels que le premier initialise un tableau utilisé par le second. En supposant qu'ils utilisent le même nombre de processeurs ($\det(P^w) = \det(P^r)$) et que la parallélisation soit sur les mêmes boucles[4], les fonctions de linéarisation des processeurs logiques ([32, p.117]) ne permettent pas de s'assurer que les mêmes processeurs physiques seront effectivement utilisés.

Le deuxième point concerne la prise en compte des permutations de boucles.

Enfin, la communication peut être nécessaire, non pas à cause des partitionnements, mais parce que les nids de boucles associés ont dus être placés sur deux étages de pipeline différents.

[4] C'est à dire *les boucles qui donnent accès aux mêmes dimensions du tableau.*

Chapitre 6

Des modèles formels aux modèles contraintes

La formalisation mathématique des éléments et des fonctions du problème représente à la fois le point fort et le point faible de la PPC. Grâce aux paradigmes des contraintes, on améliore considérablement les choses en ce qui concerne la modélisation. En effet, les contraintes garantissent une propriété de monotonie qui permet au développeur de ne pas se soucier de l'ordre dans lequel les contraintes sont introduites dans le système. La conséquence principale qu'apporte cette propriété de monotonie est qu'elle rend possible une programmation incrémentale. Cela signifie que si l'on désire modifier ou ajouter un nouveau modèle à la modélisation, on peut le faire sans avoir à se soucier des interactions avec les modèles déjà existants.

De plus, l'efficacité d'un programme PPC est essentiellement basée sur une axiomatisation ne sortant pas d'une sémantique mathématique suffisamment simple afin qu'elle soit efficacement implémentable. Ceci implique que l'axiomatisation doit pour l'essentiel être réalisée à partir des algèbres de contraintes pré-définies. Ces algèbres sont dans la plupart des cas des langages, réduits à des structures décidables telles que l'arithmétique linéaire entière ou le calcul propositionnel.

Enfin, la classe des langages de PPC permet la combinaison de la logique et des contraintes sur des structures mathématiques de type arithmétique linéaire. Les programmes de PPC, par l'intermédiaire des contraintes, met en exergue non seulement les relations locales entre les paramètres du problème, mais aussi et de façon déclarative, la résolution de problèmes combinatoires en utilisant un paradigme de recherche globale.

La seconde phase concerne la conception. Cette phase peut être découpée en deux tâches étroitement liées.

1. La première tâche, appelée «conception préliminaire», reprend la formalisation mathématique du problème pour en déduire une modélisation mathématique des contraintes et des relations du problème. Il s'agit de modéliser le problème dans l'axiomatique autorisée sous-jacente au langage de la programmation par contrainte, c'est à dire pour le cadre de la programmation par contrainte dans les domaines finis, l'arithmétique linéaire entière et le calcul propositionnel. Pour mener à bien cette étape, il faut modéliser le problème soit par le biais de conditions équivalentes soit par le biais de

«dégradation» des relations du problème. Ces dernières sont soit des conditions suffisantes ce qui permet de garantir formellement que toute solution trouvée est valide (mais il se peut que l'on perde des solutions) soit par des approximations qui rendent moins fines la modélisation.

2. La seconde tâche, appelée «conception détaillée», traduit la formalisation pendant la «conception préliminaire» en modèles «contraintes». En effet, les problèmes combinatoires abordés par la PPC qu'il s'agisse de problèmes de satisfiabilité (trouver une solution) ou de problèmes d'optimisation (trouver la meilleure solution) sont définis par un ensemble de variables et leurs domaines de viabilité et un ensemble de relations caractérisant les propriétés de la solution : les contraintes. Formellement, à tout problème P on se donnera un triplet $((v_1, \ldots, v_n); (\mathfrak{D}_{v_1}, \ldots, \mathfrak{D}_{v_n}); (\phi_1, \ldots, \phi_k))$ où les v_i sont les variables à instancier par une valeur du domaine $\mathfrak{D}_{v_i}$ de telle sorte que les formules algébriques ϕ_j (les contraintes) soient satisfaites.

Les chapitres précédents constituent l'étude préliminaire correspondant à la définition mathématique des différents modèles. Avant d'utiliser ces formules, une étude plus «contrainte» est nécessaire. L'une et l'autre de ces études sont difficiles. Nous nous sommes concentrés, lors de cette thèse, sur l'aspect modélisation mathématique d'une machine M-SPMD (et les problèmes qui en découlent). Aussi, la deuxième partie de la modélisation a été réalisée de façon à pouvoir rapidement implémenter ces modèles.

La première section de ce chapitre reprend chacun des modèles formels (aussi bien ceux communs à [32] et cette thèse que ceux présentés dans ce document) pour en donner leur adaptation au langage utilisé pour les expérimentations : Eclair[47].

La deuxième section explique comment nous avons pu exprimer avec des contraintes élémentaires des opérateurs aussi peu monotones que le PGCD et le PPCM de deux entiers.

Le chapitre suivant faisant état des résultats obtenus, la dernière section fait un point sur la stratégie de résolution adoptée, ainsi que sur les heuristiques utilisées.

1 Les modèles contraintes du placement

Nous avons regroupé dans cette section les différentes contraintes définissant le problème du placement sur une machine M-SPMD. Ce regroupement se présente sous la forme d'un tableau à 3 colonnes. La première donne le nom du sous problème concerné, la deuxième contient les contraintes nécessaire, la dernière exprime leur interprétation physique.

Le codage effectif de ces contraintes dépend fortement de l'environnement de développement utilisé. C'est pourquoi nous avons laissé les expressions mathématiques initiales, sachant qu'elles peuvent avoir différentes interprétations selon la cible choisie (Eclair©[47], Ilog Solver©[37], Chip©[25], etc).

Modèle	Contrainte	Interprétation
Partitionnement	$i = LPc + Lp + l$ (utilisation implicite)	Partitionnement 3D de l'espace d'itération.c=cycle (temps), p=partition (processeur), l=localité (mémoire).À chaque cycle c, p processeurs sont utilisés pour exécuter l calculs.
	L et P diagonales $0 \preceq c \prec c_{max}$ $0 \preceq p \prec P.1$ $0 \preceq l \prec L.1$ $\det(P) > 0$ $\det(L) > 0$	Projection parallèle aux axes non redondance des calculs dans le temps non redondance des calculs dans un cycle non redondance des calculs sur un processeur au moins 1 processeur utilisé par cycle (p. 43) au moins un calcul effectué par processeur (p. 43)
Ordonnancement	$\forall nb \in \mathrm{NB}, \forall c^{nb} \prec c^{nb}_{max}, \mathrm{d}\left(c^{nb}\right) = \alpha^{nb}.c^{nb} + \beta^{nb}$ $\alpha^{nb} = N_s.\gamma^{nb} \quad \beta^{nb} = N_s.\delta^{nb} + k^{nb}_s$	Projection des dates événementielles propres à chaque tâche sur un axe de temps logique commun à toutes (p. 91). SPMD de chaque étage implicite. Deux blocs de calculs de deux tâches différentes dans le même étage de pipeline ne s'exécutent pas simultanément. s=étage, N=nombre de tâches, k=numéro d'une tâche (p. 91).

⟶

Suite...

Modèle	Contrainte	Interprétation
	$\alpha^{nb}_{ld-1} \geqslant 1$ et $\forall n \in [1, ld-1] \quad \alpha^{nb}_{n-1} \geqslant \alpha^{nb}_{n} c^{nb}_{n}$	Conservation de l'ordre total d'exécution des blocs de calculs d'une tâche. Deux blocs de calculs issus d'une même tâche ne s'exécutent pas en même temps.
	$\alpha = \text{ppcm}_{nb}\left(\alpha^{nb}_{0}\right)$	Périodes de calcul de chaque tâche calées sur la période globale (p. 91).
	$\beta^{nb} \geqslant (\ \text{Stage}(nb) - 1) \times \alpha$	Le temps de start-up d'une tâche est au moins égal à la somme des durées événementielles des étages de pipeline précédant celui auquel cette tâche appartient (p. 91).
Dépendances	$\begin{array}{lll} AIJ & = (\max(\min(x_I, x_C), x_A) & , \quad \max(\min(y_J, y_G), y_A)) \\ BCJ & = (\max(\min(x_B, x_C), x_A) & , \quad \max(\min(y_J, y_G), y_A)) \\ CDL & = (\max(\min(x_L, x_C), x_A) & , \quad \max(\min(y_D, y_G), y_A)) \\ EKL & = (\max(\min(x_L, x_C), x_A) & , \quad \max(\min(y_K, y_G), y_A)) \\ FGK & = (\max(\min(x_F, x_C), x_A) & , \quad \max(\min(y_K, y_G), y_A)) \\ GHI & = (\max(\min(x_I, x_C), x_A) & , \quad \max(\min(y_H, y_G), y_A)) \end{array}$	Définition, pour une dimension d'un tableau, des coordonnées des sommets générateurs du polygone de dépendance correspondant (p. 73).
	$\forall nb_1, nb_2 \in \text{NB},\ \text{Stage}(nb^1) < \text{Stage}(nb^2)$, $\forall c^{nb^1}, \forall c^{nb^2}$ $\left(\mathcal{D}\left(c^{nb^1}, c^{nb^2}\right)\right) \Rightarrow \left(\exists \lambda > 0 \text{ t.q. } \text{d}^{nb^1}\left(c^{nb^1}\right) < \lambda.\alpha \leqslant \text{d}^{nb^2}\left(c^{nb^2}\right)\right)$	Deux blocs de calculs directement dépendants de deux tâches différentes situées sur deux étages différents sont séparés par au moins une synchronisation globale (p. 92).
	$\forall i, \forall j \text{ t.q. } \mathcal{D}(i,j), a_{i,j} = \max_{s \in \left\{ \begin{array}{c} \text{sommets du polytope} \\ \text{de dépendance} \end{array} \right\}} (d^j(c^j_s) - d^i(c^i_s))$	Plus grand nombre d'événements séparant la production de données d'une tâche i et leur consommation par une tâche j (p. 68).
	$\max_{r,w}(\text{d}^r(in) - \text{d}^w(in))$	Distance de dépendance la plus grande existant sur le tableau *in*.

⟶

Suite...

Modèle	Contrainte	Interprétation
Latence	$\mathrm{nbc}^k(\alpha) = \frac{\alpha}{\alpha_0^k} \times \prod_{i=1}^{ld^k-1} c_{max,i}^k$	Nombre d'événements actifs d'une tâche sur la période globale (p. 91).
	$\mathrm{T}^{\mathrm{s}}_{Exec} = \sum_{k\in\{nb\mid \mathrm{Stage}_{(nb)=s}\}} \left[\mathrm{nbc}^k(\alpha) \times \max(T_{cal}(k), T_{comm}(k)) + \min(T_{cal}(k), T_{comm}(k))\right]$	Durée en temps physique d'un étage de pipeline. Recouvrement des communications par des calculs. Les communications d'un bloc de calcul s'effectuent pendant l'exécution du bloc de calcul suivant du même nid de boucles (p. 102).
	$T_{cal}(k) = \det(L^k).T^k_{cal_{elem}}$	Durée physique d'un bloc de calcul. *durée d'une itération de la TE* × *nombre d'itérations dans un bloc de calcul* (p. 102).
	$T_{comm}(k) \leqslant \left\lceil \frac{\mathcal{V}^k_{\mathrm{ecr}} \times \mathrm{sizeOfData}^k_{\mathrm{ecr}} \times \mathrm{NbPEMax}_s}{\mathrm{BP}_{s,s+1}} \right\rceil + \mathrm{offsetComm}$	Durée d'un bloc de communications. *Volume total à transmettre / débit majoré d'un délai d'initialisation de la communication* (p. 103).
	$\left\lceil \frac{\Delta}{\alpha} \right\rceil . \max_s(\mathrm{T}^{\mathrm{s}}_{exec}) \leqslant \mathrm{lat}_{mspmd} \leqslant \left(\left\lceil \frac{\Delta}{\alpha} \right\rceil + 1\right) . \max_s(\mathrm{T}^{\mathrm{s}}_{exec})$	La latence ne peut être calculée qu'aux points de synchronisations globales. La durée entre deux synchronisations correspond à la durée de l'étage de pipeline le plus long à s'exécuter (p. 98).
	$\forall i \in \mathrm{NB} \setminus \mathrm{NB}^e, n_i = \max_{\{j\mid j\in \mathrm{succ}(i)\}}(a_{i,j} + n_j) \qquad \forall i \in \mathrm{NB}^e, n_i = 0$	Durée maximale, en nombre d'événements, avant la fin d'une période depuis la tâche i (p. 69).
	$\Delta = \max_{i\in \mathrm{NB}^s}(\beta^i + n_i) + 1$	Durée maximale d'une période de l'application en nombre d'événements (p. 69).
Mémoire	$\mathcal{V}^{\mathrm{rcv}}_r = \mathrm{nbc}^r(\alpha) \times \det(L^r) \times (\sum_{in} \mathrm{nba}(\mathrm{ZM}^{\mathrm{r}}_{\mathrm{in}}) \times \det(M^r_{in}))$	Volume mémoire à allouer en zone de réception pour un nid de boucle r : *Nombre d'événements* × *volume élémentaire* × *cumul* (p. 93).

⟶

Suite...

Modèle	Contrainte	Interprétation
	$\mathcal{V}^{\text{snd}}_{w,out} = 2 \times \det(L^w) \times \det(M^w_{out})$	Volume mémoire à allouer en zone d'émission pour un nid de boucle w pour le tableau *out* (p. 94).
	$2 \times \text{nbc}^r(\alpha) \times \det(L^r) \times \det(M^r_{in})$	Volume mémoire à allouer en zone de réception pour un nid de boucle r pour un tableau *in* lorsqu'il s'agit d'une réception hors application (I/O) (p. 97).
	$2 \times \text{nbc}^w(\alpha) \times \det(L^w) \times \det(M^w_{in})$	Volume mémoire à allouer en zone d'émission pour un nid de boucle w lorsqu'il s'agit d'une émission hors application (I/O) (p. 97).
	$\mathcal{V}^{\text{input}}_{r,in} = \text{nbc}\left(\max_{r,w}(\text{d}^r(in) - \text{d}^w(in))\right) \times \det(L^r) \times \det(M^r)$	Volume mémoire à allouer en zone interne du nid de boucles r pour un accès en lecture pour le tableau *in* (p. 95).
	$\mathcal{V}^{\text{output}}_{w,out} = \det(L^w) \times \det(M^w)$	Volume mémoire à allouer en zone interne du nid de boucles w pour un accès en écriture pour le tableau *out*. Ce volume est différent de 0 si une consommation de ces données se fait dans le même étage de pipeline (p. 95).
	$\mathcal{V}^{\text{int}}_{nb} = \sum_{in}\left(\mathcal{V}^{\text{input}}_{nb,in}\right) + \mathcal{V}^{\text{output}}_{nb,out}$	Volume de mémoire interne total (lecture et écriture) à allouer pour le nid de boucles nb (p. 95).
	$\forall s, \quad \mathcal{V}_s = \sum_{\{nb\mid \text{Stage}(nb)=s\}}\left(\mathcal{V}^{\text{rcv}}_{nb} + \mathcal{V}^{\text{snd}}_{nb} + \mathcal{V}^{\text{int}}_{nb}\right)$	Volume total de mémoire à allouer pour chaque étage, sur un processeur. C'est la somme du volume en émission, en réception et interne (p. 96).
Machine	$\left(\left\lceil \frac{\Delta}{\alpha} \right\rceil + 1\right).\max_s\left(\text{T}^{\text{s}}_{exec}\right) \leqslant lat_{max}$	La latence calculée ne peut excéder la latence maximum autorisée.

⟶

Suite...

Modèle	Contrainte	Interprétation
	$c_{max} = \left\lceil \frac{\imath_{max}}{LP} \right\rceil$	Nombre de cycles à effectuer pour une cible SPMD (p. 44).
	$\forall \mathrm{s}, \forall nb \in \{t \mid \mathrm{Stage}(t) = \mathrm{s}\}, \quad \det(P^{nb}) \leqslant \mathrm{NbPEMax}_s$	Le nombre de processeurs utilisés par chaque tâche dans chaque étage est borné par le nombre disponible de processeurs dans chaque étage (p. 104).
	$\forall s \quad \mathcal{V}_s \leqslant \mathcal{V}_{max}$	Contrainte de non dépassement de capacité mémoire disponible pour chaque étage sur un processeur (p. 96).
	$\forall nb^1, \forall nb^2, \quad \left(\exists c^{nb^1}, c^{nb^2} \quad \mathcal{D}\left(c^{nb^1}, c^{nb^2}\right)\right) \Rightarrow \left(0 \leqslant \mathrm{Stage}(nb^2) - \mathrm{Stage}(nb^1) \leqslant 1\right)$	Deux tâches en dépendance directe l'une de l'autre sont soit sur le même étage, soit sur des étages contigus (p. 92).
	$\mathrm{nba}\left(\mathrm{ZM}^r\right) = \left\lfloor \frac{\mathrm{d}^r}{\alpha} \right\rfloor - \left\lfloor \frac{\mathrm{d}^w}{\alpha} \right\rfloor + 1$	Nombre de périodes séparant la synchronisation globale précédant une production de données de la synchronisation globale suivant leur consommation à un instant donné (p. 93).

TAB. 6.1 – Récapitulatif des contraintes implémentées

2 La contrainte PGCD/PPCM

À plusieurs reprises au cours des différentes modélisations, nous avons eu besoin d'une contrainte ppcm (pour la définition de la période α) ou d'une contrainte pgcd (pour l´élimination partielle de la sur-approximation des dépendances). Rappelons leur définition respective :

Définition 6.2-1 :

Soient deux entiers positifs X et Y. Si l'on note p_i le i^{e} nombre premier, alors X et Y s'écrivent de manière unique :

$$X = \prod_{i=1}^{\infty} p_i^{k_i} \text{ avec } k_i \geqslant 0$$

$$Y = \prod_{i=1}^{\infty} p_i^{k'_i} \text{ avec } k'_i \geqslant 0$$

Le plus grand commun diviseur de X et Y est le nombre défini par :

$$\text{pgcd}(X,Y) = \prod_{i=1}^{\infty} p_i^{\min(k_i,k'_i)}$$

Le plus petit commun multiple de X et Y est le nombre défini par :

$$\text{ppcm}(X,Y) = \prod_{i=1}^{\infty} p_i^{\max(k_i,k'_i)}$$

Ces définitions permettent d'illustrer simplement le fait que ces opérateurs ne sont pas monotones. En effet, prenons par exemple 12 et X une inconnue de $\mathbb{N}^*$. Tant que l'on n'a pas d'information sur X, il est difficile de déduire autre chose de plus que $1 \leqslant \text{pgcd}(12,X) \leqslant 12$ et $12 \leqslant \text{ppcm}(12,X)$. Admettons que X ait pour valeurs possibles $\{24,25,26\}$. Pour chacune de ces valeurs on a :

$X = 24$: $\text{pgcd}(12,24) = 12 \quad \text{ppcm}(12,24) = 24$

$X = 25$: $\text{pgcd}(12,25) = 1 \quad \text{ppcm}(12,25) = 300$

$X = 26$: $\text{pgcd}(12,26) = 2 \quad \text{ppcm}(12,26) = 156$

Le comportement de chacun des opérateurs pgcd et ppcm n'est pas monotone : le premier décroît puis croît, alors que le second croît puis décroît, tandis que l'on énumère dans l'ordre croissant les valeurs possibles de X.

La monotonie est une des propriétés primordiales qu'une contrainte doit satisfaire pour que le seul événement qui augmente la taille du domaine d'une variable soit le *retour arrière*. Le principe de la propagation de contrainte est d'éliminer des domaines des variables les valeurs qui ne peuvent apparaître dans la solution en cours de construction. C'est pourquoi, si l'on s'attend à ce qu'une contrainte PGCD propage assez peu, elle doit tout de même le faire le plus possible et rapidement. La solution qui consiste à calculer tous les pgcd possible, de couples de valeurs (x,y) ($x \in \mathfrak{D}_X$ et $y \in \mathfrak{D}_Y$), est ce qui peut propager le plus, mais ce qui prend également le plus de temps dès que les domaines des variables sont quelque peu importants. Ce n'est donc pas dans cette direction que nous sommes partis.

Cette section décrit comment nous sommes parvenus à exprimer, dans le monde de la PPC, les opérateurs PGCD et PPCM. Nous n'avons pas exactement défini une contrainte PGCD-PPCM, mais une meta-contrainte dans la mesure où c'est un ensemble de contraintes élémentaires qui sert à cette définition.

Le langage utilisé dans les exemples de code est le langage Claire[14] (Eclair est le framework contraintes associé à ce langage).

2.1 L'algorithme d'Euclide

Les premiers essais d'écriture d'une contrainte pgcd ont consisté à programmer l'algorithme d'Euclide. Il repose sur les deux propriétés suivantes du pgcd.

Propriété 6.2-1 :

Si on suppose deux entiers X et Y tels que $X \geqslant Y$, alors :
$\text{pgcd}(X, Y) = \text{pgcd}(Y, X \bmod Y)$

Propriété 6.2-2 :

Pour tout entier X, on a : $\text{pgcd}(X, 0) = X$

Dès lors l'algorithme d'Euclide consiste, de façon récursive, à appliquer la propriété 6.2-1 jusqu'à pouvoir appliquer la propriété 6.2-2. La figure 6.1 donne le code correspondant à cet algorithme appliqué à deux entiers.

FIG. 6.1 – Algorithme d'Euclide pour le calcul du pgcd de deux constantes entières

```
// On suppose X >= Y
[recursive_pgcd(X:integer, Y:integer) : integer
-> if (Y = 0) X
    else
      pgcd(Y, mod(X, Y))
]
```

Si l'on souhaite exploiter ce programme pour définir la meta-contrainte PGCD, il faut dans un premier temps éliminer cette récursivité (fig. 6.2).

FIG. 6.2 – Algorithme d'Euclide, version itérative

```
// On suppose X >= Y
[iterative_pgcd(X:integer, Y:integer) : integer
-> let pgcd_res := X in
     (while (Y > 0) (
        pgcd_res := mod(X, Y),
        X        := Y,
        Y        := pgcd_res,
        pgcd_res := X
        ),
      pgcd_res
     )
]
```

Dans un deuxième temps, il faut éliminer toutes les dépendances d'écriture et les anti-dépendances de façon à rendre indépendantes les instructions utilisant les même variables, mais à des fins différentes. Bien que la méthode soit similaire, on ne cherche pas à paralléliser

l'algorithme d'Euclide. L'idée est de remplacer les variables scalaires (`X`, `Y`, `pgcd_res`) par des variables de domaines. Ainsi, on aura une version d'une meta-contrainte pour le PGCD.

L'algorithme d'Euclide converge en au plus $\lfloor 2\log_2 \max(X,Y)\rfloor + 1$ étapes ([65, théorème 4.2.1 page 84]). Ce qui signifie que l'on a besoin d'au plus $\lfloor 2\log_2 \max(X,Y)\rfloor + 3$ variables distinctes pour éliminer les dépendances citées ci-dessus. En effet, à chaque itération, deux variables de l'itération précédente sont utilisées. Par ailleurs, on veut également que le nombre d'itérations effectuées soit déterminé. On obtient alors le code de la figure 6.3, toujours appliqué à deux constantes entières.

FIG. 6.3 – Algorithme d'Euclide, version itérative, assignation unique

```
// On suppose X >= Y
[iterative_pgcd_wo_dep(X:integer, Y:integer) : integer
-> let a := make_list(integer!(2. * Log2(X)) + 3, 1),
       i := 2 in
     (a[1] := X,
      a[2] := Y,
      for loop in (1 .. integer!(2. * Log2(X)) + 1) (
        if (a[i] > 0) (
          a[i + 1] := mod(a[i - 1], a[i]),
          i :+ 1
        )
      ),
      a[i - 1]
     )
]
```

Il reste encore un point à traiter avant d'obtenir la meta-contrainte PGCD à partir de cet algorithme : le calcul du modulo de `X` par `Y`. Sachant qu'il correspond au reste de la division Euclidienne $X \div Y$, il nous faut introduire une nouvelle variable correspondant au quotient de ce rapport. Nous pouvons désormais remplacer les variables scalaires par des variables de domaine. Ainsi la meta-contrainte PGCD est donnée par le code de la figure 6.4.

Quelle peut être l'efficacité d'une telle contrainte ? Avant même de faire les premiers tests, nous pouvons nous douter qu'elle ne sera pas bonne. En effet, de part leur nature, les contraintes gardées sont connues pour ne pas être très efficaces. Par ailleurs, la contrainte *produit* propage peu et, de surcroît, intervient sur les variables `q[i]` qui ne sont contraintes nulle part ailleurs. De plus, cette approche est monodirectionnelle, puisque les seules déductions possibles se font à partir de X et Y et jamais depuis leur pgcd. Enfin, du simple point de vue de l'allocation mémoire, chaque pose de cette contrainte pgcd alloue $4 \times \log_2(X_{sup}) + 5$ variables de domaines. Dans la mesure où la plus grande valeur possible est `DSUP`$=10^{15}$, cela peut représenter 204 variables de domaines pour 1 pgcd de deux variables. Ce surplus d'allocation influence énormément la résolution car ce sont autant de variables inutiles participant à la propagation, et augmentant considérablement le nombre de points de choix de la résolution.

Cette étude permet de montrer qu'une approche algorithmique pour résoudre un problème de contraintes n'est pas une bonne solution. La PPC étant issue (en partie) de la

FIG. 6.4 – Meta-contrainte PGCD déduite de l'algorithme d'Euclide

```
// On suppose X >= Y
[metaConstraint_pgcd(X:Var, Y:Var) : Var
-> let a   := makeVarList("A", integer!(2. * Log2(X.sup)) + 3, 0, Y.sup),
       q   := makeVarList("Q", integer!(2. * Log2(X.sup)) + 1, 0, Dsup),
       res := makeVar("PGCD", 0, Y.sup),
       i   := 2 in
     (a[1] == X,
      a[2] == Y,
      for loop in (1 .. integer!(2. * Log2(X.sup)) + 1) (
        (a[i] >? 0) implies ((a[i - 1] ==? q[i] *? a[i] +?  a[i + 1]) and?
                             (a[i + 1] <? a[i - 1])),
        ((a[i] ==? 0) and? (a[i - 1] >? 0)) implies (res ==? a[i - 1]),
        i :+ 1
        )
      ),
      res
     )
]
```

programmation logique, elle est déclarative et non algorithmique. C'est pourquoi nous nous sommes intéressés aux différentes propriétés du pgcd de deux entiers. Elles vont nous permettre de le caractériser. Comme elles coïncident en partie avec celles du ppcm de deux entiers, notre contrainte est double et fait intervenir les deux notions.

Le résultat obtenu est plus efficace en temps, en qualité de propagation et en nombre d'allocations mémoire que celui que nous venons de décrire.

2.2 Caractérisation du pgcd et du ppcm par leur propriétés

Le pgcd et le ppcm de deux entiers x et y ont plusieurs propriétés que nous allons présenter. Utilisant l'aspect déclaratif de la PPC, elles vont nous permettre d'écrire une meta-contrainte liant deux entiers à leur pgcd et leur ppcm, ainsi que de profiter de la bi-directionnalité des relations pour augmenter la qualité de la propagation.

Notons tout d'abord que pour tout entiers x et y de $\mathbb{Z}$, on a :

$$\text{ppcm}(x,y) = \text{ppcm}(|x|,|y|) \geqslant 0 \quad (6.1) \checkmark$$

$$\text{pgcd}(x,y) = \text{pgcd}(|x|,|y|) \geqslant 0 \quad (6.2)$$

Par conséquent nous considérons dans la suite que x et y appartiennent à $\mathbb{N}$.

2.2.1 Propriétés conjointes pgcd, ppcm

Dès que l'on aborde des problèmes de multiples ou de diviseurs, l'élément absorbant de la multiplication, 0, pose des problèmes. Aussi les conventions suivantes sont classiquement

[31] adoptées :

$$\forall x \in \mathbb{N} \quad \text{pgcd}(0,x) = x \quad (\text{pgcd}(0,0) = 0) \tag{6.3}$$
$$\forall x \in \mathbb{N} \quad \text{ppcm}(0,x) = 0 \tag{6.4}$$
$$\forall x \in \mathbb{N} \quad \text{pgcd}(x,x) = x \tag{6.5}$$
$$\forall x \in \mathbb{N} \quad \text{ppcm}(x,x) = x \tag{6.6}$$

Supposons désormais que x et y sont non nuls. Par définition, le pgcd de x et y divise l'un et l'autre. Autrement dit, il existe deux entiers non nuls d_x et d_y tels que :

$$\text{pgcd}(x,y) \times d_x = x \tag{6.7}$$
$$\text{pgcd}(x,y) \times d_y = y \tag{6.8}$$

En reprenant les définitions, on a :

$$\begin{aligned}
\text{pgcd}(x,y) &= \prod_{i=1}^{\infty} p_i^{\min(k_i,k'_i)} \\
&= \prod_{i=1}^{\infty} p_i^{\min(k_i,k'_i)-k_i} . p_i^{k_i} \\
&= \prod_{i=1}^{\infty} p_i^{\min(k_i,k'_i)-k_i} . \prod_{i=1}^{\infty} p_i^{k_i} \\
x &= \text{pgcd}(x,y) . \prod_{i=1}^{\infty} p_i^{k_i-\min(k_i,k'_i)} \\
d_x &= \prod_{i=1}^{\infty} p_i^{k_i-\min(k_i,k'_i)}
\end{aligned}$$

De même, on a :

$$\begin{aligned}
y &= \text{pgcd}(x,y) . \prod_{i=1}^{\infty} p_i^{k'_i-\min(k_i,k'_i)} \\
d_y &= \prod_{i=1}^{\infty} p_i^{k'_i-\min(k_i,k'_i)}
\end{aligned}$$

Par ailleurs on a également :

$$\text{pgcd}(x,y) \times \text{ppcm}(x,y) = x \times y \tag{6.9}$$

Ce qui nous permet de déduire :

$$\text{ppcm}(x,y) = d_x.y \quad (6.10)$$
$$\text{ppcm}(x,y) = d_y.x \quad (6.11)$$

En effet,

$$\begin{aligned} \text{pgcd}(x,y) \times \text{ppcm}(x,y) &= x \times y \\ &= \text{pgcd}(x,y) \times d_x \times y \\ &\quad \text{x et y étant strictement positifs } \text{pgcd}(x,y) > 0 \\ \text{ppcm}(x,y) &= d_x \times y \end{aligned}$$

de même pour (6.11).

En utilisant les différentes propriétés énoncées ci-dessus et en les combinant les unes avec les autres, nous obtenons :

$$\text{ppcm}(x,y) = d_x \times d_y \times \text{pgcd}(x,y) \quad (6.12)$$ ✓

Et pour tout entiers x et y strictement positifs :

$$\max(x,y) \leqslant \text{ppcm}(x,y) \leqslant x \times y \quad (6.13)$$ ✓
$$0 < \text{pgcd}(x,y) \leqslant \min(x,y) \quad (6.14)$$

Enfin, par définition du pgcd et du ppcm, d_x et d_y sont premiers entre eux. C'est-à-dire, par application du théorème de Bezout :

$$\exists (u,v) \in \mathbb{Z}^2, u \times d_x + v \times d_y = 1 \quad (6.15)$$ ✓

Voyons comment adapter ces propriétés dans un contexte de Programmation Par Contraintes.

2.2.2 Utilisation des propriétés dans un contexte de PPC

Le plus difficile à caractériser dans la notion de pgcd n'est pas la partie *diviseur commun*, mais *plus grand*. Les contraintes 6.7, 6.8 (resp. 6.10, 6.11) suffisent à définir l'aspect diviseur (resp. multiple) commun. Les suivantes sont des contraintes redondantes pour améliorer la propagation.

La notion de *plus grand* diviseur commun est obtenue en imposant à d_x et d_y d'être premiers entre eux (que nous noterons $d_x \perp d_y$). C'est le théorème de Bezout (condition 6.15) qui le permet. Mais il fait intervenir deux variables u et v sur lesquelles nous n'avons pas d'information. Cependant nous pouvons agir sur leur domaine d'existence respectif.

Tout d'abord nous pouvons les réduire à $\mathbb{N}$. En effet, d_x et d_y étant des entiers positifs, on a équivalence entre les deux expressions :

$$\exists (u',v') \in \mathbb{Z}^2, u' \times d_x + v' \times d_y = 1$$
$$\exists (u,v) \in \mathbb{N}^2, -u \times \min(d_x,d_y) + v \times \max(d_x,d_y) = 1$$

Ces variables u et v sont respectivement bornées par d_y et d_x. Le théorème suivant nous garantit leur existence (résultat classique d'arithmétique à la Gauss).

Théorème 6.2-1 : Soient $x \in \mathbb{N}$ et $y \in \mathbb{N}^*$ et $x \leqslant y$. Si $x \perp y$, alors

$$\exists (u, v) \text{ t.q. } 0 \leqslant u < y \text{ et } 0 \leqslant v \leqslant x, \text{ t.q.} \qquad -u \times x + v \times y = 1$$

Grâce à ce théorème, nous avons réduit les domaines de u et v, que nous avons mis en relation avec d_x et d_y.

$$0 \leqslant v \leqslant d_x \tag{6.16}$$
$$0 \leqslant u < d_y \tag{6.17}$$

Ainsi, au travers de ces propriétés, nous avons une contrainte (double) :

$$Z = \text{pgcd}(X, Y)$$
$$W = \text{ppcm}(X, Y)$$

L'utilisation conjointe des contraintes 6.1 à 6.17 donne la possibilité de définir la meta-contrainte PGCD-PPCM de deux entiers.

2.3 Conclusion

La mise au point de cette contrainte PGCD-PPCM nous a permis d'illustrer qu'une approche contrainte n'est pas une approche algorithmique. L'utilisation de différentes propriétés du pgcd et du ppcm les rend déclaratifs, les relie aux variables sur lesquelles ils s'appliquent et offre une bonne propagation par rapport aux possibilités offertes vue leur nature *chaotique*.

Malgré tout, il reste encore un travail d'étude du comportement unifié de ces contraintes à effectuer. Nous pourrions ainsi éliminer les variables superflues (d_x, d_y, u, v et toutes les variables muettes introduites par l'utilisation de certaines contraintes élémentaires), et aboutir ainsi à une «vraie» contrainte *(W, Z) = PGCD-PPCM(X, Y)*.

Chapitre 7

Expérimentations

Ces expérimentations ont pour objet notre modèle de machines M-SPMD. La combinatoire induite par notre modèle, ainsi que le fait qu'elle augmente de façon exponentielle en fonction du nombre de tâches (et de la profondeur des nids de boucles) de l'application à placer, ne nous permet pas de faire nos expériences avec l'application MFR présentée dans ce document[1].

Les expérimentations qui suivent ont été réalisées sur une application constituée de trois tâches FFT, ENERGIE et SUM. Le pseudo-code Fortran correspondant est donné par la figure 7.1).

Cette application est exemplaire car elle permet quatre configurations possibles de distribution de ses tâches dans le pipeline. Il est possible de mettre chacun des nids de boucles dans un étage distinct des autres, ou bien d'associer le premier au deuxième et mettre le troisième nid de boucles dans un deuxième étage, ou encore d'associer le deuxième avec le troisième et mettre le premier nid de boucles dans le premier étage de pipeline. Malgré le changement d'axe, il est tout de même possible de trouver une quatrième option possible en mettant les trois tâches dans le même étage (parallélisation sur `t` ou sur `v`).

La recherche de solutions de placement de cette application sur une machine M-SPMD s'est déroulée sous les hypothèses initiales suivantes :

- Critère d'amélioration : latence
- Nombre d'étages du pipeline : 3
- Nombre de processeurs par étage :
  - étage 1 : 8 PE disponibles
  - étage 2 : 4 PE disponibles
  - étage 3 : 2 PE disponibles
- Durée d'un cycle machine : 20 ns. Les calculs de temps physiques se font en cycles machine. Cette donnée permet la conversion d'un nombre de cycles en durée.
- Capacité mémoire disponible par processeur : 307 200 octets

[1] Il est cependant à noter que cette application a été placée par notre outil sur une architecture parallèle SPMD, dans des conditions architecturales identiques à celles de [32]. Nous sommes désormais capable de placer automatiquement des applications 4 à 5 fois plus importantes en taille. Cependant, la combinatoire étant très importante, les modèles étant très mathématiques et peu «contraintes», trouver une solution optimale, ou une bonne solution, est très long.

FIG. 7.1 – Application test pour le mode MSPMD

```
#[FFT] FFT 1D complexe sur un vecteur de fréquence
DO t = 0, ...
  DO v = 0, 3
    DO cd = 0, 99
      FREQ[t, v, c, 0:255] = FFT(INPUT[t, v, c, 0:255])

#[ENERGIE] calcul d'énergie
DO t = 0, ...
  DO v = 0, 3
    DO cd = 0, 99
      DO rec = 0, 255
        ENERG[t, v, cd, rec] = ENERGIE(FREQ[t, v, cd, rec])

#[SUM] sommation sur les capteurs pour chaque fréquence
DO t =0, ...
  DO v = 0, 3
    DO rec = 0, 255
      ENERG_F[t, v, rec] = SUM(ENERG[t, v, 0:99, rec])
```

- Latence maximale autorisée : 1 000 000 000 ns
- Nombre de zone d'acquisition : 1 (tableau INPUT)
  - Période d'acquisition : 3 000 000 ns
  - Nombre d'octets par acquisition : 819 200 octets
- Débit de communication entre étages :
  - de l'étage 1 à l'étage 2 : 314 572 800 octets/s
    délai d'initialisation de la communication : 50 000 ns
  - de l'étage 2 à l'étage 3 : 104 857 600 octets/s
    délai d'initialisation de la communication : 50 000 ns

Les temps de résolution correspondant à chacune de ces solutions vont de 1 à 4 minutes. L'algorithme de résolution est incomplet, cela signifie que si l'on change les conditions initiales (en terme de description d'une solution partielle) les résultats obtenus peuvent être différents.

Ce chapitre a pour objet de décrire la stratégie de résolution adoptée pour la résolution du placement. La plupart des heuristiques utilisées dans [32] sont réutilisées. Les sections suivantes décrivent les différentes catégories de solutions que nous avons trouvées.

1 Stratégie de résolution

1.1 Heuristique de choix de variables

Nous avons vu dans le chapitre 1 (sec. 1) que le placement repose principalement sur les deux modèles suivants : partitionnement et ordonnancement (l'hyper-relation *dépendances*

permettant de les corréler). Par conséquent les variables décrivant ces modèles (P, L, c_{max}, *alpha* et *beta*) ont une place prépondérante dans l'arbre de recherche. Par ailleurs, du fait du modèle M-SPMD, de nouvelles variables s'ajoutent à sa structure. L'étage dans lequel s'exécute une tâche en fait partie.

Le choix des variables est statique. L'arbre de recherche correspondant est construit suivant cet ordre :

Pour chaque tâche w :

1. l'étage dans lequel elle s'exécute,
2. la permutation que l'on effectue sur les boucles externes,
3. la matrice P^w de partitionnement (processeurs),
4. la matrice L^w de partitionnement (localité mémoire),
5. le vecteur α^w de son ordonnancement,
6. le vecteur c^r_{max} de la tâche r qui lit les données produites par w,
7. le start-up β^r de la tâche r qui lit les données produites par w.

Ainsi de suite pour toutes les tâches.

Ce choix a l'avantage de propager relativement tôt des contraintes sur les noeuds finaux du graphe flot de données.

La dernière variable de l'arbre est la période logique globale α, mais c'est principalement dû au fait qu'il s'agit du PPCM des périodes logiques des blocs de calculs de chacune des tâches. En effet, la meta-contrainte PGCD-PPCM (sec. 2) n'est pas complète au sens où une énumération est nécessaire pour obtenir l'instanciation du pgcd ou du ppcm de deux entiers, lorsque ceux-ci sont déterminés. Cette énumération n'est rien d'autre que l'affectation au pgcd de sa borne supérieure (le ppcm est alors directement déduit), et inversement pour le ppcm. Les autres valeurs de ces domaines conduisent à une contradiction.

1.2 Heuristiques de choix de valeurs

De part la faible capacité de propagation des contraintes posées −produits et sommes pour les contraintes décrivant les directives opérationnelles, telles que la latence ou la capacité mémoire−, nous avons opté pour un algorithme de recherche incomplet. Dans la mesure où nous ne cherchons pas nécessairement une solution optimale, cela permet de réduire la combinatoire résultante.

Cet incomplétude se traduit par le fait que l'on n'autorise pas de retour arrière sur certaines variables. C'est le cas pour les coefficients d'ordonnancement qui ne sont énumérés que sur la borne inférieure de leur domaine, de même que pour les permutations (ce qui donne des solutions sans permutations des boucles, sauf si elles sont rendues explicites).

Le choix d'attribution d'une tâche à un étage est dirigé par la recherche de solutions utilisant le moins possible d'étages. C'est-à-dire que l'on énumère les domaines de ces variables dans l'ordre croissant. L'intérêt ici est de minimiser le nombre de communications et favoriser la localité en bénéficiant des conditions de non-communication.

Cependant, comme nous savons que la meilleure solution qui minimise les communications est une solution mono-processeur, nous allons chercher, dans le même temps, à maximiser le parallélisme. Il suffit d'énumérer dans l'ordre décroissant les domaines des coefficients des matrices P. Le choix des valeurs est tel qu'elles divisent les bornes supérieures des boucles non

partitionnées (de même pour les coefficients des matrices L, mais avec une énumération des domaines dans l'ordre croissant). Ainsi sont éliminées toutes les valeurs dont on sait qu'elles ne permettront pas d'atteindre une solution, et ce sans attendre la propagation aboutissant à une contradiction.

Pourquoi utiliser un algorithme incomplet décrit de cette façon ? À cela il y a deux raisons principales. La première est que l'obtention d'un modèle contrainte efficace demande encore une forte charge de travail. La piste la plus intéressante semble être l'utilisation de contraintes globales. La contrainte globale cumulative[2] semble être la plus prometteuse pour les modèles de latence ou de capacité mémoire. Il est cependant à noter que dans notre problème, les blocs de calcul font partie de la recherche et du même coup les ressources qu'ils occupent. Il faut alors se placer à un degré de granularité plus fin, c'est-à-dire la tâche élémentaire. Un deuxième point à noter est qu'une telle contrainte n'est pas, à l'heure actuelle, disponible dans notre solveur Eclair.

La deuxième raison est qu'il n'est pas possible de borner les coefficients des ordonnancements de manière formelle ; c'est-à-dire autrement que par une valeur arbitraire. Leur domaine respectif étant très grand, une énumération complète n'est pas envisageable.

2 Une tâche par étage, parallélisation de l'axe *temps*

Nombre de contraintes : 2507

Nombre de variables de décision : 60

Les tâches sont réparties dans les étages du pipeline ainsi :

	FFT	ENERGIE	SUM
Étage	1	2	3

Nombre de processeurs utilisés par étage :

- Étage 1 : 8 PE (sur 8 disponibles)
- Étage 2 : 4 PE (sur 4 disponibles)
- Étage 3 : 2 PE (sur 2 disponibles)

L'ordre des boucles dans chacun des nids reste inchangé (les matrices de permutations sont l'identité).

2.1 Partitionnement - Ordonnancement

On décrit ici, tâche par tâche, tous les paramètres associés au partitionnement et à l'ordonnancement.

	FFT	ENERGIE	SUM
Liste des itérateurs de boucles	(t, v, cd)	(t, v, cd, rec)	(t, v, rec)
Le vecteur d'itération initial	$(\infty, 4, 100)$	$(\infty, 4, 100, 256)$	$(\infty, 4, 256)$
La diagonale de la matrice P	(8, 1, 1)	(4, 1, 1, 1)	(2, 1, 1)
La diagonale de la matrice L	(1, 4, 100)	(2, 4, 100, 256)	(8, 4, 256)
Le vecteur c_{max}	$(\infty, 1, 1)^t$	$(\infty, 1, 1, 1)^t$	$(\infty, 1, 1)^t$
Le vecteur α	$(1, 1, 1)^t$	$(1, 1, 1, 1)^t$	$(1, 1, 1)^t$
Le temps de start-up	$\beta = 0$	$\beta = 1$	$\beta = 3$

Pour cette solution le partitionnement trouvé effectue une parallélisation temporelle. L'expérience montre qu'une solution de ce type est inefficace en latence et en mémoire. C'est ce

que nous allons constater dans la suite[2] par comparaison avec les autres solutions.

2.2 Détection des communications

Les trois étages sont actifs, par conséquent il existe une communication entre l'étage 1 et l'étage 2, et une communication entre l'étage 2 et l'étage 3. La première a comme tâche émettrice FFT et comme tâche réceptrice ENERGIE. Le second lien de communication est ouvert entre la tâche ENERGIE (émission) et la tâche SUM (réception). Le tableau suivant donne les volumes de données à transmettre ainsi que les durées des transmissions. Ces volumes dépendent de la taille des blocs de calculs (partitionnement).

	Volume à transmettre	Débit de transmission	Durée
FFT→ENERGIE	819 200 octets	314 572 800 octets/s	2 654 167 ns
ENERGIE→SUM	819 200 octets	104 857 600 octets/s	7 862 500 ns

2.3 Latence

La distance maximale de dépendance est de **4** événements (longueur du chemin critique du graphe). Une période de synchronisation dure **1 événement**, et le nombre de périodes sur la durée du chemin critique est de **4**.

La durée physique est le maximum des durées de chaque étage, sachant que la durée physique de l'étage numéro

1. est de **28 574 167 ns**.
2. est de **20 150 500 ns**.
3. est de **16 384 000 ns**.

Ainsi la durée en temps réel d'une période de synchronisation est de **28 574 167 ns**. De ce fait, la latence applicative est de **114 296 668 ns**.

Le tableau suivant récapitule les durées tâche par tâche.

	FFT	ENERGIE	SUM
Nombre de blocs de calcul sur une période	1	1	1
Durée d'un bloc de calcul	25 920 000 ns	12 288 000 ns	16 384 000 ns
Durée d'une communication	2 654 167 ns	7 862 500 ns	0 ns
Durée (comm/calcul)	25 920 000 ns	12 288 000 ns	16 384 000 ns
Coût sur une période	28 574 167 ns	20 150 500 ns	16 384 000 ns

[2]Afin de d'obtenir un exemple illustrant cette situation, on ne borne ni la mémoire ni la latence.

2.4 Mémoire

	FFT	ENERGIE	SUM
Lecture	204 800 octets	409 600 octets	2 457 600 octets
Volume interne	0 octets	0 octets	0 octets
Volume de communication	0 octets	409 600 octets	2 457 600 octets
Volume externe	204 800 octets	0 octets	0 octets
Écriture	204 800 octets	409 600 octets	16 384 octets
Volume interne	0 octets	0 octets	0 octets
Volume de communication	204 800 octets	409 600 octets	0 octets
Volume externe	0 octets	0 octets	16 384 octets
Total	409 600 octets	819 200 octets	2 473 984 octets

Allocation mémoire par processeur et par étage utilisé :
- Volume mémoire utilisé par l'étage 1 : 409 600 octets
- Volume mémoire utilisé par l'étage 2 : 819 200 octets
- Volume mémoire utilisé par l'étage 3 : 2 473 984 octets

3 Pas de communication, 2 processeurs

Les tâches sont réparties dans les étages du pipeline ainsi :

	FFT	ENERGIE	SUM
Étage	1	1	1

Nombre de processeurs utilisés par étage :
- Étage 1 : 2 PE (sur 8 disponibles)
- Étage 2 : 0 PE (sur 4 disponibles)
- Étage 3 : 0 PE (sur 2 disponibles)

L'ordre des boucles dans chacun des nids reste inchangé (les matrices de permutations sont l'identité).

3.1 Partitionnement - Ordonnancement

On décrit ici, tâche par tâche, tous les paramètres associés au partitionnement et à l'ordonnancement.

	FFT	ENERGIE	SUM
Liste des itérateurs de boucles	(t, v, cd)	(t, v, cd, rec)	(t, v, rec)
Le vecteur d'itération initial	$(\infty, 4, 100)$	$(\infty, 4, 100, 256)$	$(\infty, 4, 256)$
La diagonale de la matrice P	(1, 2, 1)	(1, 2, 1, 1)	(1, 2, 1)
La diagonale de la matrice L	(1, 1, 100)	(1, 1, 100, 256)	(1, 1, 256)
Le vecteur c_{max}	$(\infty, 2, 1)^t$	$(\infty, 2, 1, 1)^t$	$(\infty, 2, 1)^t$
Le vecteur α	$(6, 3, 3)^t$	$(6, 3, 3, 3)^t$	$(6, 3, 3)^t$
Le temps de start-up	$\beta = 0$	$\beta = 4$	$\beta = 8$

Pour cette solution le partitionnement trouvé effectue une parallélisation sur les voies. Cette distribution permet d'éviter les communications puisque les itérations nécessitant un changement d'axe restent locales aux processeurs sur lesquels elles s'exécutent.

3.2 Détection des communications

Les trois tâches étant sur le même étage du pipeline, il n'y a pas de communications.

3.3 Latence

La distance maximale de dépendance est de **15** événements (longueur du chemin critique du graphe). Une période de synchronisation dure **6 événements**, et le nombre de périodes sur la durée du chemin critique est de **3**.

La durée en temps réel d'une période de synchronisation est de **17 056 000 ns**. De ce fait, la latence applicative est de **51 168 000 ns**.

Le tableau suivant récapitule les durées tâche par tâche.

	FFT	ENERGIE	SUM
Nombre de blocs de calcul sur une période	2	2	2
Durée d'un bloc de calcul	6 480 000 ns	1 536 000 ns	512 000 ns
Durée d'une communication	0 ns	0 ns	0 ns
Durée (comm/calcul)	6 480 000 ns	1 536 000 ns	512 000 ns
Coût sur une période	12 960 000 ns	3 072 000 ns	1 024 000 ns

3.4 Mémoire

	FFT	ENERGIE	SUM
Lecture	102 400 octets	25 600 octets	25 600 octets
Volume interne	0 octets	25 600 octets	25 600 octets
Volume de communication	0 octets	0 octets	0 octets
Volume externe	102 400 octets	0 octets	0 octets
Écriture	25 600 octets	25 600 octets	1 024 octets
Volume interne	25 600 octets	25 600 octets	0 octets
Volume de communication	0 octets	0 octets	0 octets
Volume externe	0 octets	0 octets	1 024 octets
Total	128 000 octets	51 200 octets	26 624 octets

L'allocation mémoire par processeur de l'étage 1 est de 205 824 octets.

4 Pas de communication, 4 processeurs

Les tâches sont réparties dans les étages du pipeline ainsi :

	FFT	ENERGIE	SUM
Étage	1	1	1

Nombre de processeurs utilisés par étage :
- Étage 1 : 4 PE (sur 8 disponibles)
- Étage 2 : 0 PE (sur 4 disponibles)
- Étage 3 : 0 PE (sur 2 disponibles)

L'ordre des boucles dans chacun des nids reste inchangé (les matrices de permutations sont l'identité).

4.1 Partitionnement - Ordonnancement

On décrit ici, tâche par tâche, tous les paramètres associés au partitionnement et à l'ordonnancement.

	FFT	ENERGIE	SUM
Liste des itérateurs de boucles	(t, v, cd)	(t, v, cd, rec)	(t, v, rec)
Le vecteur d'itération initial	(∞, 4, 100)	(∞, 4, 100, 256)	(∞, 4, 256)
La diagonale de la matrice P	(1, 4, 1)	(1, 4, 1, 1)	(1, 4, 1)
La diagonale de la matrice L	(1, 1, 100)	(1, 1, 100, 256)	(1, 1, 256)
Le vecteur c_{max}	$(\infty, 1, 1)^t$	$(\infty, 1, 1, 1)^t$	$(\infty, 1, 1)^t$
Le vecteur α	$(3, 3, 3)^t$	$(3, 3, 3, 3)^t$	$(3, 3, 3)^t$
Le temps de start-up	$\beta = 0$	$\beta = 1$	$\beta = 2$

Comme pour la solution précédente, le partitionnement trouvé effectue une parallélisation sur les voies. Cette distribution permet d'éviter les communications puisque les itérations nécessitant un changement d'axe restent locales aux processeurs sur lesquels elles s'exécutent.

4.2 Détection des communications

Les trois tâches étant sur le même étage du pipeline, il n'y a pas de communications.

4.3 Latence

La distance maximale de dépendance est de **3** événements (longueur du chemin critique du graphe). Une période de synchronisation dure **3 événements**, et le nombre de périodes sur la durée du chemin critique est de **1**.

La durée en temps réel d'une période de synchronisation est de **8 528 000 ns**. De ce fait, la latence applicative est de **8 528 000 ns**.

Le tableau suivant récapitule les durées tâche par tâche.

	FFT	ENERGIE	SUM
Nombre de blocs de calcul sur une période	1	1	1
Durée d'un bloc de calcul	6 480 000 ns	1 536 000 ns	512 000 ns
Durée d'une communication	0 ns	0 ns	0 ns
Durée (comm/calcul)	6 480 000 ns	1 536 000 ns	512 000 ns
Coût sur une période	6 480 000 ns	1 536 000 ns	512 000 ns

Cette solution est dix fois plus rapide que la précédente, alors que la différence, au niveau du partitionnement, est le nombre de processeurs utilisé qui est doublé. Il y a donc plus de calculs simultanés. Cette différence impacte directement l'ordonnancement des calculs qui peuvent se faire dans la même période. D'où cette accélération.

4.4 Mémoire

	FFT	ENERGIE	SUM
Lecture	51 200 octets	25 600 octets	25 600 octets
Volume interne	0 octets	25 600 octets	25 600 octets
Volume de communication	0 octets	0 octets	0 octets
Volume externe	51 200 octets	0 octets	0 octets
Écriture	25 600 octets	25 600 octets	512 octets
Volume interne	25 600 octets	25 600 octets	0 octets
Volume de communication	0 octets	0 octets	0 octets
Volume externe	0 octets	0 octets	512 octets
Total	76 800 octets	51 200 octets	26 112 octets

L'allocation mémoire par processeur de l'étage 1 est de 205 824 octets. Cet occupation mémoire est identique à la solution précédente car la dimension parallélisée (les voies v) permet un découpage sans dépendance.

5 Une tâche par étage

Les tâches sont réparties dans les étages du pipeline ainsi :

	FFT	ENERGIE	SUM
Étage	1	2	3

Nombre de processeurs utilisés par étage :
- Étage 1 : 8 PE (sur 8 disponibles)
- Étage 2 : 4 PE (sur 4 disponibles)
- Étage 3 : 1 PE (sur 2 disponibles)

L'ordre des boucles dans chacun des nids reste inchangé (les matrices de permutations sont l'identité).

5.1 Partitionnement - Ordonnancement

On décrit ici, tâche par tâche, tous les paramètres associés au partitionnement et à l'ordonnancement.

	FFT	ENERGIE	SUM
Liste des itérateurs de boucles	(t, v, cd)	(t, v, cd, rec)	(t, v, rec)
Le vecteur d'itération initial	$(\infty, 4, 100)$	$(\infty, 4, 100, 256)$	$(\infty, 4, 256)$
La diagonale de la matrice P	(1, 2, 4)	(1, 1, 2, 2)	(1, 1, 1)
La diagonale de la matrice L	(1, 2, 25)	(1, 4, 50, 128)	(1, 4, 256)
Le vecteur c_{max}	$(\infty, 1, 1)^t$	$(\infty, 1, 1, 1)^t$	$(\infty, 1, 1)^t$
Le vecteur α	$(1, 1, 1)^t$	$(1, 1, 1, 1)^t$	$(1, 1, 1)^t$
Le temps de start-up	$\beta = 0$	$\beta = 1$	$\beta = 2$

5.2 Détection des communications

Les trois étages sont actifs, par conséquent il existe une communication entre l'étage 1 et l'étage 2, et une communication entre l'étage 2 et l'étage 3. La première a comme tâche

émettrice FFT et comme tâche réceptrice ENERGIE. Le second lien de communication est ouvert entre la tâche ENERGIE (émission) et la tâche SUM (réception). Le tableau suivant donne les volumes de données à transmettre ainsi que les durées des transmissions. Ces volumes dépendent de la taille des blocs de calculs (partitionnement).

	Volume à transmettre	Débit de transmission	Durée
FFT→ENERGIE	102 400 octets	314 572 800 octets/s	375 521 ns
ENERGIE→SUM	102 400 octets	104 857 600 octets/s	1 026 563 ns

5.3 Latence

La distance maximale de dépendance est de **3** événements (longueur du chemin critique du graphe). Une période de synchronisation dure **1 événement**, et le nombre de périodes sur la durée du chemin critique est de **3**.

La durée en temps réel d'une période de synchronisation est de **3 615 521 ns**. De ce fait, la latence applicative est de **10 846 563 ns**.

Le tableau suivant récapitule les durées tâche par tâche.

	FFT	ENERGIE	SUM
Nombre de blocs de calcul sur une période	1	1	1
Durée d'un bloc de calcul	3 240 000 ns	1 536 000 ns	2 048 000 ns
Durée d'une communication	375 521 ns	1 026 563 ns	0 ns
Durée (comm/calcul)	3 240 000 ns	1 536 000 ns	2 048 000 ns
Coût sur une période	3 615 521 ns	2 562 563 ns	2 048 000 ns

5.4 Mémoire

	FFT	ENERGIE	SUM
Lecture	25 600 octets	51 200 octets	204 800 octets
Volume interne	0 octets	0 octets	0 octets
Volume de communication	0 octets	51 200 octets	204 800 octets
Volume externe	25 600 octets	0 octets	0 octets
Écriture	25 600 octets	51 200 octets	2 048 octets
Volume interne	0 octets	0 octets	0 octets
Volume de communication	25 600 octets	51 200 octets	0 octets
Volume externe	0 octets	0 octets	2 048 octets
Total	51 200 octets	102 400 octets	206 848 octets

Allocation mémoire par processeur et par étage utilisé :
- Volume mémoire utilisé par l'étage 1 : 51 200 octets
- Volume mémoire utilisé par l'étage 2 : 102 400 octets
- Volume mémoire utilisé par l'étage 3 : 206 848 octets

6 Une tâche sur le premier étage, deux sur le deuxième

Les tâches sont réparties dans les étages du pipeline ainsi :

	FFT	ENERGIE	SUM
Étage	1	2	2

Nombre de processeurs utilisés par étage :
- Étage 1 : 5 PE (sur 8 disponibles)
- Étage 2 : 2 PE (sur 4 disponibles)
- Étage 3 : 0 PE (sur 2 disponibles)

L'ordre des boucles dans chacun des nids reste inchangé (les matrices de permutations sont l'identité).

6.1 Partitionnement - Ordonnancement

On décrit ici, tâche par tâche, tous les paramètres associés au partitionnement et à l'ordonnancement.

	FFT	ENERGIE	SUM
Liste des itérateurs de boucles	(t, v, cd)	(t, v, cd, rec)	(t, v, rec)
Le vecteur d'itération initial	$(\infty, 4, 100)$	$(\infty, 4, 100, 256)$	$(\infty, 4, 256)$
La diagonale de la matrice P	(1, 1, 5)	(1, 1, 1, 2)	(1, 1, 2)
La diagonale de la matrice L	(1, 4, 10)	(1, 4, 100, 128)	(1, 4, 128)
Le vecteur c_{max}	$(\infty, 1, 2)^t$	$(\infty, 1, 1, 1)^t$	$(\infty, 1, 1)^t$
Le vecteur α	$(2, 2, 1)^t$	$(2, 2, 2, 2)^t$	$(2, 2, 2)^t$
Le temps de start-up	$\beta = 0$	$\beta = 2$	$\beta = 3$

6.2 Détection des communications

Deux étages sur les trois sont actifs, par conséquent il existe une communication entre l'étage 1 et l'étage 2. La tâche émettrice est FFT et la tâche réceptrice est ENERGIE. Le tableau suivant donne les volumes de données à transmettre ainsi que les durées des transmissions. Ces volumes dépendent de la taille des blocs de calculs (partitionnement).

	Volume à transmettre	Débit de transmission	Durée
FFT→ENERGIE	51 200 octets	314 572 800 octets/s	212 761 ns

6.3 Latence

La distance maximale de dépendance est de **4** événements (longueur du chemin critique du graphe). Une période de synchronisation dure **2 événements**, et le nombre de périodes sur la durée du chemin critique est de **2**.

La durée en temps réel d'une période de synchronisation est de **5 396 761 ns**. De ce fait, la latence applicative est de **10 793 522 ns**.

Le tableau suivant récapitule les durées tâche par tâche.

	FFT	ENERGIE	SUM
Nombre de blocs de calcul sur une période	2	1	1
Durée d'un bloc de calcul	2 592 000 ns	3 072 000 ns	1 024 000 ns
Durée d'une communication	212 761 ns	0 ns	0 ns
Durée (comm/calcul)	2 592 000 ns	3 072 000 ns	1 024 000 ns
Coût sur une période	5 396 761 ns	3 072 000 ns	1 024 000 ns

6.4 Mémoire

	FFT	ENERGIE	SUM
Lecture	40 960 octets	102 400 octets	51 200 octets
Volume interne	0 octets	0 octets	51 200 octets
Volume de communication	0 octets	102 400 octets	0 octets
Volume externe	40 960 octets	0 octets	0 octets
Écriture	20 480 octets	51 200 octets	1 024 octets
Volume interne	0 octets	51 200 octets	0 octets
Volume de communication	20 480 octets	0 octets	0 octets
Volume externe	0 octets	0 octets	1 024 octets
Total	61 4400 octets	153 600 octets	52 224 octets

Allocation mémoire par processeur et par étage utilisé :
- Volume mémoire utilisé par l'étage 1 : 61 440 octets
- Volume mémoire utilisé par l'étage 2 : 205 825 octets

7 Conclusion

Les solutions obtenues sont dans l'ensemble similaires. Les différences les plus marquantes touches les variables de paramètres opérationnels (latence et volume mémoire). Les solutions détaillées dans ce document sont représentatives des informations qu'elles portent. On a pu voir que tous les processeurs disponibles ne sont pas systématiquement utilisés.

La première des solutions présentées illustre ce que les experts savent déjà, c'est-à-dire qu'une parallélisation sur la dimension temporelle (la boucle la plus externe de chaque nid) n'est pas une bonne solution. La consommation mémoire et la latence sont démesurées (cette expérience a d'ailleurs été réalisée en ne limitant ni l'espace mémoire de chaque processeur, ni la latence maximale). Cette démesure est d'autant plus importante si les tâches constituant l'application ne sont pas sur le même étage du pipeline. En effet, il faut alors un minimum de périodes correspondant au nombre d'étages pour réaliser l'application, et multiplier par le nombre d'étages tous les espaces mémoire.

Les autres solutions permettent de vérifier l'intérêt du M-SPMD par rapport au SPMD. Une solution sur un seul étage est plus rapide est consomme moins de mémoire (par processeur) qu'une solution pipelinée, mais a un débit moindre. C'est-à-dire que pour produire en sortie le même nombre de données, il faudra plus de temps (à nombre de processeurs utilisés égal) en SPMD qu'en M-SPMD. Pour schématiser, si une optimisation en latence d'un mode SPMD met 8,5 ms pour calculer en sortie 1 Ko de données, un M-SPMD mettra 10,8 ms pour en calculer autant, mais avec 2 fois moins de ressources de calcul.

Ces expériences nous ont néanmoins permis de constater que notre modèle d'entrée/sortie n'était pas complet. Il a besoin d'être raffiné au niveau des relations entre les temps logiques (événements de calculs) et les temps physiques (où doivent apparaître des temps d'attentes si les calculs sont plus rapides que les récurrences d'entrées). C'est pourquoi, les durées sur une période de synchronisation ne sont pas proportionnelles au temps de récurrence d'entrée. Cette amélioration pourra faire l'objet de travaux futurs.

Chapitre 8

Conclusion

1 Contributions

L'ambition des travaux exposés ici est de repousser les limites applicatives et architecturales posées par la thèse de Christophe Guettier. C'est pourquoi nous nous sommes appuyés sur des applications radar plus complexes et plus importantes (chap. 3 sec. 1) que celles utilisées alors (entre 25 à 30 tâches aujourd'hui au lieu des 7 à 8 constituant les applications sonar de l'époque). Ainsi nous avons pu mettre à jour les points à étendre dans les modèles de [32] afin de prendre en compte quelques unes des particularités de ces applications (chap. 3 sec. 2).

Nous avons introduit la modélisation d'une machine M-SPMD (chap. 4 sec. 2). Cette modélisation nous a amenés à nous poser la question de la détection des communications entre processeurs à partir du partitionnement (chap. 5 sec. 4). Une de nos hypothèses étant l'absence d'échange de données entre processeurs appartenant à un même étage de pipeline, il nous est nécessaire de s'assurer que les partitionnements proposés en solution la respectent. De manière générale, cette problématique est indépendante de la machine, qu'elle soit SPMD, SIMD, M-SPMD ou autre. Elle n'est liée qu'au partitionnement qui décide de l'allocation spatiale et temporelle (localement) des données.

L'intérêt porté sur l'optimisation du débit plutôt que de la latence, nous a naturellement conduit à modéliser les entrées/sorties (chap. 5 sec.3). En effet, ces dernières étant généralement cadencées à une fréquence imposée par un contexte extérieur, cela pose des contraintes de débit à respecter.

Avant cela, nous avons également cherché à améliorer certains modèles établis dans [32] (chap. 5, sec. 1 et sec. 2) afin de réduire le nombre des approximations effectuées. Le volume élémentaire de mémoire utilisé par un bloc de calcul est plus fin, mais en contre partie plus complexe en terme de contraintes. Ce qui a pour effet de donner une solution plus proche de la réalité, mais obtenue moins rapidement (du fait de l'aspect non linéaire des contraintes associées à ce modèle). Ce retard peut être compensé par l'utilisation d'heuristiques. Il en est de même pour la modélisation approchée des dépendances entre blocs de calcul. Nos résultats sont de meilleure qualité mais le problème lié aux approximations introduites (par les différentes projections effectuées, la considération de la couverture entière du polytope

de dépendances entre autres) reste présent.

Du point de vue de l'outil Apotres, nous avons pu exprimer le pgcd et le ppcm de deux variables de domaine. Ces deux opérateurs apparaissent respectivement dans l'affinement du modèles des dépendances, dans le calcul de la période événementielle α de synchronisation globale d'une machine M-SPMD. Le comportement de ces opérateurs est loin d'être monotone, ce qui rend difficile la moindre déduction d'une information sur les domaines des variables. Cependant, le fait d'avoir pu les exprimer à l'aide de contraintes, permet tout de même de profiter un minimum des variations des domaines et ne pas attendre que les variables soient instanciées pour vérifier la cohérence du système.

Un autre point concernant Apotres est que nous sommes passés d'un outil de *placement automatique* à un outil d'*aide au placement*. La nuance est importante pour l'utilisateur qui reste maître des décisions de l'outil. Cela suppose un échange interactif entre l'utilisateur et Apotres. Cela se traduit par l'utilisation d'algorithmes de recherche incomplets permettant une recherche d'optimum incrémentale jusqu'à satisfaction de l'utilisateur.

Bien que nous ayons pu implémenter toutes les contraintes présentées dans ce document, il serait intéressant de passer des modèles formels mathématiques à des modèles utilisant davantage le monde des contraintes. Par exemple, il faudrait adapter la contrainte globale dite *cumulative*[2] à notre modèle mémoire. Le travail n'est pas si simple, car les tâches utilisant les ressources mémoires n'ont sont pas de durée de vie connues, puisqu'il s'agit des cycles de calcul qui dépendent du partitionnement des nids de boucles et de leur ordonnancement.

Par ailleurs, il est possible qu'une étude un peu plus poussée sur l'interaction des différents modèles déjà établis permette de mettre à jour des contraintes redondantes, améliorant les performances globales de la recherche de solution. L'introduction de nouvelles heuristiques (du domaine du placement ou plus générales) ferait de même.

Enfin le point le plus crucial est la génération de code à partir des directives de placements que nous délivrons. Nous trouvons des solutions de partitionnement et d'ordonnancement respectant les contraintes opérationnelles, mais sont-elles facilement implémentables (automatiquement ou non) ? Du fait que notre modélisation de l'occupation mémoire est capacitive, les fonctions d'allocations sont-elles simples à définir ? La section suivante ouvre une voie en ne s'intéressant qu'à la génération de code de contrôle.

2 Ouverture sur la génération de code

La solution de placement que nous donnons est exprimée sous formes de matrices et de vecteurs. Nous ne donnons que des directives de placement ainsi que les fonctions d'ordonnancement des calculs. Nous voulons ouvrir une voie pour la génération de code de contrôle à partir de ces directives et des ordonnancements.

Nous nous appuyons principalement sur les travaux de Jean-François Collard [17] [18] et de Denis Barthou et François Irigoin [9] [39]. Jean-François Collard obtient la génération d'un code de longueur constante, quel que soit l'ordonnancement des blocs de calculs. Cependant il introduit un surcoût des temps de calcul, car le contrôle dynamique qu'il obtient doit calculer la prochaine date et chercher à quel calcul cette date est associée.

Denis Barthou et François Irigoin ont cherché à obtenir une expression régulière à partir des fonctions d'ordonnancement, pour une période de calcul. Ainsi, l'ordonnancement des calculs est implicite au code obtenu. L'inconvénient est que, de manière générale, ce code n'a pas une taille raisonnable. La taille du code est proportionnelle à la longueur de l'expression régulière obtenue. Une solution serait de contraindre les ordonnancements pour prendre en compte ce phénomène, mais cela pourrait éliminer de bonnes solutions de placement.

En fait, nous pouvons constater que les inconvénients de l'un sont les avantages de l'autre et réciproquement. Mixer ces deux approches nous permettrait de générer un code de taille constante (quels que soient les ordonnancements), sans avoir à calculer les dates d'exécutions des différents blocs de calcul.

Voyons cela sur un exemple. Soit une application composée de trois tâches *carre, diff, integ* :

Tâche Carré : C

```
DO I=0,INFINITE
  DO J=0,7
   TAB12[I,J] = TAB1[I,J] * TAB1[I,J]
  ENDDO
ENDDO
```

Tâche Diff : D

```
DO I=0,INFINITE
  DO J=0,3
    TAB23[I,J] = TAB12[2*I,J] -
                 TAB12[2*I+1,J]}
  ENDDO
ENDDO
```

Tâche Integ : I

```
DO I=0,INFINITE
  S=0
  DO J=0,3
    S = S + TAB23[I,J]
  ENDDO
  TAB3[I] = S
ENDDO
```

Supposons que les directives de placement soient :

$$P_C = \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix} \qquad P_D = \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix} \qquad P_I = \begin{pmatrix} 1 \end{pmatrix}$$

$$L_C = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad L_D = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad L_I = \begin{pmatrix} 1 \end{pmatrix}$$

$$up(c_C) = \begin{pmatrix} +\infty \\ 2 \end{pmatrix} \quad up(c_D) = \begin{pmatrix} +\infty \\ 1 \end{pmatrix} \quad up(c_I) = \begin{pmatrix} +\infty \end{pmatrix}$$

que les fonctions d'ordonnancement soient :

$$\begin{array}{lcllcl} \alpha_C & = & \left(\ 3,\ 3\ \right), & \beta_C & = & 0 \\ \alpha_D & = & \left(\ 6,\ 3\ \right), & \beta_D & = & 4 \\ \alpha_I & = & \left(\ 6\ \right), & \beta_I & = & 5 \end{array}$$

Si un bloc de calcul de chacune des tâches *carré*, *diff* et *integ* est représenté respectivement par les symboles **C, D, I**, l'expression régulière associée à ces ordonnancements est $(C^2DI)^*$. Ce qui donne le code de contrôle suivant :

```
while (true) {
  for i = 0 to 1
    C;
  D;
  I;
}
```

Avec un contrôle dynamique, si l'on désigne par `c{C, D, I}` les vecteurs de temps logique d'exécution d'un bloc de calcul des tâches C, D et I ; par `succ(.)` la fonction qui calcule le successeur lexicographique d'un vecteur ; que l'on nomme `d{C, D, I}`(.) les fonctions d'ordonnancement et d la date logique courante, on obtient le code suivant :

```
while (true) {
 if (d == dateC) {
   C;
   dateC = dC(succ(cC));
 }
 else
   if (d == dateD) {
     D;
     dateD = dD(succ(cD));
   }
   else
     if (d == dateI) {
       I;
       dateI = dI(succ(cI));
     }
  d++;
}
```

Quel que soit l'ordonnancement trouvé, la structure de ce code ne change pas. Du même coup, le surcoût engendré par les contrôles (pour une itération du `while`) est constant et est évaluable indépendamment de l'ordonnancement choisi. Ce n'est pas le cas avec les expressions régulières qui sont, elles, entièrement dépendantes de l'ordonnancement. Il faut d'abord connaître l'ordonnancement pour déterminer l'expression régulière engendrant le code de contrôle.

L'intérêt des expressions régulières est de ne pas avoir à calculer les dates d'exécution. Celui du contrôle dynamique est de produire un code constant quel que soit l'ordonnancement trouvé. La combinaison des deux méthodes consiste à utiliser les expressions régulières pour déterminer la succession des blocs de calculs des tâches les uns par rapport aux autres,

appliqué au code produit par la deuxième méthode. Ainsi on regroupe les avantages des deux méthodes.

Du même coup, la complexité concernant la recherche d'une bonne expression régulière est diminuée puisqu'il nous en suffit d'une, et pas nécessairement une optimale. La trace sur une période pourrait être suffisante. En fait, nous avons simplement besoin de connaître l'ordre dans lequel les blocs de calculs se succèdent sur cette période.

La génération de code pour notre exemple pourrait donner :

```
#define length_regExp 4
#define dC 0
#define dD 1
#define dI 2
byte who[length_regExp] = {dC, dC, dD, dI};
int  next = 0;

while (true) {
  d = who[next];
  if (d == dC)
    C;
  else
    if (d == dD)
      D;
    else
      if (d == dI)
        I;
next = (next == length_regExp - 1)?0:next +1;
}
```

Malgré tout, nous n'avons pas là la solution miracle car il ne s'agit que du code de contrôle. Il faut également y introduire les communications, les barrières de synchronisation globales et l'allocation des tableaux de données. De plus, le stockage de l'expression régulière pose un problème de surcoût de la mémoire programme, d'autant plus s'il s'agit de la trace d'exécution.

Dans le cadre d'une génération de code pour une machine M-SPMD, le positionnement des communications dans le code est facilité par le fait que nous connaissons les tâches émettrices et les moments d'initialisations de ces communications. Les barrières de synchronisations sont posées à chaque fin de période, c'est-à-dire lorsque l'on revient en début de trace périodique.

Pour finir, nous dirons qu'il peut être intéressant d'un point de vue outil de connaître (et maîtriser) la taille du code produit. Ainsi, l'utilisateur sait que le code généré n'excédera pas le volume mémoire disponible, quelle que soit la solution de placement proposée. Si le surcoût induit (en mémoire et en latence) reste négligeable, alors la solution obtenue est «parfaite».

La solution de poser des contraintes sur les fonctions d'ordonnancement pour faciliter la génération de code est envisageable si l'on accepte de perdre de potentielles bonnes solutions de placement. Par exemple, nous pourrions obliger les coefficients des vecteurs α de tâches dépendantes à être multiples les uns des autres. Ainsi, les boucles générées à partir des ordonnancements s'emboîtent les unes dans les autres. Il est possible que la solution obtenue

ne soit pas optimale pour le critère de volume mémoire.

Il s'agit, pour le problème de la génération de code, de trouver un compromis entre éloignement de l'optimum et facilité d'implémentation de la solution proposée.

Annexe 𝔄

How Does Constraint Technology Meet Industrial Constraints ?

Workshop ESA, *On-Board Autonomy*, octobre 2001, Hollande.

How does constraint technology meet industrial constraints ?

Philippe Gérard Simon de Givry Jean Jourdan
Juliette Mattioli Nicolas Museux
Pierre Savéant
PLATON[1] Group
THALES Research & Technology France
Domaine de Corbeville
91404 Orsay cedex FRANCE
{philippe.gerard, simon.degivry, jean.jourdan}@thalesgroup.com
{juliette.mattioli,nicolas.museux,pierre.saveant}@thalesgroup.com

August 31, 2001

Abstract

This paper describes our experience for introducing Constraint Programming in operational systems of THALES[a]. Many of them are on-board and real-time systems. We give the requirements of a technology for developing real-time systems with combinatorial optimization capabilities. We present the constraint technology that has the ability to fulfill most of the requirements. The current limitations of state of the art Constraint Programming solvers yield to research works that are described. Finally, we give an overview of the constraint technology at THALES.

[a]Ex Thomson-CSF.

Introduction

Industrial motivations

In many areas of industry and business, the optimization of resource allocation is becoming crucial. Services and products must fulfill the increasingly specific needs of customers. What is the best way to allocate costly resources and improve competitiveness ? Re-directing resources to handle shortages or rearranging the sequence of activities can lower costs, reduce waste, shorten cycle time and speed up delivery time. Planning and scheduling systems will provide a competitive advantage on the market.

Planning determines the sequence of tasks (the plan) and scheduling decides on the choice of specific resources, operations and their timing to perform the tasks (the schedule). Planning and scheduling problems are highly complex and involve making many decisions, including which set of resources are to be assigned to each demand. Each decision has a limited number of possible alternatives and must satisfy operational constraints. These problems belong to a class of problems called combinatorial optimization problems. Most of them are computationally difficult and require considerable development time and expertise, in both the application domain (for modeling) and algorithm design (for solving).

[1]PLAnning opTimization & decisiON making

As on-board real-time systems evolve from hardware to software solutions, their resource management functions require more sophisticated algorithms dealing with complex planning and scheduling problems. Such capabilities are more and more required in applications areas such as C3I , sensor management, weapon allocation or deployment, telecom resource management, mission planning ...

The following section gives the requirements of a technology for developing real-time systems with combinatorial optimization capabilities.

Requirements of a suited technology

Real-line systems have a difficult operational context compared to classical off-line systems. They must react continuously to follow changes of an external environnement. The response time can be very short. A valid plan/schedule has to be provided at each deadline. And there is also a memory space limit.

Moreover, in several applications, the life cycle of the system can be very long. The system has to be maintained during a long period, over twenty years in the case of a Defense system. During this period, there will be many retrofitting of the system. The system has to be robust to the addition of new functionalities and should derive benefit from platform evolutions (faster computers).

The construction and the maintenance of complex large scale software systems, including heterogeneous, multi-components, multi-functions applications become significant in the software marketplace demand. One way to solve a combinatorial problem is to develop a specific algorithm completely from scratch. This has the advantage of exploiting all the problem-specific knowledge. The disadvantage is that it can take a long time to develop a really satisfactory solution. System maintenance is dependent on the software author's particular way of thinking. Moreover, if the problem changes in subtle ways, the algorithm must be scrapped and reinvented, that can be very costly.

The target technology must have the following properties :
- Time and space guarantees,
- Robustness to retrofitting of the system,
- Modularity property through a safe incremental programming approach (useful for complex and large systems involving several engineers and for heterogeneous systems),
- Reuse property through a high level of abstraction because reading code is a skill that not even software experts do very well. It is difficult to fully understand something which is thousand lines of code, without making a significant investment. This property allows :
  - an easy reasoning on the fundamental properties of the system,
  - the reduction of the gap between specification and code,
  - and the opportunity to capitalize at the model level,
- Traceability property through a formal approach enabling the refinement of requirements.

Note that some industrial companies, for confidentiality or market protection reasons, may prefer to keep in house the entire development process and its know-how. The technology has to be mastered in house.

In the following part, the constraint technology is presented as a key technology for modeling and solving real world combinatorial problems.

1 Constraint Programming for modeling and solving real world combinatorial problems

Traditional modeling languages are particularly strong in mathematical programming application (e.g. linear or integer programming). Some of the combinatorial problems are naturally expressed using algebraic or logic notations. However, a number of combinatorial applications, such as job-shop scheduling and resources allocation problems are out-side the scope of these languages. On the one hand, these problems are rarely expressed naturally using only algebraic constraints. On

the other hand, it is often important in these applications to guide the solver toward solutions by specifying an appropriate search procedure or a domain heuristic.

In the following sections, the Constraint Programming is introduced. A clean separation between the modeling and the solving parts is shown. The underlying mechanisms are briefly presented. Finally, an industrial analysis of the fundamental properties of the constraint technology is given.

A longer introduction of Constraint Programming can be found in [Wallace, 1995; Barták, 1998]. Reference books are [Van Hentenryck, 1989; Tsang, 1993; Marriott & Stuckey, 1998].

1.1 A short history of Constraint Programming

Constraint Programming has a long tradition in Artificial Intelligence. The Constraint Satisfaction Problem (CSP) was first formalized and studied in vision research for solving line-labeling problems during the early seventies. This class of problems is important because any combinatorial optimization problem can be represented in this paradigm.

But Constraint Programming as a language comes from the integration of consistency techniques in Logic Programming in the mid-eighties. Logic Programming brought theoretical foundations and the language point of view thanks to its built-in nondeterminism. Consistency techniques and Operations Research provided efficient implementation methods mainly based on graph theory. Constraint Logic Programming (CLP) emerged in the mid 80's, from European and American labs. Most influential was the work of the CLP(R) team at IBM, who coined the word CLP and set its theoretical foundations. Prolog II was then the only instance of promising scheme. A definite interest in finite domains arose from the work on the CHIP concept (Constraints Handling in Prolog) at the joint Bull-ICL-Siemens European Computer-Industry Research Center. Both CHIP and Prolog III later turned into products.

The first startups appeared in early 90's : Cosytec, Axia, Ingenia and Ilog. Several academic solvers were also developed among them SICStus in Suede, Eclipse at the Imperial College Park (UK), Oz, CHR and IF/Prolog in Germany, B-Prolog in Japan, clp(FD) and ncl at INRIA (France) and many more. Most of corporate research laboratories of industrial companies have developed also their own solvers, for instance THALES (ex. Thomson-CSF), France-Telecom, Daimler, etc

Cosytec is the godfather of the so-called global constraints which allow to easily specify and solve complex problems in areas like planning, scheduling, sequencing, packing, configuration and routing. Note that the Constraint solving paradigm came out from Logic Programming to integrate general purpose languages such as Ilog SOLVER, the world leader, which is not Prolog-based.

The industrial impact started in 1993 with the first operational applications. Today the technology has spread out many markets such as manufacturing, transportation, telecommunications and building.

1.2 Constraint Programming as a modeling language

"Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming : the user states the problem, the computer solves it." [E. Freuder, Constraints, April 1997]

The essence of Constraint Programming is based on a clean separation between the statement of the problem (the variables and the constraints), and the resolution of the problem (the algorithms).

The basic way to express and model a combinatorial problem is with :

- Variables, which are defined with a specified domain and should allow to model the possible decisions one can take for determining a solution to the problem.
- Constraints, describing all the restrictions on variables and all relations between these variables that must be satisfied for the given problem. The important feature of constraints is their declarative manner, i.e., they specify what relationship must hold without specifying a computational procedure to enforce that relationship.

– Objective function(s), specifying what must be optimized. An objective function is usually expressed as a function of (a part of) the decision variables or related to the variables in a more implicit or complex manner.

In order to partially overcome the lack of expression and deductive power of conventional Constraint Programming languages, global constraints have been introduced. One of the main advantages of global constraints is to take into account more globally a set of elementary constraints [Beldiceanu & Contejean, 1994]. It can lead to substantial speedups because they allow the use of propagation algorithms based on mathematical properties of constraints using the notions from graph theory or mathematical programming.

Most of the constraint realizations we have worked on are built from two types of models :

– Extension models. This is the usual model that most people build using constraint technology. For this type of model we can represent in extension all the elements involved in the model, for instance the set of tasks and resources of the scheduling problem.
– Comprehension models. We develop such models when it is impossible to represent all the elements of a model. This occurs in three cases for continue phenomena, for dynamic aspects and sometimes for scaling up issues. The amount of effort and the methodology required to build comprehension models is not comparable to those to build extension models.

The difficulty in building comprehension model is to preserve the interactions with other models. Otherwise, it is not worth using Constraint Programming. The way we set up comprehension models are based on an approximation process, which consists in finding out a set of sufficient conditions. In most of the case studies, this modeling approach has preserved the use of constraint technology. Sometimes the constructed sufficient conditions have led to too rough approximations. In such cases the solution lies in the combination of several resolution techniques, in particular, non linear solving methods [Hentenryck *et al.*, 1997]. Unfortunately sometimes, it is still an open issue and even worth impossible to achieve.

1.3 How does Constraint Programming work ?

Computation behind Constraint Programming is based on the cooperation of two processes : local consistency maintenance on one hand and problem decomposition on the other one.

Local consistency is achieved by filtering algorithms which remove combinations of values that cannot appear in the solution under construction. This process is also called constraint propagation. If all bad combinations were discarded then solutions would be reached ; but this problem is NP-hard and polynomial algorithms are available only for 2 or 3 variables, that is why consistency is not complete. With Finite Domain Constraint Programming variables range over finite domains which are integers or intervals over integers and each constraint defines its own filtering algorithm.

Decomposition is based on committing choices to divide the problem into sub-problems until they get simple enough to be solved in a straightforward way. It is usually implemented by a tree-search algorithm. For optimization problems the algorithm is extended to a so-called branch-and-bound algorithm which principle is to avoid to search in sub-trees for which a proof has been made that no better solution can appear. This way of enumerating could be sufficient but its complexity is obviously exponential. That is why in practice solutions are reached by the interleaving of the two processes : the propagation engine is requested each time a choice is committed.

1.4 Software engineering properties provided by the constraint technology

Constraint programming appears to be a very appealing technology for most of the combinatorial optimization problems. The interest of industrials in using this technology is mainly based on its software engineering capabilities.

Indeed it is well known that the constraint technology can reduce the development time of the planning/optimization applications by orders of magnitude and provides several others advantages from industrial prospects, which have not been too much highlighted and discussed so far. The

analysis below does not pretend to be exhaustive but is the first attempt to summarize what we have learned from an industrial viewpoint during 10 years trying to introduce the constraint technology in operational systems of THALES :

- Thanks to its declarative nature this technology appears more to be a modeling technology than a programming technology. Indeed, application engineers don't have to care about the order the constraints are set in the system and furthermore they don't have to worry about the interactions with later added constraints. This seems trivial but provides a safe incremental programming approach, which becomes fundamental for designing complex and large systems involving several engineers.
- Thanks to its compositionality property, this technology provides a nice way of representing multiple redundant models or complementary models. The generic modeling framework, presented in [Jourdan, 1995a; 1995b] introduces the notion of redundant dual models. This modeling approach goes a step forward the notion of redundant constraints and leads to important improvements in term of scalability [Cheng *et al.*, 1999].
- Thanks to its high level of abstraction, it helps in reducing the gap between the specifications and the realizations. The debugging process is made easier, errors are conceptual errors instead of programming errors. A constraint program consists in a system that reasons on the fundamental properties of the system, which are not far from its specifications. For industrials in charge of huge programs, requirement traceability is very important and even contractually required. Furthermore, the high level of abstraction provided by Constraint Programming offers the opportunity to capitalize at the modeling level rather than at the programming level, which yields to higher gain in productivity.
- Thanks to its formal approach, it helps in refining the specifications themselves. The mathematical essence of Constraint Programming imposes to transform specifications into equations (in a broad sense). Consequently a lot of imprecise or fuzzy requirements are removed.

However the constraint technology is faced with difficulties which are explained below. Research directions are given to overcome the limitations. A French research project is presented. It aims at producing a real-time Constraint Programming framework.

2 Toward a real-time Constraint Programming framework

Current limitations of state of the art Constraint Programming generic solvers are :

- Due to the complexity of constraint reasoning, it can be difficult to debug a constraint or a model. There is a need for an automatic explanation of the algorithm working.
- There is a lack of abstraction for expressing search algorithms. In Constraint Programming, one has to program its search algorithm rather than specify it.
- Of course, combinatorial problems remain difficult to solve. In limited time, the results of a search may have a high variability in term of quality. The usual search algorithm in Constraint Programming solver, based on a complete search tree, does not fit to a limited time context. Moreover if the search algorithm is stopped before its normal end, there is no information on the quality of the results. This is a problem if we want to validate the results of a real-time optimization system.
- There is no guarantee on space and time.
- The solver does not take into account the time contract. Therefore it will not take advantage of the evolutions of the platforms.

A French project, called EOLE[2] [EOLE web, 2000], aims to overcome these limitations. This project is a RNRT[3] project supported by the French Research Office. It began on April 2000, and it will be finished at the end of 2002. The work plan is divided into six sub-projects. The first one consists

[2]Environnement d'Optimisation en-LignE dédié Télécom / On-Line Optimization Framework dedicated for Telecom domain

[3]Réseau National de Recherche en Télécom

in establishing foundations, concepts and perimeter of use of an optimization framework for on-line combinatorial problems. This sub-project should supply with formalisms to express algorithms the behavior of which answers the defined characteristics. The second sub-project consists in offering primitives implementing the previous formalism. Finally, it aims at realizing the integration of these tools and libraries in an object-oriented framework. The following three sub-projects allow the validation through a set of benchmarks led on three experimental applications (ATM network management, network reconfiguration and frequency assignment). And the last sub-project attempts to define an industrialization plan of the developed technology.

The proposed approach consists in combining :
- the flexibility of modeling of Constraint Programming,
- the efficiency of hybrid algorithms,
- the adequation to real-time operational constraints of anytime algorithms[4] and parameterized search algorithms,
- and the capitalization capabilities of optimization frameworks, based on high level search primitives, templates of search and code generator.

The following sections give an insight of these approaches.

2.1 Scalability to large scale combinatorial problems

They are two orthogonal approaches to face large scale combinatorial problems.

The first approach concerns the modeling phase. A refinement process of the model will enhance the constraint propagation efficiency. The goal is to find the best tradeoff between the propagation computation time and the resulting pruning of the search space. The addition of redundant models will also improve the stability and quality of the results [Jourdan, 1995a; Cheng *et al.*, 1999].

The second approach concerns the solving methods. There are two kinds of methods : tree search methods and local search methods. Tree search methods use a problem decomposition scheme for producing solutions [Kumar, 1992]. Without any time limit, these methods are complete. In a limited time context, the notion of completeness is given up, and the notion of non systematic search is introduced [Harvey & Ginsberg, 1995; Harvey, 1995]. The main idea is to follow the heuristics in a more cleaver way, by trying to diversify the search progressively.

Local search methods transform a (complete) assignment of the decision variables iteratively, by following a cost gradient and making a few random choices to escape from local minima [Aarts & Lenstra, 1997]. These methods are well adapted to a limited time context, but they do not benefit from the constraint technology.

Recent works are around the hybridization of tree search and local search methods. Hybrid search methods are divided into three categories :
- embedded hybridization combines the basic mechanisms of tree and local search methods during the search (e.g. [Mazure *et al.*, 1996]),
- hybridization by composition clearly separates tree search and local search methods during the search (e.g. [Wallace, 1996]),
- and methodological hybridizations where a tree search or a local search method is used to solve a distinct but complementary problem to help the search on the initial problem (e.g. [Kask & Dechter, 1996]).

An attempt to insert local search methods in Constraint Programming is a very important research direction, see works from [Pesant & Gendreau, 1999].

2.2 Taking into account time and space limits

In a few pathological cases, tree search methods and constraint propagation can use a very large amount of memory. The computing depth of these recursive algorithms has to be bounded in order

[4] An anytime algorithm must be able to supply at any time a useful solution, even if it is not optimal or if its optimality was not established

to have a guarantee on space. The time guarantee is obtained by an operating system alarm. And for very short computation time, partial solutions are returned, built in a deterministic way.

An important issue is to exploit the knowledge of a time contract for the search. This knowledge allows to control the search in order to produce better quality results.

In the case of a hybrid search algorithm by composition of several tree/local searches, the time contract will be divided among the different searches [Horvitz & Zilberstein, 2001]. Each search will have its own deadline. The way the time contract is divided depends on a given temporal policy. For a local search method, its complexity is easily controlled by its number of iterations. For a tree search method, its complexity can be controlled by parameterizing the method. For that, we introduce special parameters dealing with incompleteness of the search [Givry *et al.*, 1999]. The parameters are automatically tuned, by using time estimation tools [Lobjois & Lemaître, 1998].

2.3 High level primitives for designing search algorithms

Besides the modeling phase there is still a complex programming phase for designing an accurate and efficient search algorithm. The next challenge to be addressed by the constraint community will be to extend the modeling approach to the algorithmic phase. Recent works, like SaLSA [Laburthe, 1998] and search building blocks in OPL [Hentenryck, 1999; Perron, 1999], go in that direction.

Our interest is to go further in the concept of concurrent constraint model-based programming [Jourdan, 1995a] defining an infrastructure which will make the modeling task within the reach of a non constraint expert user. The reuse of models is a major step toward that ambitious goal. To enhance flexible reuse, the models can be broken down into reusable components, containing also specific search methods and specific domain heuristics.

3 Overview of the constraint technology at THALES

Several constraint languages have been designed at THALES. The first one was Meta(F) [Codognet *et al.*, 1992]. The idea behind this meta constraint propagator, built on top of Sicstus prolog, was to say that the performance of the propagator itself was not so important comparing to the gain obtained by pruning the search space. At the end, Meta(F) was great to prototype applications but not to develop operational applications. Later we have designed CMeta, it was the first constraint language implemented as a full C library. The library includes C routines for the constraints and C routines for the implementation of non determinism search algorithms. The performances of CMeta occurred to be very interesting but it appeared very difficult to extent the language with new constraints, almost impossible to enable user defined constraints, and not easy to write search algorithms. From these two interesting experiences, the main learned lesson is, that it is important to use an efficient host programming language and to not sacrifice the high level of abstraction required by a good constraint language.

The new technology we are currently devising at THALES Research and Technology is built on this last remark. Our current constraint technology, called Eclair(c) [Laburthe *et al.*, 1998; OpenEclair, 2000; Eclair1.4, 2001], is based on the host language Claire(c) [Caseau & Laburthe, 1996]. Claire(c) provides most of the basic mechanisms necessary to design a constraint language. Moreover, Claire(c) is a very efficient source to source compiler which enables to not be stick with a specific target programming language. This last point answers another problem we have to face, it is to deploy constraint realizations in various technical contexts from embedded systems to large scale distributed systems where the zoology of programming languages is heterogeneous (Fortran, Ada, C, Java, ...). Consequently, we would like to be able to derive our models, written in the high level constraint language, on several usual target programming language.

The Eclair(c) acronym stands for Expressing Constraints with a Library of Algorithms for Inference and Resolution. Eclair(c) is a finite domain constraint solver over integers written in the Claire(c) programming language. The Eclair(c) library has an open architecture, which allows the user to

add its own constraints in a straightforward manner. Actually, there is no difference between a user defined constraint and Eclair(c) constraints. Both are designed at the same level of abstraction. The current release of Eclair(c) [Eclair1.4, 2001] includes arithmetic constraints, global constraints and Boolean combinations. Eclair(c) provides a standard labeling procedure for solving problems and a branch-and-bound algorithm for combinatorial optimization problems. Programmers can also design easily their own non-deterministic procedures thanks to the dramatically efficient trailing mechanism available in Claire(c).

Eclair(c) has been designed for on-board real-time applications. Primitives are offered for the control of memory allocation and time bounding. Applications written in Eclair(c) have guarantees on time and space.

On top of Eclair(c) we are currently adding high level primitives for expressing hybrid search algorithms taking into account a time contract. A template mechanism is introduced which allows the reuse of anytime search methods.

Conclusion

Constraint Programming languages are among the very few new language proposals that are seeing industrial success while being based on a novel programming paradigm. Constraint Programming incorporates techniques from Mathematics, Artificial Intelligence and Operations Research, and it offers significant advantages in these areas since it supports fast program development, economic program maintenance, and efficient runtime performance, thanks to :

- a safe incremental programming approach,
- a high level abstraction which allows reasoning on the fundamental properties of the system, reducing the gap between specifications and code, and gives the opportunity to capitalize at the model level,
- a formal approach enabling the refinement of requirements,
- the combination of search and incremental constraint solving capabilities, and the relative efficiency of the resulting applications.

These characteristics are added to the strength in general-purpose symbolic processing of the underlying logic programming kernel on which they are often built. Constraints extend this kernel to numerical domains and beyond, offering a natural platform in which applications combining symbolic and numeric computing can be easily developed.

The following research directions will allow the current limitations of Constraint Programming to be overcome :

- For algorithm efficiency : hybridizations between tree search, local search and constraint propagation.
- For system adaptability : computational complexity, performance prediction and real-time monitoring of solution quality. Learning of search control strategies.
- For capitalization : high level primitives for designing search algorithms and domain frameworks.

Note that there is a workshop at the annual Constraint Programming conference (CP-2001, 1st December 2001) on on-line combinatorial problem solving and Constraint Programming

(see http ://www.lcr.thomson-csf.com/projects/www_eole/workshop/olcp.html).

References

[Aarts & Lenstra, 1997] E. Aarts, J. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.

[Barták, 1998] Roman Barták. On-line guide to constraint programming. http ://kti.ms.mff.cuni.cz/~bartak/constraints/, 1998.

[Beldiceanu & Contejean, 1994] N. Beldiceanu, E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12) :97–123, 1994.

[Caseau & Laburthe, 1996] Y. Caseau, F. Laburthe. Introduction to the claire programming language. Technical Report LIENS technical report 96-15, Ecole Normale Supérieure, 1996.

[Cheng *et al.*, 1999] B.M.W. Cheng, K.M.F. Choi, J.H.M. Lee, J.C.K. Wu. Increasing constraint propagation by redundant modeling : an experience report. *Constraints*, 4(2) :167–1992, 1999.

[Codognet *et al.*, 1992] P. Codognet, F. Fages, J. Jourdan, R. Lissajoux, T. Sola. On the design of meta(f) and its application in air traffic control. In *Proc. of JICSLP workshop on constraint logic programming*, Washington, DC, 1992.

[Eclair1.4, 2001] PLATON, THALES Research & Technology, Orsay, France. *Eclair reference manual*, v1.4 edition, 2001.

[EOLE web, 2000] EOLE project : On-line optimization framework for telecom. http ://www.lcr.thomson-csf.com/projects/www_eole (in french), 2000.

[Givry *et al.*, 1999] Simon de Givry, Pierre Savéant, Jean Jourdan. Optimisation combinatoire en temps limité : Depth first branch and bound adaptatif. In *Proc. of JFPLC-99*, pages 161–178, Lyon, France, 1999.

[Harvey & Ginsberg, 1995] William D. Harvey, Matthew L. Ginsberg. Limited discrepancy search. In *Proc. of IJCAI-95*, pages 607–613, Montréal, Canada, 1995.

[Harvey, 1995] William D. Harvey. *NONSYSTEMATIC BACKTRACKING SEARCH*. PhD thesis, Stanford University, March 1995.

[Hentenryck *et al.*, 1997] P. Van Hentenryck, L. Michel, Y. Deville. *Numerica : A Modeling Language for Global Optimization*. The MIT Press, Cambridge, Mass., 1997.

[Hentenryck, 1999] P. Van Hentenryck. *OPL : The Optimization Programming Language*. The MIT Press, Cambridge, Mass., 1999.

[Horvitz & Zilberstein, 2001] E. Horvitz, S. Zilberstein. Computational tradeoffs under bounded resources. *Artificial Intelligence*, 126(1-2), 2001.

[Jourdan, 1995a] Jean Jourdan. *Concurrence et Coopération de Modèles Multiples dans les Langages de Contraintes CLP et CC : Vers une Méthodologie de Programmation par Modélisation*. PhD thesis, Université Denis Diderot U.F.R. Informatique, 1995.

[Jourdan, 1995b] Jean Jourdan. Concurrent constraint multiple models in CLP and CC languages : toward a programming methodology by modelling. In *Proc. of INFORMS*, New Orleans, USA, October 1995.

[Kask & Dechter, 1996] K. Kask, R. Dechter. A graph-based method for improving gsat. In *Proc. of AAAI-96*, pages 350–355, Portland, OR, 1996.

[Kumar, 1992] V. Kumar. Algorithms for constraint satisfaction problems : A survey. *AI Magazine*, 13(1) :32–44, 1992.

[Laburthe *et al.*, 1998] François Laburthe, Pierre Savéant, Simon de Givry, Jean Jourdan. Eclair : A library of constraints over finite domains. Technical Report technical report ATS 98-2, Thomson-CSF LCR, Orsay, France, 1998.

[Laburthe, 1998] François Laburthe. SaLSA : a language for search algorithms. In *Proc. of CP-98*, pages 310–324, Pisa, Italy, October 26-30 1998.

[Lobjois & Lemaître, 1998] L. Lobjois, M. Lemaître. Branch and Bound Algorithm Selection by Performance Prediction. In *Proc. of AAAI-98*, pages 353–358, Madison, WI, 1998.

[Marriott & Stuckey, 1998] Kim Marriott, Peter J. Stuckey. *Programming with Constraints : An Introduction*. The MIT Press, 1998.

[Mazure *et al.*, 1996] B. Mazure, L. Saïs, E. Grégoire. Boosting complete techniques thanks to local search methods. *Symposium on Math&AI-96*, 1996.

[OpenEclair, 2000] Eclair solver, 2000. Open source version at http ://www.lcr.thomson-csf.com/project/openeclair.

[Perron, 1999] Laurent Perron. Search procedures and parallelism in constraint programming. In *Proc. of CP-99*, pages 346–360, Alexandria, Virginia, October 11-14 1999.

[Pesant & Gendreau, 1999] G. Pesant, M. Gendreau. A constraint programming framework for local search methods. *Journal of Heuristics*, 1999.

[Tsang, 1993] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[Van Hentenryck, 1989] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, 1989.

[Wallace, 1995] Mark Wallace. Constraint programming. Technical report, IC-Parc„ Imperial College, London, UK, September 1995.

[Wallace, 1996] R. Wallace. Enhancements of branch and bound methods for maximal constraint satisfaction problem. In *Proc. of AAAI-96*, pages 188–195, Portland, OR, 1996.

Table des figures

Liste des tableaux

Bibliographie

[1] A.Demeure and Y. Del Gallo. An Approach for Signal Processing Design. In *SAME*, System on Chip Session, 1998.

[2] A. Aggoun and N. Beldiceanu. Extending CHIP in Order to Solve Complex Scheduling Problems. *Journal of Mathematical and Computer Modelling*, 17(7) :57–73, 1993.

[3] C. Ancourt, D. Barthou, C. Guettier, F. Irigoin, B. Jeannet, J. Jourdan, and J. Mattioli. Automatic Data Mapping of Signal Processing Applications. *IEEE International Conference on Application Specific Systems, Architectures, and Processors*, pages 350–362, 1997.

[4] C. Ancourt and F. Irigoin. Entrées-Sorties. Rapport de recherche E/234/CRI, ENSMP/CRI, 2000. Contrat RNRT Prompt.

[5] D. Aulagnier. Benchmark - Radar aéroporté à forme d'onde MFR. Technical Report SBU.IS/DSTN/ELA99/NT/P1, Thomson-CSF / Detexis, 1999.

[6] M. Barreteau, J. Jourdan, J. Mattioli, C. Ancourt N. Museux, and F. Irigoin. Programmation Logique avec Contraintes appliquée au placement : une méthodologie, un outil. In INRIA, editor, *5e Workshop Adéquation Algorithme Architecture*, pages 204–212, janvier 2000.

[7] M. Barreteau, J. Mattioli, T. Granpierre, Y. Sorel C. Lavarenne and, P. Bonnot, P. Kajifasz, F. Irigoin, C. Ancourt, and B. Dion. PROMPT : A mapping environnment for telecom applications on *System-On-a-Chip*. In *Compilers, Architecture, and synthesis for embedded systems*, pages 41–48, november 2000.

[8] D. Barthou, C. Guettier, J. Mattioli, B. Jeannet, F. Irigoin, and J. Jourdan. Modèle d'ordonnancement événementiel pour le placement automatique d'applications traitement du signal sur machine spmd. Technical Report IAS-96-3, Thomson-CSF/LCR, 1996.

[9] D. Barthou and F. Irigoin. Méthodes de génération de code à partir d'un ordonnancement. Technical Report E/208/CRI, ENSMP/CRI, Septembre 1996.

[10] D. Beauquier, J. Berstel, and Ph. Chrétienne. *Élément d'algorithmique*. Masson, 1992.

[11] A. J. Bernstein. Analysis of Programs for Parallel Processing. *IEEE Transactions on El. Computers, EC-15*, 1966.

[12] C. Bessière and M.O. Cordier. Arc-Consistency and Arc-Consistency Again. In *Proc. of AAAI-93*, pages 108–113, Washington, DC, 1993.

[13] C. Bessière, E. C. Freuder, and J-C Régin. Using Inference to Reduce Arc Consistency Computation. In *Proc. of IJCAI-95*, volume 1, pages 592–598, Montréal, Canada, 1995.

[14] Y. Caseau and F. Laburthe. The CLAIRE Programming Language. Rapport technique 96-16, École Normale Supérieure, 4, rue d'Ulm, 75005 Paris, 1996.

[15] N. V. Chernikova. Algorithm for finding a general fomula for the non-negative of a system of inequalities. *USSR Comput. Math. Phys.*, 5 :228–233, 1965.

[16] F. Le Chevalier. Bases physiques du radar. *Revue technique Thomson-CSF*, 1993.

[17] J-F. Collard. Code generation in automatic parallelizers. In *International Conference on Applications in Parallel and Distributed Computing*, IFIP WG 10.3, pages 185–194, North-Holland, Avril 1994.

[18] J-F. Collard. *Parallélisation automatique des programmes à contrôle dynamique*. Informatique, Université P. et M. Curie, Paris VI, Janvier 1995.

[19] OpenMP Consortium. OpenMP Fortran Application Program Interface, Version 1.0, october 1997. www.openmp.org.

[20] A. Darte. *Techniques de parallélisation automatique de nids de boucles*. PhD thesis, LIP, 1993.

[21] A. Darte. On the complexity of loop fusion. *Parallel Coomputing*, 26(9) :1175–1193, August 2000.

[22] A. Darte, C. Diderich, M. Gengler, and F. Vivien. Scheduling the Computations of a Loop Nest with Respect to a Given Mapping. In *Eighth International Workshop on Compilers for Parallel Computers*, CPC2000, pages 135–150, january 2000.

[23] J-L Dekeyser, A. Demeure, P. Marquet, and J. Soula. Compilation Array-OL : Rapport intermédiaire LIFL-TMS. Technical report, Thomson-CSF / Marconi Sonar - LIFL, October 1998.

[24] Y. Deville and P.V. Hentenryck. An Efficient Arc Consistency Algorithm for a Class of CSP Problems. In *Proc. of IJCAI-91*, pages 325–330, Sydney, Australia, 1991.

[25] M. Dincbas, P.V. Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F.Berthier. The Constraint Logic Programming Language CHIP. In *The International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, 1988.

[26] P. Feautrier. Some efficient solutions to the affine scheduling problem, part i : one dimensional time. *International Journal of Parallel Programming*, 21(5) :389–420, octobre 1992.

[27] P. Feautrier. Toward Automatic Distribution. *Parallel Processing Letters*, 4(3) :233–244, 1994.

[28] R. E. Gomory. *Recent Advances in Math. Programming*, chapter An algorithm for integer solutions to linear programs, pages 269–302. editors R. L. Graves and P. Wolfe, Mac-Graw Hill, New-York, 1963. chap. 34.

[29] M. Grabisch and M. Roubens. Application of the Choquet integral in multicriteria decision making. In M. Grabisch, T. Murofushi, and M. Sugeno, editors, *Fuzzy Measures and Integrals — Theory and Applications*, pages 348–374. Physica Verlag, 2000.

[30] P. Gérard, S. de Givry, J. Jourdan, J. Mattioli, N. Museux, and P. Savéant. How does constraint technology meet industrial constraints ? In *Workshop ESA on On-Board Autonomy*, Technology for autonomous operation, decision process, planning and scheduling, discrete control, fdir, Octobre 2001.

[31] D. Gries and F. B Schneider. *A Logical Approach To Discret Math.* Texts and Monographs in Computer Sciences, springer-Verlag edition, 1995.

[32] C. Guettier. *Optimisation globale et placement d'applications de traitement du signal sur architectures parallèles utilisant la programmation par contraintes*. PhD thesis, École Nationale Supérieure des Mines de Paris, November 1997.

[33] C. Guettier, J. Jourdan, and C. Ancourt. Ressources configuration of parallel architecture for multi-dimensional digital processing. Technical Report IAS-96-1, Thomson-CSF/LCR, 1996. RCAAS - Action 1995 - (I).

[34] C. Guettier, J. Mattioli, and J. Jourdan. Modèle de partitionnement 3D. Technical Report IAS-96-2, Thomson-CSF/LCR, 1996. RCAAS - Action 1995 - (II).

[35] W. Harvey. Computing two-dimensional integer hulls. *Society for Industrial and Applied Mathematics*, 28(6) :2285–2299, 1999.

[36] W. D. Harvey and M. L. Ginsberg. Limited Discrepancy Search. In *Proc. of IJCAI-95*, pages 607–613, Montréal, Canada, 1995.

[37] ILOG SA, 9, rue de Verdun, Gentilly. *ILOG-SOLVER : Reference Manual and User Manual*, 1996.

[38] F. Irigoin. *Partitionnement de boucles imbriquées, une technique d'optimisation pour les programmes scientifiques*. PhD thesis, Université Pierre et Marie Curie, June 1987.

[39] F. Irigoin. Génération de boucles pour deux tâches ordonnancées par α/β. Technical Report E/203/CRI, ENSMP/CRI, Novembre 1996.

[40] F. Irigoin and R. Triolet. Supernode partitionning. In *Fiftennth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 319–329, San Diego, California, january 1988.

[41] J. Jaffar and J-L Lassez. Constraint Logic Programming. In *POPL*, Munich, Jan 1987.

[42] J. Jourdan. *Concurrence et coopération de modèles multiples dans les langages de contraintes CLP et CC : Vers une méthodologie de programmation par modélisation*. PhD thesis, Université D. Diderot - Paris VII, February 1995.

[43] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr. Zosel, and M. Zosel. *The Hight Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[44] U. Kremer. NP-Completeness of Dynamic Remapping. In *International Workshop on Compilers for Parallel Computers*, CPC1993, pages 135–141, Delft, Netherlands, december 1993.

[45] Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4) :406–471, December 1999.

[46] Y. Kwok, I. Ahmad, M-Y. Wu, and W. Shu. A Graphical Tool for Automatic Parallelization and Scheduling of Programs on Multiprocessors. *International Conference on Parallel Processing 26*, pages 294–301, 1997.

[47] F. Laburthe, P. Savéant, S. de Givry, and J. Jourdan. ECLAIR : A Library of Constraints over Finite Domains. Technical Report ATS 98-2, Thomson-CSF LCR, Orsay, France, 1998.

[48] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2) :83–93, february 1974.

[49] Lockheed Martin. *GEDAE Users' Manual / GEDAE Training Course Lectures*.

[50] J. Mattioli, C. Guettier, B. Jeannet, and J. Jourdan. Spécifications de la maquette clp pour le placement automatique d'applications traitement du signal sur machine spmd. Technical Report IAS-96-4, Thomson-CSF/LCR, 1996.

[51] J. Mattioli, N. Museux, S. de Givry, J. Jourdan, and P. Savéant. A Constraint Optimization Framework for Mapping a Digital Signal Processing Application onto a Parallel Machine. In Springer Verlag, editor, *Principles and Practice of Constraint Programming*, Innovative Applications, pages –, Novembre 2001.

[52] B. Meister. Localité des données dans les opérations stencil. In *Treizième Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, Compilation et Parallélisation automatique, pages 37–42, avril 2001.

[53] R. Mohr and T.C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28 :225–233, 1986.

[54] N. Museux. De la sur-approximation des dépendances. Rapport de recherche E/227/CRI, ENSMP/CRI, 2000.

[55] N. Museux. évaluation de la capacité mémoire locale nécessaire à l'exécution d'une itération temporelle d'une tâche. Rapport de recherche TSI-00-10, Thomson-CSF/LCR, 2000.

[56] N. Museux. Extension de quelques hypothèses applicatives. Rapport de recherche TSI-00-30, Thomson-CSF/LCR, 2000.

[57] N. Museux. Radar aéroporté à forme d'onde MFR - Transcription. Rapport technique E/226/CRI - TSI/99/906, Thomson-CSF / LCR - ENSMP/CRI, 2000.

[58] N. Museux and F. Irigoin. Détection des communications à partir du partitionnement de nids de boucles. Rapport de recherche A/327/CRI, ENSMP/CRI, 2001.

[59] N. Museux and F. Irigoin. Le modèle machine Multi-SPMD. Rapport de recherche E/239/CRI, ENSMP/CRI, 2001.

[60] N. Museux, F. Irigoin, M. Barreteau, and J. Mattioli. Parallélisation Automatique d'Applications de Traitement du Signal sur Machines Parallèles. In *Treizième Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, Compilation et Parallélisation automatique, pages 55–60, avril 2001.

[61] L. Schäfers and C. Scheidler. TRAPPER : A Graphical Programming Environment for Embedded MIMD Computers. In IOS Press, editor, *1993 World Transputer Congress*, Transputer Applications and Systèmes'93, pages 1023–1034, 1993.

[62] Y. Sorel and C. Lavarenne. http ://www-rocq.inria.fr/Syndex/pub.htm.

[63] E. A. Lee team. http ://ptolemy.eecs.berkeley.edu/papers.

[64] VSIPL Consortium. http ://www.vsipl.org.

[65] H. S. Wilf. *Algorithms and Complexity*. Internet Edition, 1994.

Publications personnelles

Rapports de recherche

- *De la sur-approximation des dépendances*, rapport de recherche E/227/CRI, ENSMP/CRI.
- *Évaluation de la capacité mémoire locale nécessaire à l'exécution d'une itération temporelle d'une tâche.*, rapport de recherche TSI-00-10, Thomson-CSF/LCR.
- *Extension de quelques hypothèses applicatives*, rapport de recherche TSI-00-30, Thomson-CSF/LCR.
- *Le modèle machine Multi-SPMD*, en collaboration avec F. Irigoin, rapport de recherche E/239/CRI, ENSMP/CRI.
- *Détection des communications à partir du partitionnement de nids de boucles*, en collaboration avec F. Irigoin, rapport de recherche A/327/CRI, ENSMP/CRI.

Rapport technique

- *Radar aéroporté à forme d'onde MFR - Transcription*, Rapport technique Thomson-CSF/LCR - ENSMP/CRI (TSI/99/906)-(E/226/CRI).

Conférences et workshop nationaux

- *Programmation Logique avec Contraintes appliquée au placement : une méthodologie, un outil*, en collaboration avec C. Ancourt, M. Barreteau, F. Irigoin, J. Jourdan et J. Mattioli, Workshop AAA-2000, Janvier 2000, Inria Rocquencourt.
- *Parallélisation Automatique d'Applications de Traitement du Signal sur Machines Parallèles*, en collaboration avec M. Barreteau, F. Irigoin, et J. Mattioli, Treizième Rencontres Francophones du Parallélisme des Architectures et des Systèmes (RenPar'13), Avril 2001, Citée des sciences de la Vilette.

Conférences et workshop internationaux

- *How does constraint technology meet industrial constraints ?*, en collaboration avec P. Gérard, S. de Givry, J. Jourdan, J. Mattioli et P. Savéant, Workshop ESA "On-Board Autonomy", Octobre 2001, Hollande.
- *A Constraint Optimization Framework for Mapping a Digital Signal Processing Application onto a Parallel Machine*, en collaboration avec S. de Givry, J. Jourdan, J. Mattioli et P. Savéant, Principles and Practice of Constraint Programming conference 2001 (CP 2001), Novembre 2001, Chypre.

Table des matières

Résumé

La puissance de calcul des ordinateurs toujours croissante et leur architecture toujours plus complexe ouvrent de nouveaux horizons en matière de développement d'algorithmes de calculs scientifiques. La recherche d'une parfaite adéquation entre une application et son architecture cible est loin d'être simplifiée lorsqu'il s'agit de systèmes embarqués, exigeant généralement de courts temps de réponse, et contraints par les ressources de calcul de la machine (CPU, mémoires, etc). Cette recherche d'adéquation est reconnue être un problème NP-complet et hautement combinatoire. Paradoxalement les délais de développement accordés sont de plus en plus courts, de façon à accroître la réactivité face au marché. Par conséquent, bénéficier d'outils d'aide au développement dès la conception des algorithmes et/ou des machines est primordial.

Dans cette optique, THALES RESEARCH & TECHNOLOGY - FRANCE (ex-Laboratoire Central de Recherches de Thomson-CSF) et le Centre de Recherche en Informatique, de l'École des Mines de Paris, ont proposé dès 1995 une méthode de placement automatique d'applications de traitement du signal systématique reposant sur la modélisation concurrente et la programmation par contraintes : PLC^2. Le contexte applicatif et architectural était alors idéalisé puisqu'il s'agissait d'en étudier la faisabilité.

Cette thèse a eu pour ambition d'étendre ce contexte en prenant en compte des algorithmes et des architectures réels plus complexes. De ce fait, les extensions du domaine applicatif effectuées, la modélisation d'une architecture multi-SPMD et une étude formelle sur la détection des communications dépendantes du placement ont été étudiées dans ce document.

Mots-clés : programmation par contraintes - parallélisation automatique - modélisation concurrente - partitionnement - ordonnancement - multi-spmd - détection de communications

---

Abstract

Because of the increase of both the computing capabilities and the architecture complexity of today's computers, more design possibilities are offered to programmers. But finding the perfect mapping of an algorithm onto a machine is more difficult for on-board algorithms. These real-time algorithms are constrained by the computing resource of the target machine (such as cpu or memories). Such mapping problems are known to be NP-complete and strongly combinatorial. Moreover, the time allowed to develop a program is getting shorter in order to be reactive to the market. Thus providing tools to help the development of an application and/or the architecture from the design phase is crucial.

With this objective, THALES RESEARCH & TECHNOLOGY - FRANCE (formerly named Thomson-CSF Corporate Research Laboratory) in collaboration with the Centre de Recherche en Informatique of École des Mines de Paris, proposed in 1995 an automatic mapping methodology for systematic signal processing applications based on Constraint Programming and on concurrent modeling, called PLC^2. At the time the applicative and architectural context was simplified as it was a feasibility study.

The applicative context is now extended by taking into account more complex real applications and target machines. This document presents the domain applicative extensions realised. It also presents a mathematic model for a multi-SPMD machine, as well as a formal study of the detection of physical data flows between processors which depend on loop-tiling.

Key-words : constraint programming - automatic parallelisation - concurrent modeling - tiling - scheduling - multi-spmd - data transfer detection