Analyses statiques et transformations de programmes: de la parallélisation à la différentiation

Laurent Hascoët
INRIA Sophia-Antipolis

30 mars 2004

1 Introduction

Le rapport que nous présentons fait une synthèse de recherches effectuées dans le domaine des outils logiciels pour l'analyse et la transformation de programmes. Ceci est un résumé en français du rapport. Pour faciliter la lecture, la numérotation des chapitres est identique dans les deux documents.

La motivation des recherches n'est pas nouvelle: pour éviter la programmation manuelle longue, complexe et source d'erreurs, il faut construire certaines parties des programmes automatiquement, à l'aide d'outils logiciels. Cette génération peut partir de spécifications écrites dans des formalismes spécialisés. Alternativement, elle peut partir d'un programme existant et modifier ou augmenter ses fonctionnalités. Nos recherches se placent dans cette deuxième optique. Ce domaine est naturellement proche de celui de la compilation.

Dans ce rapport, nous examinerons plus particulièrement deux transformations de programmes: la Parallélisation et la Différentiation.

La Parallélisation est le sujet le plus connu, et il mobilise toujours de nombreuses équipes de recherche. Les modèles de parallélisation évoluent constamment, peut-être aussi au gré des modes. Le dernier avatar semble être la programmation des grilles de calcul ("grid computing"). Mais pour tous ces modèles, la problématique change peu: il faut toujours identifier des portions de calcul indépendantes, ainsi que la répartition mémoire qui

minimise les communications. Ce qui change est le coût relatif des calculs, de la mémoire et des communications. Nos recherches sur la parallélisation automatique des codes remontent à la période 1990-2000. Dans ce rapport, nous présentons la contribution originale de nos recherches, mais aussi les outils conceptuels classiques du domaine. La parallélisation sert de justification et d'introduction à ces outils, que nous transposerons au deuxième sujet: la Différentiation Automatique.

La Différentiation Automatique (DA) de programmes est un sujet beaucoup moins largement connu, dont les applications potentielles en Calcul Scientifique sont immenses. Il s'agit aussi d'un domaine de recherche relativement récent. Le principe est de transformer le programme qui évalue une fonction mathématique F, qui peut être très compliquée, pour obtenir le programme qui calcule analytiquement les dérivées de F. Les dérivées obtenues peuvent ensuite être utilisées dans des applications de Calcul Scientifique. En particulier, les problèmes d'optimisation, les algorithmes de Newton ou encore les problèmes inverses utilisent des *gradients* d'une fonction coût scalaire. Ces gradients peuvent être obtenus grâce au mode dit *inverse* de la DA, avec un coût théorique qui n'est qu'un petit multiple du coût du programme initial. Notre sujet de recherche de ces dernières années a consisté à utiliser les concepts de la compilation et de la parallélisation pour construire des programmes différentiés inverses qui s'approchent de ce coût théorique.

La figure 1 montre la structure des chapitres 3 et 4 du rapport, en insistant sur cette transposition des concepts de la parallélisation vers la DA. Le chapitre 4 comporte en outre une présentation détaillée de la DA, ainsi que deux exemples d'utilisation de gradients calculés par le mode inverse pour des problèmes de calcul scientifique en vraie grandeur.

Tant pour la Parallélisation que pour la Différentiation, nous pensons qu'il est nécessaire que les idées provenant de la recherche soient implémentées de manière efficace dans des outils réels, disponibles pour des utilisateurs académiques et industriels. Cela permet en particulier de faire remonter dans les équipes de recherche les besoins des utilisateurs. Cela signifie des contraintes de robustesse et d'efficacité. Il faut que l'outil traite correctement l'intégralité du langage de programmation considéré, et pas seulement un sous-langage. Cela met souvent en lumière des aspects injustement négligés, les boucles `WHILE` par exemple. Il faut enfin que les algorithmes implémentés passent à l'échelle sur des gros codes. Les sections 3.4 et 4.6 décrivent deux outils complets, pour la parallélisation des boucles et la DA respectivement, qui sont ou ont été utilisés régulièrement sur des codes industriels.

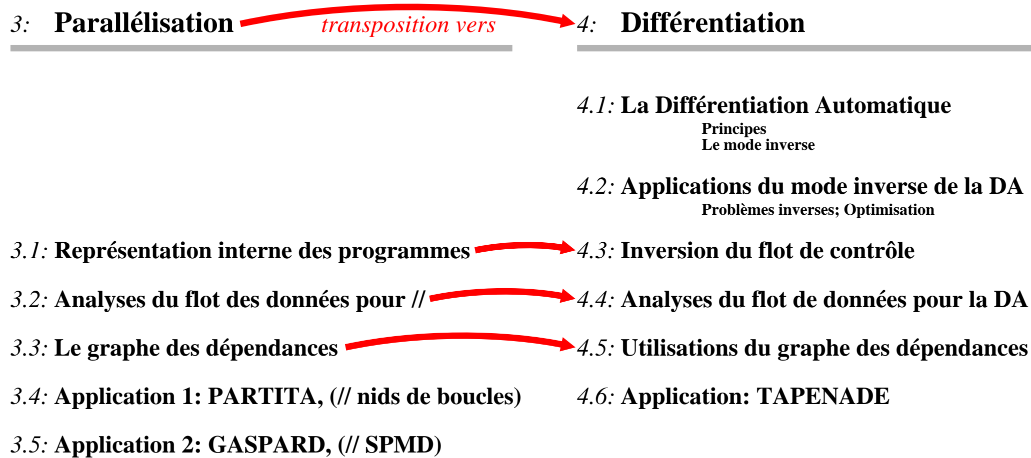

FIG. 1 – *Structures comparées des chapitres 3 et 4*

2 Les analyses statiques et les transformations de programmes

Nous présentons des notions classiques de compilation, spécialement utiles pour les analyses et transformations de programme.

2.1 Analyses statiques

Les analyses dont nous parlerons sont *statiques*. Cela signifie qu'elles ne dépendent pas d'un jeu de données particulier à l'appel. Les résultats des analyses statiques doivent être valables quelles que soient les données. Ces résultats doivent donc être plus abstraits que de simples valeurs de variables. Les valeurs possibles d'une variable pourront par exemple être représentées par un intervalle, un ensemble ou une expression régulière. On doit donc calculer l'effet des opérations arithmétiques du programme sur ces valeurs abstraites. Un cadre théorique classique est "l'interprétation abstraite".

Théoriquement, les analyses statiques sont indécidables: il existera toujours un programme pour lequel un algorithme d'analyse statique ne saura pas décider entre "oui" et "non". Mais cette limite est théorique et très lointaine: en pratique, on rencontre de nombreux obstacles quasi insurmontables bien avant l'indécidabilité théorique. On choisit souvent de laisser un résul-

tat indéterminé simplement parce que l'analyse plus fine pour le déterminer existe mais serait trop coûteuse. Cela arrive souvent avec les pointeurs, les accès aux tableaux ou simplement un flot de contrôle complexe. En conclusion, toute analyse statique non triviale doit être prête à utiliser et rendre trois réponses possibles au lieu de deux: “oui”, “non”, ou “ne sait pas”.

Un autre problème vient du flot de contrôle. Un fragment d'un programme représente en fait l'ensemble de ses exécutions (ses “instances”). La propriéte analysée peut varier selon les instances. Doit-on alors ne calculer qu'une seule propriété, plus faible mais valable pour toutes les instances (“généralisation”), ou plutôt une propriéte particulière, précise, pour chaque instance (“spécialisation”)? Ce choix conditionne énormément les résultats. Pour des raisons évidentes d'efficacité, on a tendance a préférer la généralisation (boucles, procédures), et à réserver la spécialisation à des emplacements bien précis, souvent désignés par l'utilisateur final.

Nous illustrons les techniques ci-dessus sur “l'évaluation partielle”, une analyse statique qui tente de spécialiser un programme pour un sous-ensemble particulier de ses données. On espère ainsi obtenir un programme spécialisé plus petit et plus efficace. Les deux difficultés principales correspondent à ce que nous avons évoqué plus haut:

- On veut propager des informations sur les valeurs des variables, à partir des informations connues sur les données. On choisit une représentation abstraite de ces valeurs, puis on combine ces valeurs abstraites, ce qui conduit immanquablement à des approximations.
- Au niveau des procédures, on doit choisir entre spécialisation et généralisation. Cela est décidé par des heuristiques.

L'exemple choisi est un peu particulier, puisque le programme est un ensemble de règles d'inférence, qui sont ensuite combinées par un constructeur d'arbres de preuve. Les procédures sont donc remplacées par les règles d'inférence. Néanmoins, l'évaluation partielle est possible. Le programme initial réalisait la reconnaissance d'un mot donné par une expression régulière donnée. Après évaluation partielle pour une expression régulière donnée E, nous obtenons un nouvel ensemble de règles d'inférence qui reconnaît efficacement si un mot donné appartient au langage de E.

2.2 Représentation interne des programmes

Nous décrivons la représentation interne des programmes qui paraît la mieux adaptée aux analyses statiques, mais aussi aux transformations de programmes telles que la DA (*cf* section 4.3). Cette représentation consiste en un graphe d'appel au niveau supérieur, dans lequel chaque nœud représente une procédure. Chaque procédure est elle-même représentée par son graphe de flot de contrôle, dans lequel chaque nœud ("bloc de base") est une liste d'instructions toujours exécutées séquentiellement. Chaque instruction est représentée par son arbre de syntaxe abstraite. Enfin, à chaque bloc de base, on associe une table des symboles, qui collectionne les informations statiques attachées aux symboles (types, valeur constante...)

Nous décrivons comment un graphe de flot de contrôle peut être structuré, réalisant ainsi une synthèse avec les avantages d'une représentation arborescente structurée classique.

2.3 Analyses du flot des données

Les analyses de flot de données sont des analyses statiques qui recherchent des informations sur les valeurs des variables. Nous décrivons comment ces analyses peuvent être effectuées, sur le graphe de flot de contrôle, par un algorithme itératif qui résout un ensemble d'équations souvent ensemblistes. Ces équations spécifient l'analyse statique considérée, et se nomment les "équations data-flow". Nous donnons une condition nécessaire très utile pour montrer que l'algorithme itératif termine. Nous décrivons aussi des techniques d'implémentation efficace. Nous illustrons cela sur un exemple classique: l'analyse des variables "Lues-Ecrites", c'est-à-dire les variables utilisées et réécrites par une procédure donnée. Sur cet exemple, nous soulignons une représentation interne des informations sur les variables qui permet de passer aisément de l'espace de nommage de la procédure appelée à celui de la procédure appelante, dans le cas général autorisant des variables globales. Nous utilisons cette représentation dans les implémentations décrites.

2.4 Difficultés classiques

Nous soulignons les difficultés classiques liées aux tableaux, aux pointeurs, et à l'aliasing. Pour les tableaux, nous décrivons l'état de l'art qui consiste à identifier des régions de tableaux, correspondant le plus souvent à des accès

réguliers. L'aliasing est le problème qui survient lorque deux variables de noms différents peuvent accéder en fait à la même place mémoire. Dans ce cas, de nombreuses transformations peuvent rendre un résultat faux. Nous donnons un exemple en parallélisation.

3 Parallélisation

3.1 Représentation interne des programmes

Nous montrons comment nous utilisons la représentation interne définie plus haut pour la parallélisation. Le "front-end" d'un outil de parallélisation doit donc construire un graphe d'appel, un graphe de flot par procédure, et un arbre de syntaxe abstraite par instruction atomique. A la différence d'un compilateur, nous voulons que le paralléliseur reconstruise un programme source: nous montrons comment le "back-end" de l'outil effectue cette remontée progressive vers un nouveau programme textuel. Au centre, entre "front-end" et "back-end", se trouvent les analyses statiques, dont le résultat principal est le graphe des dépendances (*cf* section 3.3), outil central de la parallélisation.

3.2 Analyses du flot des données pour la parallélisation

Nous décrivons les analyses statiques nécessaires à une bonne parallélisation, ainsi que leurs dépendances mutuelles. Il y a d'ailleurs là un problème subtil puisque certaines informations ne pourraient être obtenues qu'après plusieurs itérations de la suite des analyses, du fait de dépendances mutuelles entre ces analyses.

Néanmoins, une suite raisonnable d'analyses statique va détecter:

1. les destinations possibles des pointeurs,
2. les contraintes sur les valeurs des variables (intervalles),
3. le code mort dû à des tests toujours vrais ou faux,
4. les variables d'induction dans les boucles,
5. les paramètres lus et écrits par chaque procédure,
6. les tableaux totalement écrits par les boucles. Il s'agit d'une approximation grossière de l'analyse des régions de tableaux.

3.3 Le graphe des dépendances

Nous donnons une définition d'une dépendance de données, qui est une contrainte d'ordre d'exécution entre deux instances d'instructions, résultant de l'ordre d'exécution standard, séquentiel, du programme initial non parallélisé. L'ensemble des dépendances de données forme le graphe des dépendances, un graphe entre toutes les instances d'instructions du programme. Ce graphe est acyclique: c'est un sous-ordre de l'ordre d'exécution standard. Son intérêt principal est que tout réordonnancement du programme, qui définit un nouvel ordre d'exécution (ou un ensemble d'ordres d'exécution dans le cas de langages parallèles), conservera les résultats du programme initial si le nouvel ordre admet aussi le graphe des dépendances comme sous-ordre.

Dans la pratique, on projette ce graphe des dépendances sur l'ensemble des instructions textuelles, c'est-à-dire qu'on fusionne toutes les instances d'une même instruction en un seul nœud. On obtient alors un graphe cyclique, que l'on peut utiliser comme le premier graphe des dépendances (acyclique) à la condition d'ajouter une information de "distance" sur chacune des dépendances projetées. Cette distance représente en général la différence entre les indices des boucles englobantes, entre l'origine et la destination de la dépendance. Ce graphe des dépendances réduit a perdu une partie de l'information par rapport à sa version acyclique, mais il est souvent le seul qu'on puisse construire réellement.

Les dépendances de données représentent également le trafic mémoire entre les différentes parties du programme, et sont donc l'outil fondamental pour étudier la gestion mémoire des programmes parallélisés avec mémoire distribuée.

Dans le cas des boucles qui utilisent des tableaux, le calcul des distances des dépendances est particulièrement important. Considérons deux accès à un tableau `T`. Il y a une dépendance entre eux lorqu'ils sont susceptibles d'accéder au même emplacement mémoire. Lorsque l'on sait, grâce à la détection des variables d'induction par exemple, relier les indices des accès aux tableaux aux indices des boucles englobantes, on peut ramener le problème de la dépendance entre ces deux accès à un ensemble d'équations reliant les indices des boucles englobantes au moment de l'exécution de l'un et l'autre accès. Lorsque ces équations sont de plus linéaires, par exemple parce que les indices des tableaux sont affines par rapport aux indices de boucle, on est ramené à un système d'équations linéaires, plus quelques inéquations lorqu'on utilise les informations disponibles sur les bornes des valeurs des variables.

On résout ce système pour vérifier que la dépendance a bien lieu ou pas, et si elle a lieu obtenir des contraintes sur la distance de dépendance.

3.4 Application 1: PARTITA, un outil de parallélisation de nids de boucles

PARTITA est un outil semi-automatique de parallélisation de programmes, que nous avons développé à l'intérieur de l'environnement commercial FORESYS d'analyse de programmes FORTRAN. Il est question ici de parallélisation de boucles, qui construit des programmes équivalents vectorisés ou parallélisés dans des dialectes tels que FORTRAN95, FORTRAN-HPF, ou OPENMP. Le principe de PARTITA est de construire le graphe des dépendances pour chaque nid de boucles ou fragment de programme, puis de construire une représentation arborescente des boucles contenues dans ce fragment, que nous appellerons l'arbre des groupes. La figure 2 montre l'arbre des groupes d'un fragment de code déjà parallélisé. Pour chaque arbre des groupes, le graphe

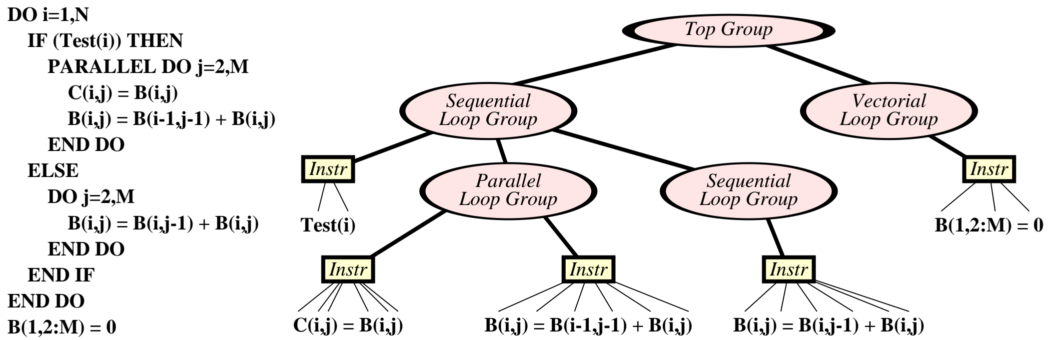

FIG. 2 – *L'arbre des groupes d'un nid de boucles*

des dépendances entre les opérations individuelles du programme se projette naturellement entre les nœuds de l'arbre des groupes. On obtient alors de petits graphes de dépendances réduits, à l'intérieur de chaque niveau de boucle imbriquée (*cf* figure 3). Le rapport donne les règles précises de cette projection, qui étend l'idée de "dependence nesting level" (Kennedy-Allen).

Pour chaque niveau de boucle, le graphe de dépendance réduit détermine les transformations possibles de l'arbre des groupes. Par exemple, on sait que les cycles de ce graphe correspondent aux parties non parallélisables

immédiatement. Déterminer une distribution d'une boucle qui sépare les parties parallélisables des autres revient donc à une condensation acyclique du graphe de dépendances.

Plus généralement, le graphe de dépendance projeté sur l'arbre des groupes permet de décider de l'applicabilité de plusieurs transformations de boucles, qui chacune se traduiront par une transformation de l'arbre des boucles. Il s'agit de:

- l'extraction du code invariant de boucle,
- l'expansion des variables,
- la distribution (fission) et la fusion de boucles,
- la détection des opérations de réduction,
- la rotation de boucle,
- l'échange de boucles imbriquées.
- Les transformations unimodulaires, le "strip mining" et le "tiling", par une interface avec l'outil BOUCLETTE développé au LIP.

Une heuristique pilote ces différent choix, pour aboutir itérativement à un arbre de groupes final, que l'on considère optimal pour le langage parallèle cible. PARTITA propose une paramétrisation très complète de cette cible, qui décrit en particulier les possibilités (vectorielles, parallèles, etc...) du langage cible, ainsi que les coûts des diverses opérations, structures de contrôle, ou accès mémoire. Cela permet à l'heuristique de refuser certaines transformations si elles mènent à des structures de contrôle parallèle non disponibles ou si le coût fixe de parallélisation est trop élevé par rapport au bénéfice attendu.

Pour illustrer la généralité de la méthode, la figure 3 montre qu'elle permet de paralléliser aussi des boucles "naturelles", c'est-à-dire des boucles construites avec des `GOTO` au lieu de `DO`. Cela est hors de portée de la plupart des autres outils de parallélisation commerciaux. Le résultat est une boucle séquentielle, qui compte les itérations et effectue les opérations non parallélisables, suivi d'une boucle parallèle qui utilise ce nombre d'itérations.

L'étape finale, assez mécanique, reconstruit un fragment de code parallélisé à partir de l'arbre des groupes final. La figure 4 illustre le résultat, en insistant sur la différence entre les programmes construits si l'on change la spécification de la cible de parallélisation.

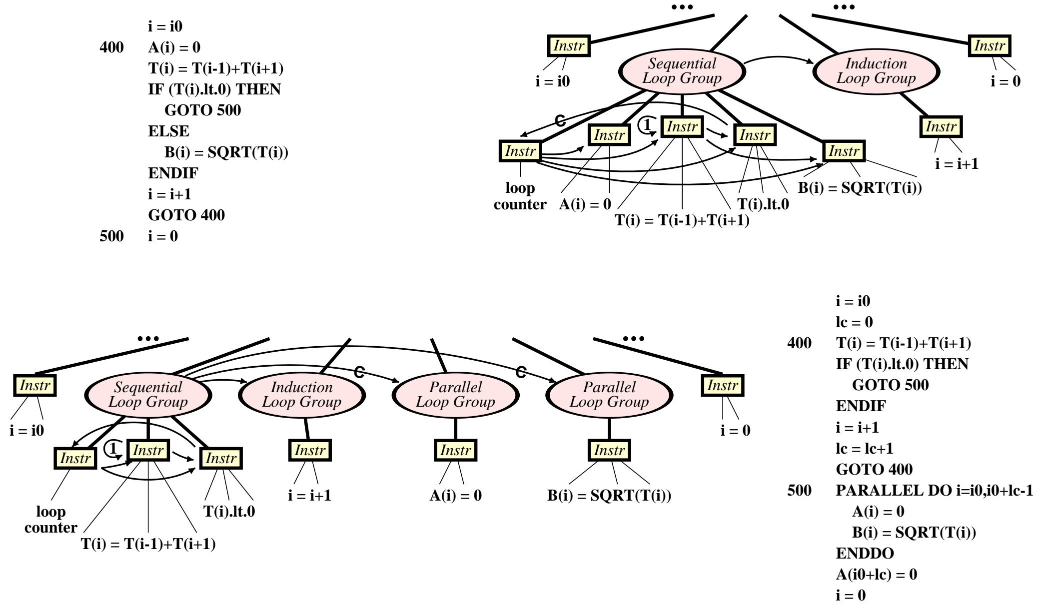

Fig. 3 – *Parallélisation par distribution d'une boucle naturelle*

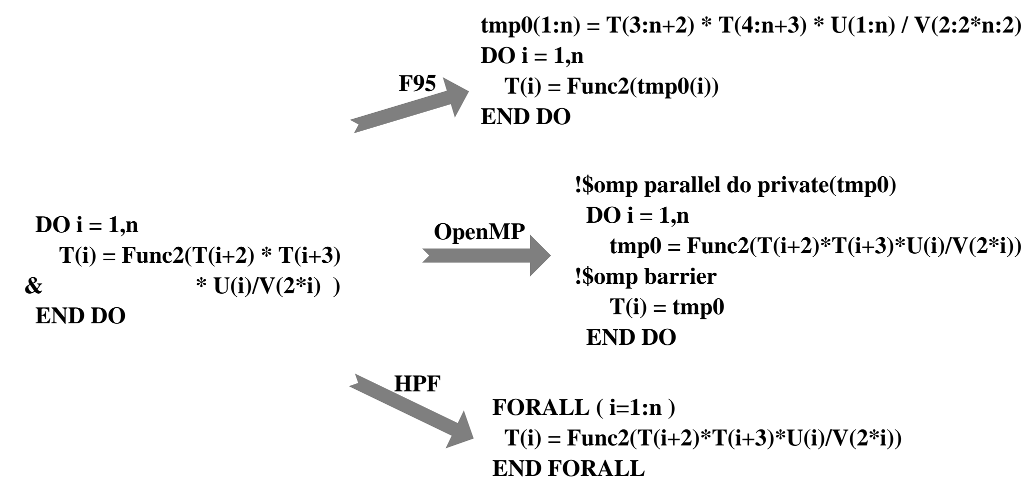

Fig. 4 – *Resultats de l'heuristique générale pour des cibles de parallélisation différentes*

3.5 Application 2: GASPARD, un outil de parallélisation SPMD

GASPARD est un prototype d'outil de parallélisation SPMD (Single Program Multiple Data). GASPARD a été construit au-dessus de PARTITA, pour bénéficier de son analyse de dépendances. Pour GASPARD, nous avons défini une méthode de parallélisation originale, qui essaie de formaliser les décisions intuitives des programmeurs.

Dans le modèle SPMD, le même programme tourne sur des processeurs séparés, sur ses propres données, et communique si nécessaire avec les autres processeurs par échange de messages. Ce modèle est particulièrement adapté en calcul scientifique, lorsqu'un calcul identique est effectué sur chaque nœud d'un gros maillage. On peut alors partitionner ce maillage et affecter chaque morceau à un processeur différent. Lorsqu'un nœud est à l'interface entre deux morceaux, il reçoit les valeurs nécessaires par un message. En général, on définit une certaine quantité de recouvrement, c'est-à-dire une duplication des nœuds de l'interface, pour que ces communications puissent être regroupées dans le temps. Soulignons aussi que les valeurs définies sur un maillage peuvent être attachées aux nœuds, mais aussi aux arêtes, aux faces ou aux éléments, ou enfin être réduites à l'état d'un scalaire global.

L'idée de la méthode est de définir des “états” de synchronisation de ces valeurs, par rapport aux interfaces entre sous-maillages, et plus exactement par rapport aux recouvrements prévus aux interfaces. Par exemple, une valeur attachée aux arêtes peut être complètement synchrone, c'est-à-dire que les arêtes des recouvrements ainsi que leurs arêtes d'origine contiennent les mêmes valeurs, et cette valeur est la même que celle que l'on calculerait à cet endroit dans le programme séquentiel. Alternativement, la valeur contenue dans l'arête du recouvrement peut être inexacte. On sait alors qu'on ne doit pas l'utiliser dans la suite du calcul avant de l'avoir re-synchronisée, ce qui demandera un échange de messages.

Enfin, à l'occasion des calculs sur le maillage, les valeurs sur les nœuds par exemple peuvent servir à calculer des valeurs sur les arêtes, et réciproquement, et cela propage aussi progressivement un certaine désynchronisation sur les recouvrements.

On en arrive donc à définir un automate fini qui décrit la façon dont l'état des variables évolue au cours du calcul, depuis des états synchrones vers des états non synchrones. Arrivé à un certain niveau de désynchronisation, on ne peut plus utiliser la variable avant d'avoir remis les valeurs à jour par un

échange de messages. Cet échange est lui aussi représenté par une transition de l'automate fini qui "remonte" vers des états synchronisés.

En déroulant cet automate sur le programme, c'est-à-dire en trouvant une injection des nœuds du graphe des dépendances vers les états de l'automate, et des dépendances vers les transitions de l'automate, on détermine les emplacements du programme auxquels il faut ajouter les instructions de communication pour préserver le résultat final du programme.

4 Différentiation Automatique

4.1 La Différentiation Automatique des programmes

La Différentiation Automatique (DA) est une technique de transformation de programme pour différentier au sens mathématique la fonction calculée par un programme. Etant donné un programme `P` qui évalue une fonction $F : X \in I\!R^m \mapsto Y \in I\!R^n$, il s'agit de construire un programme `P'` qui calcule *analytiquement* certaines dérivées de F.

Le principe est de considérer le programme `P` comme une suite d'instructions

$$\mathtt{I}_1; \mathtt{I}_2; \ldots \mathtt{I}_{\mathtt{p}-1}; \mathtt{I}_{\mathtt{p}};$$

éventuellement variable suivant les variations du contrôle, puis d'identifier cette suite d'instructions à une composition de fonctions élémentaires, une par instruction:

$$f = f_p \circ f_{p-1} \circ \ldots \circ f_1$$

Chaque fonction f_k opère dans l'espace des vecteurs des valeurs de toutes les variables. En un "point" X, vecteur de valeurs initiales passées à `P`, on différentie f comme une fonction composée:

$$\begin{aligned} f'(X) \quad = \quad & (f'_p \circ f_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(X)) \\ & .\ (f'_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(X)) \\ & .\ \ldots \\ & .\ (f'_1(X)) \\ = \quad & f'_p(X_{p-1})\ .\ f'_{p-1}(X_{p-2})\ .\ \ldots\ .\ f'_1(X_0) \end{aligned}$$

On a simplifié la notation en introduisant $X_0 = X$ et $X_k = f_k(X_{k-1})$, les valeurs des variables juste après l'exécutions des k premières instructions.

En pratique, la dimension des vecteurs, c'est-à-dire le nombre de variables du programme, est très grande et on préfère ne pas calculer entièrement cette Jacobienne $f'(X)$. Pour de nombreuses applications, on sait se contenter d'une projection particulière:

- On peut avoir besoin d'une dérivée directionnelle suivant une direction $\dot{X}$, qui est:

  $$\dot{Y} = f'(X).\dot{X} = f'_p(X_{p-1}) \;.\; f'_{p-1}(X_{p-2}) \;.\; \ldots \;.\; f'_1(X_0) \;.\; \dot{X}.$$

  Cela correspond au mode "direct" (ou "tangent") de la DA. Comme il est plus économique de calculer le produit ci-dessus de la gauche vers la droite (produits matrice × vecteur), on voit que le calcul des dérivées suit directement le calcul initial. La construction du programme différentié est donc relativement aisée, comme illustré sur l'exemple suivant:

original: `T,U ↦ e`	**mode tangent:** `T,Ṫ,U,U̇ ↦ e,ė`
	`ė2 = 0.0`
`e2 = 0.0`	`e2 = 0.0`
`do i=1,n`	`do i=1,n`
	`   ė1 = Ṫ(i)-U̇(i)`
`   e1 = T(i)-U(i)`	`   e1 = T(i)-U(i)`
	`   ė2 = ė2 + 2.0*e1*ė1`
`   e2 = e2 + e1*e1`	`   e2 = e2 + e1*e1`
`end do`	`end do`
	`ė = 0.5*ė2/SQRT(e2)`
`e = SQRT(e2)`	`e = SQRT(e2)`

- On peut avoir besoin d'un gradient d'une combinaison linéaire des résultats Y, définie par des pondérations $\overline{Y}$. Ce gradient est alors:

  $$\overline{X} = f'^t(X).\overline{Y} = f'^t_1(X_0) \;.\; f'^t_2(X_1) \;.\; \ldots \;.\; f'^t_{p-1}(X_{p-2}) \;.\; f'^t_p(X_{p-1}) \;.\; \overline{Y}.$$

  Cela correspond au mode "inverse" (ou "adjoint") de la DA. On observe à nouveau qu'il est plus économique de calculer le produit ci-dessus de la gauche vers la droite, mais cela entraîne d'effectuer le calcul des dérivées dans l'ordre *inverse* des instructions du programme initial. Cette approche donne une complexité théorique excellente, puisque le calcul du gradient complet coûte seulement un petit multiple du coût de la fonction f. Cependant ce coût augmente dans la pratique car il faut un

mécanisme pour fournir aux instructions dérivées les valeurs intermédiaires du programme dans l'ordre inverse de leur création. Reprenons l'exemple choisi pour le mode tangent. On voit sur le tableau suivant que le code "inverse" effectue les instructions différentiées proprement dites dans une phase appelée *passe arrière*, et qu'on doit la faire précéder d'une *passe avant* qui exécute le programme initial en préparant au passage la pile des valeurs nécessaires aux dérivées. On remarque que le flot de contrôle de la passe arrière est l'exact inverse du programme initial.

mode inverse: $\mathtt{T,U,\overline{e}} \mapsto \mathtt{\overline{T},\overline{U}}$	
passe avant:	*passe arrière:*
`e2 = 0.0`	$\mathtt{\overline{e2} = 0.0}$
`do i=1,n`	$\mathtt{\overline{e1} = 0.0}$
`PUSH(e1)`	$\mathtt{\overline{e2} = \overline{e2} + 0.5*\overline{e}/SQRT(e2)}$
`e1 = T(i)-U(i)`	$\mathtt{\overline{e} = 0.0}$
`PUSH(e2)`	`do i=n,1,-1`
`e2 = e2 + e1*e1`	`POP(e2)`
`end do`	$\mathtt{\overline{e1} = \overline{e1} + 2*e1*\overline{e2}}$
`e = SQRT(e2)`	`POP(e1)`
	$\mathtt{\overline{T}(i) = \overline{T}(i) + \overline{e1}}$
	$\mathtt{\overline{U}(i) = \overline{U}(i) - \overline{e1}}$
	$\mathtt{\overline{e1} = 0.0}$
	`end do`
	$\mathtt{\overline{e2} = 0.0}$

Cette restauration des valeurs antérieures est bien sûr la grande faiblesse du mode inverse, car un gros programme demande une taille de pile très (trop?) importante. Le "checkpointing" ainsi que les autres optimisations que nous avons étudiées, répondent à ce problème. Ces techniques sont décrites dans la suite.

Les gradients sont d'un intérêt immense en calcul scientifique, comme on le verra dans la section 4.2. L'essentiel de nos recherches a donc porté sur l'amélioration du mode inverse que peuvent apporter les techniques issues de la compilation et de la parallélisation. Remarquons d'ailleurs que le système de pile que nous décrivons ici n'est pas l'unique solution. On peut aussi restaurer les valeurs intermédiaires en recalculant le programme initial à partir d'un point de départ choisi à l'avance. On économise de la mémoire au prix de calculs suppémentaires. C'est l'approche de l'outil de DA TAF. Un

problème ouvert est de concilier ces deux approches dans un cadre général, pour trouver la combinaison optimale. Quelle que soit l'approche, un compromis stockage-recalcul à un plus haut niveau est nécessaire, qui se nomme "checkpointing". Nous l'expliquons pour notre approche de stockage des valeurs intermédiaires, mais il possède un équivalent dans l'approche "recalcul". Le checkpointing consiste, pour un fragment donné du programme, à ne pas mémoriser les valeurs intermédiaires pendant la passe avant. Bien entendu, la passe arrière ne peut fonctionner normalement que jusqu'à rencontrer la fin du fragment. A ce moment, on réinstalle les valeurs qui existaient au début du fragment, et on relance la passe avant de ce fragment, *avec* mémorisation cette fois. La passe arrière peut alors se terminer normalement. Le bénéfice est une réduction de la taille maximum de la pile. Le coût est une exécution dupliquée du fragment, ainsi qu'une mémorisation des valeurs nécessaires à son exécution, qu'on appelle une "photo". Idéalement, la taille mémoire maximale et le nombre maximum de recalculs peuvent augmenter seulement comme le logarithme de la longueur de l'exécution du programme.

4.2 Applications emblématiques de la DA inverse

On décrit deux des grands domaines d'utilisation des gradients. Le premier recouvre l'identification de paramètres mal connus, ou l'assimilation des données. Plus généralement, ce sont des problèmes inverses. Supposons que des valeurs inconnues γ déterminent des valeurs observables W, à travers un modèle physique que l'on sait modéliser par des équations mathématiques (souvent des équations aux dérivées partielles) $\Psi(\gamma,W) = 0$. On définit alors une fonction coût $j(\gamma)$ qui mesure l'écart entre le W calculé et le W observé. Le gradient $j'(\gamma)$ indique la direction du prochain pas d'optimisation pour trouver des valeurs de γ qui "collent" au mieux aux observations. Si la résolution en W de $\Psi(\gamma,W) = 0$ ainsi que le calcul de j sont implémentés par un programme `P`, la différentiation de `P` en mode inverse fournit un programme $\overline{\mathtt{P}}$ qui calcule effectivement $j'(\gamma)$.

L'autre domaine porte sur l'optimisation de formes, principalement en mécanique des fluides. Nous avons étudié ce problème pour l'application de la réduction du bang sonique en dessous d'un avion supersonique. Ici aussi, on cherche à minimiser une fonction coût définie sur un état W, relié implicitement aux paramètres de forme γ par un système d'équations aux dérivées partielles $\Psi(\gamma,W) = 0$. Dans notre étude, nous recherchons un compromis efficace entre DA et résolution manuelle des équations du gradient (méthode

dite des équations adjointes). L'idée est de remonter aux équations mathématiques qui définissent le gradient, et de séparer dans ces équations les parties qu'on implémentera par Différentiation Automatique de parties du programme initial, des parties qu'on implémentera par un algorithme de résolution adapté. En fait, on considère que la DA du programme initial global est inefficace parce qu'elle différentie non seulement l'assemblage des systèmes linéaires à résoudre, ce qui est parfaitement justifié, mais aussi leur résolution, ce qui est plus discutable. On retrouve ainsi le problème classique suivant: la différentiation d'une résolution itérative converge-t-elle aussi bien que la résolution du problème initial? Pour une large classe de problèmes mathématiques, la réponse est que la différentiée de la résolution converge bien, mais pas nécessairement à la même vitesse. Notre approche permet de contourner la difficulté, en ne différentiant pas la partie "résolution" proprement dite.

4.3 Inversion du flot de contrôle sur le graphe de flot

On a vu que la passe arrière du mode inverse doit reproduire l'inverse du contrôle du programme initial. Nous avons donné une manière d'inverser ce contrôle qui utilise la représentation de graphe de flot. Les autres outils de DA inversent ce contrôle sur des arbres de syntaxe abstraite. Or le contrôle est mal représenté sur ces arbres dans le cas des sauts (`GOTO`, `EXIT`, `CYCLE` et autres `exception`), ce qui fait que la plupart de ces outils imposent des contraintes de style de programmation sur le programme à différentier.

En revanche, les graphes de flot représentent le contrôle d'une manière simple et uniforme. La construction du graphe de flot inverse est alors mécanique. La reconstruction du programme différentié à partir de ce graphe de flot ne présente pas de difficulté particulière. La remarque essentielle est que l'on doit respecter la structure de pile des mémorisations entre les passes avant et arrière, et plus généralement la symétrie entre ces deux passes. On ne doit donc pas mémoriser le contrôle là où les flèches *divergent*, mais là où elles *convergent*, pour pouvoir le réutiliser là où les flèches de la passe arrière divergent. La figure 5 illustre cette technique sur deux graphes de flot représentatifs.

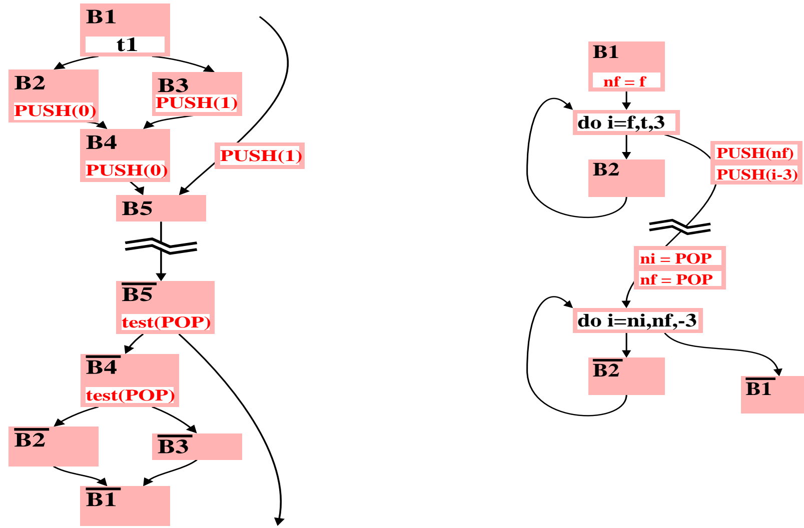

FIG. 5 – *Inversion du contrôle sur le graphe de flot*

4.4 Analyses statiques du flot de données pour la DA

Nous décrivons principalement deux analyses statiques essentielles, nommées *activité* et *TBR*.

L'activité est une analyse classique de la DA. Dans le cas très fréquent où seules sont utiles les dérivées de certains résultats par rapport à certaines entrées, cette analyse propage sur chaque occurrence de variable dans le programme les deux informations suivantes:

- **variabilité:** La variable dépend de manière différentiable d'une donnée désignée. Sa dérivée partielle par rapport aux entrées est donc non nulle a priori.
- **utilité:** La variable influence de manière différentiable un résultat désigné. Sa dérivée est donc utile au résultat final.

Une variable est dite *active* si elle est à la fois *variable* et *utile*. Seules les différentielles des variables actives doivent être calculées par le programme dérivé. L'activité est une analyse déjà classique, même si la partie *utilité* est souvent négligée. Notre contribution a été de donner une spécification claire de ces analyses en termes d'équations data-flow, d'étudier leur complexité et de prouver leur terminaison. Nous montrons aussi comment déduire des

équations optimisées pour les graphes de flot *structurés*.

L'analyse TBR ("*To Be Restored*") est spécifique du mode inverse. Elle détermine si une variable du programme initial, donc calculée lors de la passe avant, est utilisée dans les instructions différentiées de la passe arrière. Typiquement, les variables qui n'interviennent que dans des expressions linéaires n'apparaissent pas dans les instructions dérivées. En revanche, les variables qui sont utilisées par des expressions non-linéaires, ou dans des indices de tableaux dérivés, seront utilisées dans la passe arrière et devront donc être sauvegardées d'une manière ou d'une autre. L'analyse TBR permet donc de réduire la consommation mémoire du mode inverse. Notre contribution a été de participer à la mise en lumière de cette analyse, et d'en donner une spécification claire en termes d'équations data-flow. Nous avons également prouvé la terminaison et déduit des équations plus efficaces dans le cas structuré.

Nous décrivons aussi deux autres analyses statiques pour le mode inverse, dont l'étude est commencée et que nous poursuivrons jusqu'à leur implémentation dans notre outil TAPENADE, la détection de *photos optimales* et l'élimination du *code mort adjoint*.

Lorsqu'on décide d'appliquer le *checkpointing* à un (fragment de) sous-programme P, on doit prendre en *photo* l'ensemble minimal des variables nécessaires à la réexécution ultérieure de P. Quel est cet ensemble? Une première réponse est l'ensemble des variables lues par P. En fait, on peut raffiner cette estimation: seules devront être prises en photo les variables dont la valeur est réécrite entre la première et la deuxième exécution de P. Les constantes, par exemple, ne changent pas et ne doivent pas être prises en photo. Par ailleurs, on remarque que la deuxième exécution ne concerne pas P, mais plutôt $\overline{\texttt{P}}$ (passe avant suivie de passe arrière de P). Or, les ensembles de variables lues et écrites par l'adjoint d'un (fragment de) sous-programme P sont toujours inclus dans les variables lues et écrites par P, et bien souvent strictement. Plutôt que de construire $\overline{\texttt{P}}$ et de lui appliquer l'analyse générale des variables Lues-Ecrites, ce qui ne pourrait pas prendre en compte la structure spécifique d'un code adjoint, on préfère construire une analyse spécifique de P, que l'on nommera l'analyse $\overline{\text{Lues-Ecrites}}$.

L'analyse du code mort adjoint est en fait très similaire. On remarque que la dernière instruction de P est inutile dans $\overline{\texttt{P}}$. En effet elle ne sert a aucune dérivée, et le résultat de P sera de toute manière progressivement détruit par les restaurations de valeurs intermédiaires de la passe arrière de $\overline{\texttt{P}}$. La dernière instruction de P est donc du code mort pour l'adjoint. Par extension, un certain nombre d'instructions "proches de la fin" de P sont du code mort

pour l'adjoint. Ici encore, on ne doit pas utiliser un algorithme général de détection de code mort, qui ne saurait pas utiliser la structure particulière de $\overline{\mathtt{P}}$. On doit plutôt spécifier une analyse dédiée.

Dans les deux cas, seule une implémentation à l'intérieur de TAPENADE permettra d'évaluer les bénéfices sur des applications en vraie grandeur.

4.5 Utilité du graphe des dépendances pour la DA

Plusieurs améliorations du modèle de différentiation inverse reposent sur des changements de l'ordre d'exécution des instructions. Bien souvent, ces améliorations sont apportées à la main par les utilisateurs sur les programmes différentiés, sans garantie de correction. Notre but est de fournir un cadre pour formaliser ces améliorations et trouver leurs conditions de correction, et aussi pour en proposer de nouvelles. Notre apport est d'utiliser le graphe des dépendances, qui est le support privilégié pour étudier les réordonnancements des instructions d'un programme.

Une telle amélioration consiste à détecter les instructions qui remettent à zéro une variable dérivée, et celles qui l'incrémentent. Ces instructions sont très fréquentes en mode inverse. L'étude des dépendances permet de savoir quand regrouper ces instructions. On construit alors des instructions condensées qui sollicitent moins la mémoire, évitant les défauts de cache.

On peut également considérer le mode "multi-directionnel" de la DA (tangente ou inverse). Dans ce mode, chaque instruction différentiée est incluse dans une boucle. En mode tangent, cela permet de calculer les dérivées dans plusieurs directions autour d'un même point X. Le mode multi-directionnel est donc équivalent à plusieurs exécutions du mode tangent, mais l'évaluation de la fonction f n'est faite qu'une seule fois. Nous utilisons l'analyse des dépendances pour rapprocher ces boucles dans le programme. Comme elles ont toutes le même espace d'itération (l'ensemble des directions de différentiation distinctes), on peut ensuite les fusionner, ce qui économise le coût constant de chaque boucle individuelle.

Enfin, nous montrons une propriéte importante de la passe arrière du code inverse: son graphe des dépendances entre variables dérivées est isomorphe au graphe des dépendances inversé du programme initial. La démontration repose principalement sur deux observations:

- Tout d'abord, nous savons qu'il n'y a une dépendance entre deux accès successifs à une variable `x` que si l'un ou l'autre accès est une écriture. Il n'y a pas de dépendance entre deux lectures de `x`, car on peut les

intervertir sans changer le résultat du calcul. Nous raffinons cette définition en observant que deux *incrémentations* successives de la variable `x` (c'est-à-dire `x += ...`) sont également indépendantes: les intervertir ne change pas le résultat. Petite difficulté toutefois: les incrémentations doivent être *atomiques*, sinon on risque de modifier le résultat au cas où les deux incrémentations sont faites au même instant. Mais cette condition est bien sûr vérifiée dans le cas d'une exécution séquentielle.

- Si une instruction `I` effectue une lecture de la variable `x`, son instruction adjointe $\overline{\mathtt{I}}$ effectue une incrémentation de $\overline{\mathtt{x}}$. De même, si `I` incrémente `x`, $\overline{\mathtt{I}}$ ne fait que lire $\overline{\mathtt{x}}$.

Grâce à cette symétrie entre opérations de lecture et d'incrémentation, on montre que les deux graphes de dépendances sont isomorphes. Dans le cas fréquent où certaines variables ne sont pas actives, on montre même que le graphe des dépendances de la passe arrière est plus petit que l'isomorphe du graphe des dépendances initial, ce qui est bien intéressant car les dépendances sont des contraintes qui restreignent les réordonnancements possibles.

Outre ses conséquences pour le passage à l'adjoint des propriétés de parallélisabilité d'un progamme donné, cet isomorphisme permet de montrer la correction d'une manipulation fréquente, qui consiste à permuter une boucle parallèle et l'opération de différentiation en mode inverse, comme illustré par la figure 6. On diminue ainsi la taille de la pile de mémorisation des variables

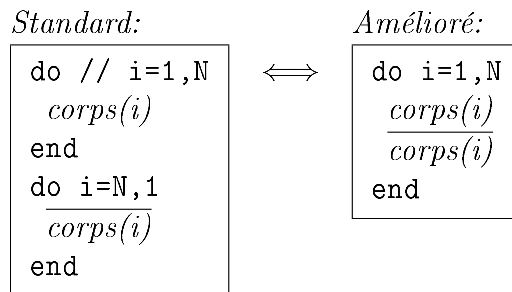

FIG. 6 – *Transformation équivalente de l'adjoint d'une boucle parallèle*

intermédiaires, d'un facteur `N` pour cette boucle.

4.6 Application: TAPENADE, un outil de Différentiation Automatique

Nous décrivons l'outil de DA TAPENADE, qui incorpore les algorithmes résultant des travaux précédents.

L'architecture de TAPENADE présente un "front-end" et un "back-end" clairement séparés de l'outil, pour favoriser l'indépendance par rapport au langage de l'application. En fait, TAPENADE accepte les programmes à différentier dans un formalisme abstrait, qui réalise l'union des constructeurs syntaxiques de FORTRAN, FORTRAN95 et C. L'extension aux langages orientés objets n'est pas encore réalisée. Au centre de l'outil se trouve un analyseur qui construit et maintient la représentation abstraite décrite plus haut, et au-dessus duquel se trouve le module qui implémente la DA. Pour l'instant, TAPENADE traite correctement tout le langage FORTRAN77 ainsi que ses extensions classiques, et l'extension à FORTRAN95 est bien avancée.

TAPENADE est le successeur d'ODYSSÉE. Son développement a commencé en 1999. TAPENADE est distribué par l'INRIA. A l'heure actuelle, il est utilisé de manière régulière sur des codes réels par des industriels tels que Dassault Aviation, CEA Cadarache, Rolls-Royce (GB), BAe (GB), Alenia (Italie). Des collègues universitaires l'utilisent pour des cours, des formations et des expériences à Grenoble, Lille, Oxford (GB), Queen's University Belfast (GB), et Argonne National Lab (USA).

TAPENADE peut être utilisé directement et sans installation depuis n'importe quel navigateur web: nous avons installé TAPENADE comme un serveur de DA à l'adresse:

`http://tapenade.inria.fr:8080/tapenade/index.jsp`

Il est aussi possible d'installer une copie locale de TAPENADE, par exemple pour l'appeler depuis un `Makefile` comme n'importe quel compilateur.

5 Conclusion

Nous continuons les recherches autour de la DA. Nous pourrions classer les pistes qui se présentent en trois catégories:

- **Calcul scientifique et numérique.** Pour développer les applications de la DA en calcul scientifique, il ne suffit pas de bien calculer des dérivées, il faut aussi définir des stratégies pour les utiliser en calcul scientifique. Il faut en fait collaborer avec les numériciens pour adapter

nos dérivées à leurs besoins, et définir de nouveaux algorithmes de part et d'autre. L'usage des gradients est appelé à se développer, avec l'essor de l'optimisation. Même des méthodes telles que les algorithmes évolutionnaires ou génétiques gagnent à interagir avec les méthodes par gradient. D'autre part, les dérivées d'ordre supérieur seront bientôt nécessaires aux algorithmes d'optimisation à venir, et leur calcul par DA doit se développer encore.

- **Algorithmes et modèles de différentiation.** C'est la partie plus spécifiquement informatique, en relation avec la parallélisation. Un des défis est de concilier les deux méthodes de restauration de variables en mode inverse, par stockage ou par recalcul, pour ensuite trouver un optimum. Il y a là des similarités avec les problèmes d'allocation de registres, que nous voulons explorer. Plus généralement, il reste encore des pistes pour améliorer ce mode inverse. Par exemple, la méthode de "checkpointing" est incontournable, mais elle manque de flexibilité quant à où l'utiliser dans le code. Les études manquent sur la DA des programmes parallèles par messages: il semble que certaines classes de programmes se différentieront facilement, d'autres pas. En particulier les messages asynchrones posent des problèmes. Les dérivées d'ordre supérieur sont un domaine de recherche encore peu exploré. Enfin, peu d'études existent sur le problème des discontinuités (non-différentiables) dues aux changements de contrôle. Le contrôle présent dans un programme rend la fonction calculée différentiable seulement par morceaux. Comment prévenir l'utilisateur final de ce risque? Nous avons commencé une étude sur ce sujet dans le cadre d'une thèse.
- **Outils de Différentiation Automatique.** L'évolution des langages d'application est rapide et difficile à suivre. Le passage de TAPENADE à C est imminent, avec le problème technique des pointeurs. A plus longue échéance, nos collègues se posent la question de la DA des programmes objet. La mise en place de directives est une autre étape importante vers un outil interactif de DA vraiment efficace.