Chapter 4

Différentiation automatique et programmation parallèle

4.1 Introduction

La simulation numérique s'attache dans un premier temps à contruire un modèle prédisant le phénomène étudié. Concentrons nous sur les calculs réalisés pour la conception d'un objet aérodynamique ou hydrodynamique (avion, plateforme). Le modèle fournit une analyse de l'écoulement. Son utilisation pour plusieurs valeurs des paramètres de conception contribue au design de l'objet. Cependant, les services d'études cherchent à identifier le design qui rendra l'objet optimal. À cet effet, on souhaite que l'outil de simulation soit capable de calculer la sensibilité de l'écoulement par rapport aux paramètres afin par exemple d'alimenter une boucle d'optimisation de type méthode de gradient. La sensibilité désigne donc ici une dérivée du champ d'écoulement. Une dérivée particulièrement importante pour les applications est l'état adjoint, utilisé dans les boucles de design optimal, d'identification et assimilation. Transformer un code de simulation en code produisant un calcul de sensibilité est un travail volumineux et délicat. Il faut typiquement considérer chaque instruction du code et développer les instructions dérivées qui contribueront au calcul de sensibilité. Depuis une vingtaine d'année des chercheurs développent des outils de Différentiation Automatique de programme (DA). Une douzaine de laboratoires tout au plus ont développé des outils généralistes de DA. L'équipe Tropics en fait partie et développe le différentiateur TAPENADE.

Dans le cadre d'ECINADS, on se propose d'étendre et consolider les capacités de TAPENADE à différentier un code combinant une gestion des allocations de mémoire complexe et des passages de message en MPI. Ces progrès sont validés en développant à l'aide de TAPENADE le logiciel adjoint du code AIRONUM (écoulement turbulent RANS-LES) afin de l'introduire dans une boucle d'optimisation du maillage de calcul. En avançant vers de la programmation de plus en plus moderne et complexe, on découvre des modes de programmation non-ambigus pour le calcul mais ambigus pour la différentiation.

Ce programme inclut donc différents types de travaux:

(a)- Expérimentation, analyse de l'action de TAPENADE sur AIRONUM.

(b)- Adaptations et extension dans TAPENADE.

(c)- Analyse et différentiation des procédures MPI.

(d)- Modifications de AIRONUM.

Le but ici est de calculer l'adjoint du code AIRONUM. Ce code de plus 60 000 lignes est écrit en Fortran 95 (utilisant de la bibliothèque MPI et l'allocation dynamique). Pour cela plusieurs options sont possibles. En effet nous pouvons utiliser les différences divisées, différentier puis discrétiser ou encore discrétiser puis différentier. L'utilisation des différences divisées se base sur l'égalité suivante:

$$f^{'}(\kappa).\delta k = \lim_{\varepsilon \to 0} \frac{f(\kappa + \varepsilon \delta\kappa) - f(\kappa)}{\varepsilon}.$$

Le problème est que nous sommes limités par la machine et le choix de ε très petit à cause des problèmes d'arrondi, notamment pour les divisions.

Les codes adjoints nécessaires à l'assimilation peuvent être écrits à la main, par discrétisation et résolution des équations adjointes. Ce processus est long, coûteux et source potentielle d'erreurs. De plus, un code adjoint étant construit par référence à un code de simulation donné, dit code initial ou " direct ", il doit être remis à jour après chaque modification ou raffinement du code direct. Cela survient fréquemment dans le cycle de vie de notre code AIRONUM, code complexe calculant des solutions d'un modèle turbulents.

Au lieu de partir de la représentation mathématique des fonctions à différencier. La D.A. utilise le programme informatique comme la base pour le calcul des dérivées. La différentiation automatique ne donne pas une approximation des dérivées, mais elle écrit un nouveau programme, qui génère les dérivées analytiques de la fonction. Elle propose deux types de calcul des dérivées : on dit que le différentiateur automatique est en mode direct pour calculer le code tangent et en mode inverse pour calculer le code adjoint. Celui qui est particulièrement intéressant pour notre problème à forte dimension d'entrée relativement au nombre de sorties. Ce qui est le cas pour le problème qui nous intéresse.

Les outils de DA existent depuis une vingtaine d'années. Cependant, le mode inverse présente des caractéristiques qui rendent particulièrement délicate la construction automatique d'un code différentié efficace. Le problème central est la reconstruction pour l'adjoint des états successifs du calcul direct, et ce dans l'ordre inverse. Les outils proposant un mode inverse efficace sont donc apparus plus tardivement. Il reste d'ailleurs un certain nombre de problèmes ouverts, objets de recherches actives. Dans cette thèse, nous avons pris en charge les travaux de type (a),(c),(d), les travaux (b) ayant été pris en charge par l'équipe de développement de TAPENADE, Valérie Pascual et Laurent Hascoët. Dans ce chapitre, nous présentons d'abord des rappels sur la DA, des éléments d'explication sur les problèmes de différentiation des allocations dynamiques, et nous présentons la méthode qui a régi la différentiation des procédures MPI.

Nous allons utiliser la différentiation automatique dans le but de calculer un adjoint qui permettra l'adaptation de maillage. En effet la métrique optimale s'exprime en fonction des états direct et adjoint:

$$\mathcal{M}_{opt} = Fonct(W_h, W_h^*)$$

L 'état adjoint W_h^* est défini par la résolution du système linéaire:

$$(\frac{\partial \Psi_h}{\partial W_h}|_{(\kappa,W_h)})^* W_h^* = (\frac{\partial J}{\partial W_h}|_{(\kappa,W_h)})^*$$

AIRONUM calcule l'état W_h, solution de: $\Psi_h(W_h) = 0$. Si $j_h(\kappa) = (g, W_h(\kappa))$ alors $W_h^* = -\frac{dj_h}{d\kappa}|_{\kappa=0}$.

4.2 Différentiation Automatique

4.2.1 Exemple de programme différentié

Nous nous intéressons à deux modes de différentiation, le mode tangent et le mode inverse. Expliquons cela à partir d'un exemple. Voici un petit programme calculant la norme d'un vecteur à deux dimensions.

```
s=a*a
s=s+b*b
s=sqrt(s)
```

Le code differentié en mode tangent est le suivant. Nous pouvons construire son programme dérivé qui calcule sa différentielle c'est à dire qui calcule pour une petite variation donnée de ses entrées a et b la petite variation au premier ordre de sa sortie s, c'est le mode "tangent".

$$\begin{aligned}
&\mathtt{s} = 2\mathtt{a}\dot{\mathtt{a}}\\
&\mathtt{s} = \mathtt{a} * \mathtt{a}\\
&\dot{\mathtt{s}} = \dot{s} + 2b\dot{b}\\
&\mathtt{s} = \mathtt{s} + \mathtt{b} * \mathtt{b}\\
&\mathtt{s} = \dot{\mathtt{s}}/2\sqrt{\mathtt{s}}
\end{aligned}$$

Chaque instruction est accompagné d'une nouvelle intruction dite dérivée qui propage la différentielle ($\dot{\mathtt{a}}, \dot{\mathtt{b}}$ *et* $\dot{\mathtt{s}}$)

Nous pouvons aussi construire la dérivée en mode "inverse", qui calcule le gradient du résultat r par rapport à l'ensemble des entrées(a et b), par une propagation à rebours des gradients partiels.

$$\begin{aligned}
&\mathtt{s} = \mathtt{a} * \mathtt{a}\\
&\mathtt{s} = \mathtt{s} + \mathtt{b} * \mathtt{b}\\
&\bar{\mathtt{s}} = \bar{\mathtt{s}}/2\sqrt{\mathtt{s}}\\
&\bar{\mathtt{b}} = \bar{\mathtt{b}} + 2\mathtt{b}\bar{\mathtt{s}}\\
&\bar{\mathtt{a}} = \bar{\mathtt{a}} + 2\mathtt{a}\bar{\mathtt{s}}
\end{aligned}$$

Cette fois-ci, les instructions dérivées sont regroupées à la fin du programme, et suivent une copie du programme initial. De plus les instructions dérivées apparaissent dans l'ordre inverse. L'avantage majeur du mode inverse visible sur cet exemple, est que l'on obtient les gradients $\bar{a}$ et $\bar{b}$ en une seule exécution du programme dérivé inverse (coût une multiplication, une division et une racine carrée).

Si au contraire nous avions utilisé le programme dérivée tangent, nous aurions du appeler une fois pour chaque composante de l'espace d'entrée (a et b) pour un coût total de 4 multiplications, 2 divisions et deux racines carrées.

4.2.2 Principe de la D.A.

Un programme P est une suite d'instructions :

$$\{I_1; I_2; \cdots ; I_p; \}$$

Chaque instruction I_i est une transcription d'une fonction mathématique $f_i : X_{i-1} \mapsto X_i$. Le programme P a donc une transcription mathématique $F : X \mapsto Y$ qui est la composée de fonctions :

$$F = f_p \circ f_{p-1} \circ \cdots \circ f_1$$

où $X \in I\!R^n$ et $Y \in I\!R^m$.

La caractéristique essentielle des langages de programmation impératifs classiques (Fortran, C,C++,Java, etc.) est qu'ils autorisent la modification, la réécriture de la valeur contenue dans une variable informatique. Les variables informatiques sont donc différentes des variables mathématiques, dont la valeur est fixée, déterminée, par les équations qui s'y rapportent. Pour relier les deux notions, nous dirons qu'une variable informatique est un réceptacle pouvant contenir successivement plusieurs variables mathématiques, au cours de l'exécution du programme. Considérons un programme donné sous la forme générale suivante :

```
Initialiser V avec X
(I_1)  V := f_1(V)
...
(I_k)  V := f_k(V)
...
(I_p)  V := f_p(V)
Recuperer Y dans V
```

Dans laquelle V est la collection de toutes les variables informatiques individuelles et les f_i sont des opérations de base du langage utilisé. Comme la variable V contient successivement les variables mathématiques X_1 à X_p, ce programme a exactement comme transcription mathématique l'équation :

$$F = f_p \circ f_{p-1} \circ \cdots \circ f_1$$

Ayant à notre disposition une fonction mathématique, nous pouvons maintenant utiliser la règle de dérivation pour les fonctions composées qui donne :

$$F^{'}(X) = f_p^{'}(X_{p-1}) \times f_{p-1}^{'}(X_{p-2}) \times \cdots \times f_1^{'}(X_0)$$

qui est l'expression mathématique de la dérivée désirée. Toute la difficulté est maintenant de "remonter" au niveau des programmes, c'est à dire d'écrire un programme informatique $P^{'}$ qui implémentera la fonction mathématique $F^{'}$.

Il est possible d'adapter l'algorithme P tel qu'il calcule $F^{'}(X)$ en plus de $F(X)$. Cela peut être fait en étendant l'instruction I_1 qui calcule $X_1 = f_1(X_0)$ avec un morceau de code qui calcule la jacobienne $J_1 = f_1^{'}(X_0) \times Id$. En faisant cela pour toutes les instructions I_k où on ajoute $J_k = F_k^{'} \times J_{k-1}$. Cette transformation est locale à chaque instruction I_k.

Etudions les deux modes de différentiations. Commencons par le mode tangent. Il s'agit d'évaluer $\dot{Y} = F^{'}(X) \times \dot{X}$ c'est à dire la dérivée de F le long de la direction $\dot{X}$.

$$F^{'}(X) \times \dot{X} = f_p^{'}(X_{p-1}) \times f_{p-1}^{'}(X_{p-2}) \times \cdots \times f_1^{'}(X_0) \times \dot{X}$$

Cette formule est équivalente à la suivante, qui a l'avantage d'être décomposée en étapes explicites :

$$\begin{aligned}
&X_0 = X \\
&\dot{X}_0 = \dot{X} \\
&X_1 = f_1(X_0) \\
&\dot{X}_1 = f_1^{'}(X_0) \times \dot{X}_0 \\
&\cdots \\
&X_k = f_k(X_{k-1}) \\
&\dot{X}_k = f_k^{'}(X_{k-1}) \times \dot{X}_{k-1} \\
&\cdots \\
&X_p = f_p(X_{p-1}) \\
&\dot{X}_p = f_p^{'}(X_{p-1}) \times \dot{X}_{p-1} \\
&Y = X_p \\
&\dot{Y} = \dot{X}_p
\end{aligned}$$

Il est facile de retranscrire cette formule sous la forme d'un programme. De même que les variables informatiques V contiennent successivement les variables mathématiques X_i; les nouvelles variables informatiques $\dot{V}$ contiennent successivement les variables mathématiques:

Initialiser V avec X et $\dot{V}$ avec $\dot{X}$
($\dot{\mathtt{I}}_1$) $\dot{\mathtt{V}} := f_1^{'}(\mathtt{V}) \times \dot{\mathtt{V}}$
($\mathtt{I}_1$) $\mathtt{V} := \mathtt{f}_1(\mathtt{V})$
$\cdots$
($\dot{\mathtt{I}}_k$) $\dot{\mathtt{V}} := \mathtt{f}_k^{'}(\mathtt{V}) \times \dot{\mathtt{V}}$
($\mathtt{I}_k$) $\mathtt{V} := \mathtt{f}_k(\mathtt{V})$
$\cdots$
($\dot{\mathtt{I}}_p$) $\dot{\mathtt{V}} := \mathtt{f}_p^{'}(\mathtt{V}) \times \dot{\mathtt{V}}$
($\mathtt{I}_p$) $\mathtt{V} := \mathtt{f}_p(\mathtt{V})$
Recuperer Y dans V et $\dot{\mathtt{Y}}$ dans $\dot{\mathtt{V}}$

Remarquons que chaque déclaration $\dot{I}_k$ précède I_k, parce que I_k écrase l'ancienne valeur de V.

Le mode adjoint consiste à évaluer le produit matrice vecteur $\bar{X} = \bar{Y} \times F^{'}(X)$. Ce qui se developpe:

$$\bar{X} = \bar{Y} \times f_p^{'}(X_{p-1}) \times f_{p-1}^{'}(X_{p-2}) \times \cdots \times f_1^{'}(X_0)$$

$\bar{Y}$ est un vecteur, cette formule se décompose de la manière suivante toujours explicite:

$$
\begin{aligned}
&X_0 = X \\
&X_1 = f_1(X_0) \\
&\dots \\
&X_k = f_k(X_{k-1}) \\
&\dots \\
&X_p = f_p(X_{p-1}) \\
&Y = X_p \\
&\bar{X_p} = \bar{Y} \\
&\bar{X_{p-1}} = \bar{X_p} \times f_p^{'}(X_{p-1}) \\
&\dots \\
&\bar{X_0} = \bar{X_1} \times f_1^{'}(X_0) \\
&\bar{X} = \bar{X_0}
\end{aligned}
$$

Ici la transcription en programme est délicate car l'ordre de calcul des $\bar{X_k}$ est décroissant. Nous devons d'abord calculer $\bar{X}_{p-1}$ ce qui conduit à utiliser la valeur de X_{p-1}. Par conséquent, remplacer toutes les X_i par V et les $\bar{X_i}$ par V dans la formule mathématiques conduit à un programme faux. Pour réaliser cette "inversion du flot de données", plusieurs méthodes peuvent être employées. Nous pouvons d'une part recalculer chaque X_i juste avant son utilisation dans $f_p(X_{p-1})$, en recopiant les instructions d'origine I_1 à I_p

Cette méthode s'appelle le "recompute-all", elle a le désavantage de répéter plusieurs fois les mêmes calculs. Ce qui rend le programme moins performant.

Une autre méthode consiste à ne pas recalculer les X_k mais de plutôt les mémoriser dans une pile . Ce qui conduit au programme suivant :

```
Initialise V avec X et V̄ avec Ȳ
push(V))
(I_1) V := f_1(V)
...
push(V)
(I_k) V := f_k(V)
...
push(V)
(I_{p-1}) V := f_{p-1}(V)
pop(V)
(Ī_p) V̄ := V̄ × f'_{p-1}(V)
...
pop(V)
(Ī_k) V̄ := V̄ × f'_k(V)
...
pop(V)
(Ī_1) V̄ := V̄ × f'_1(V)
Recuperer X̄ dans V̄
```

On reconnaît les deux phases successives que l'on avait vues sur l'ensemble de la section 2.1. La première phase("passe avant") calculeX_i et les stocke sur la pile. La deuxième phase("passe arrière") calcule les dérivées en sens inverse, en récupérant les X_i depuis la pile au fur et à mesure des besoins.

4.3 Différentiation d'un programme contenant des allocations dynamiques

Nous présentons les problèmes rencontrés dans un programme contenant des allocations dynamiques. Nous illustrerons chaque problème par exemple, ce qui amènera à une solution qui est différente suivant que nous sommes dans le cas inverse ou direct.

4.3.1 Gestion des allocations dans le Mode tangent

Présentons un programme contenant l'allocation d'une variable dynamique V qui est déalloué à la fin du programme.

```
Declarer V
(A1)  allocate(V)
Initialiser V avec X
(I1)  V := f1(V)
...
(Ik)  V := fk(V)
...
(Ip)  V := fp(V)
Recuperer Y dans V
(d1)  Deallocate(V)
```

Le code tangent de ce programme conformément aux explications sur le principe général de la D.A contient des variables dérivées. La variable dérivée de V est notée $\dot{V}$, étant donné qu'elle contiennent des valeurs de variables mathématiques appartenant au même espace. Il est logique que $\dot{V}$ soit aussi une variable dynamique. Il faut donc maintenant savoir où placer les instructions d'allocation et de déallocation de $\dot{V}$. La réponse est qu'il faut placer la déclaration l'allocation et la déallocation de la variable $\dot{V}$ au même endroit que pour la variable V. Pour l'exemple précédent nous obtenons le code tangent suivant:

```
     Declarer V
     Declarer V̇
(A1) allocate(V)
(Ȧ1) allocate(V̇)
Initialiser V avec X et V̇ avec Ẋ
(İ1) V̇ := f1'(V) × V̇
(I1) V := f1(V)
...
(İk) V̇:= fk'(V) × V̇
(Ik) V := fk(V)
...
(İp) V̇:= fp'(V)×V̇
(Ip) V := fp(V)
     Recuperer Y dans V et Ẏ dans V̇
(D1)  deallocate(V)
(Ḋ1)  deallocate(V̇)
```

Nous pouvons remarquer que la résolution de ce problème est assez similaire au principe général de la DA en mode tangent, puisqu'on ajoute à chaque instruction de déclaration, d'allocation et une déallocation pour la variable dérivée la même instruction mais cette fois pour la variable dérivée.

Prenons maintenant un exemple où après la déallocation d'une variable V nous re-allouons une seconde fois après l'avoir au préalable déalloué :

```
Declarer V
allocate(V)
Initialiser V avec X
(I1) V := f1(V)
...
Deallocate(V)
allocate(V)
(Ik) V := fk(V)
...
(Ip) V := fp(V)
Recuperer Y dans V
Deallocate(V)
```

Pour le cas du mode tangent il suffit d'employer la même technique.
C'est à dire déclarer, allouer et déallouer V et $\dot{V}$. Ainsi nous obtenons ceci :

```
Declarer V
Declarer V̇
allocate(V)
allocate(V̇)
Initialiser V avec X et V̇ avec Ẋ
(İ1)  V̇ := f1'(V) × V̇
(I1)  V := f1(V)
...
deallocate(V)
deallocate(V̇)
allocate(V)
allocate(V̇)

(İk)  V̇:= fk'(V) × V̇
(Ik)  V := fk(V)
...
(İp)  V̇:= fp'(V)×V̇
(Ip)  V := fp(V)
Recuperer Y dans V et  Ẏ dans V̇
deallocate(V)
deallocate(V̇)
```

Nous pouvons voir que dans pour le cas du mode tangent, le problème se résout de la manière suivante :

- Pour la déclaration d'une variable V, on ajoute dans le code dérivé une déclaration de la variable $\dot{V}$.

- Pour l'allocation d'une variable V, on ajoute dans le code dérivé une allocation de la variable $\dot{V}$.

- Pour la déallocationd'une variable V, on ajoute dans le code dérivé une déallocation de la variable $\dot{V}$.

4.3.2 Gestion des allocations dans le Mode inverse

La gestion des allocations dynamiques dans le cas inverse est plus compliqué car une variable V déallouée est perdue. Elle ne peut plus être récupérée lors d'un $pop(V)$. Il faut donc toujours s'assurer qu'une variable n'est pas déallouée avant d'être enlevée de la pile. Cette recherche oblige à une analyse plus fine.

Reprenons le même exemple que précedemment et donnons le code différentié approprié.

```
Declarer V
allocate(V)
Initialiser V avec X
(I_1)  V := f_1(V)
...
deallocate(V)
allocate(V)
(I_k)  V := f_k(V)
...
(I_p)  V := f_p(V)
Recuperer Y dans V
Deallocate(V)
```

Le code en mode inverse est similaire à celui du tangent:

```
Declarer V
Declarer V̄
allocate(V)
allocate(V̇)
Initialise V avec X et  V̄ avec Ȳ
push(V)
(I_1) V := f_1(V)
...
push(V)
(I_k) V := f_k(V)
...
push(V)
(I_{p-1}) V := f_{p-1}(V)
pop(V)
(Ī_p) V̄ := V̄ × f'_p(V)
...
pop(V)
(Ī_k) V̄ := V̄ × f'_k(V)
...
Mettre dans V la valeur de X
(Ī_1) V̄ := V̄ × f'_1(V)
Recuperer X̄ de V̄
deallocate(V)
deallocate(V̄)
```

Reprenons le deuxième exemple.

```
Declarer V
Declarer V̄
allocate(V)
allocate(V̇)
Initialise V avec X et V̄ avec Ȳ
push(V)
(I_1) V := f_1(V)
...
push(V)
deallocate(V)
allocate(V)
(I_k) V := f_k(V)
...
push(V)
(I_{p-1}) V := f_{p-1}(V)
pop(V)
(Ī_p) V̄ := V̄ × f'_p(V)
...
pop(V)
(Ī_k) V̄ := V̄ × f'_k(V)
deallocate(V̄)
allocate(V̄)
...
Mettre dans V la valeur de X
(Ī_1) V̄ := V̄ × f'_1(V)
Recuperer X̄ de V̄
deallocate(V)
deallocate(V̄)
```

L'exemple ci-dessus est faux car on déalloue la variable V lors de la passe avant. Nous sommes donc dans l'impossibilité de récupérer sa valeur dans la passe arrière lors d'un pop puisqu'on a avec la déallocation supprimé la valeur dans la mémoire. Pour le moment nous n'avons encore résolu ce problème délicat, ce travail est en cours de recherche.

4.4 Différentiation d'un code MPI

Dans cette partie nous verrons comment différentier un code en mode inverse. Pour cela dans un premier temps nous décrirons un code MPI différentié, afin d'introduire la démarche utilisée pour différentier le code en mode inverse. Dans un deux temps nous expliquerons la méthode de différentiation des codes MPI. Et dans un troisième temps nous expliquons comment résoudre un problème provenant d'un flot de données non déterministe.

4.4.1 Exemple d'un programme MPI différentié

Prenons un code MPI et différentions le en mode inverse. Dans note exemple le processus maitre calcule $\mathtt{V} := \mathtt{f_1}(\mathtt{V}); \cdots \mathtt{V} = \mathtt{f_n}(\mathtt{V})$ puis diffuse le résultat de la valeur

de la variable V aux autres processus par l'intermédiaire de **MPI_bcast**. À l'aide de la valeur V les processus calculent $[\mathtt{V} := \mathtt{f_{n+2}}(\mathtt{V}); \cdots \mathtt{V} = \mathtt{f_p}(\mathtt{V})]$ et récupèrent le résultat dans la variable Y.

```
Initialise V avec X
if (processeur maitre)
V := f_1(V)
...
V = f_n(V)
endif
call mpi_bcast(V, processeurmaitre)
V = f_{n+2}(V)
...
V = f_p(V)
Recuperer la variable Y la valeur de V
```

Voici le code différentié adjoint de ce programme, utilisant une approche "store-all" pour l'inversion du flot de données.

```
Initialise V avec X et  V̄ avec Ȳ
if (processeurmaitre)
V := f_1(V)
push(V)
...
push(V)
V = f_n(V) endif
call mpi_bcast(V, processeurmaitre)
push(V)
V = f_{n+2}(V)
...
push(V)
V = f_p(V)
pop(V)
V̄ = V̄ × f'_p(V)
pop(V)
...
V̄ = V̄ × f'_{n+2}(V)
call mpi_reduce( tm̄p, V̄, SUM)
V̄ = tm̄p
pop(V)
if(processeurmaitre)
pop(v)
V̄ = V̄ × f'_n(V)
...
pop(V)
V̄ = V̄ × f'_1(V)
end if
Recupere X̄ de V̄
```

Nous reconnaissons sur code adjoint l'architecture déjà décrite, et en particulier les deux passes successives, avant et arrière. La passe arrière effectue les opérations différentiées des instructions d'origine dans l'ordre inverse. Pour que ce programme soit correct, l'instruction différentiée de l'appel"mpi_bcast" doit être un "mpi_reduce". Dans la suite, nous justifions ce choix et nous proposons une démarche systématique pour trouver les instructions différentiées adjointes des principales primitives de communication MPI.

4.4.2 Trouver l'adjoint d'une routine MPI

Précédemment, nous avons vu le principe général de la D.A où chaque instruction I_i est retranscrite en fonction mathématique f_i, puis est dérivée en $f_i^{'}$ et retranscrit en instruction informatique $I_i^{'}$.

Étudions maintenant le cas d'un code utilisant la librairie MPI. Notons I_{Mi} une instruction contenant des primitives MPI. Cette instruction I_{Mi} ne peut pas être retranscrite en fonction mathématique dans la plupart des cas, car les routines MPI représentent des communications entre processus.

La démarche que nous utilisons ici est différente du principe général. Dans un premier temps nous séquentialiserons I_{Mi} en I_{Si}. Dans un deuxième temps nous différentions en mode inverse I_{Si} pour obtenir $\bar{I}_{Si}$. Enfin nous proposerons un nouveau code parallèle $\bar{I}_{Mi}$ dont la séquentialisation correspond à $\bar{I}_{Si}$

Plus précisement pour chaque primitive MPI MPI_fPD (fPD=**f**onction **p**arallèle en mode **d**irecte) est transformée en fonction sequentielle SEQ_fSD (fSD=**f**onction **s**équentielle en mode **d**irecte) .
SEQ_fSD est différencié en mode inverse pour obtenir SEQ_fSA (fSR=**f**onction **s**équentielle en mode **a**djoint) .
Enfin SEQ_fSA est mise sous une forme parallèle : MPI_fPA (fPA=**f**onction **p**arallèle en mode **A**djoint).

La démarche se schématise ainsi :

$$\begin{array}{ccc} MPI\_fPD & & MPI\_fPA \\ \downarrow & & \uparrow \\ SEQ\_fSD & \Longrightarrow & SEQ\_fSA \end{array}$$

La première étape est de séquentialiser toutes les routines MPI nous obtenons N routines :

$$SEQ\_fSD_1 \dots SEQ\_fSD_N$$

Puis en passant au mode adjoint nous obtenons :

$$SEQ\_fSA_1 \dots SEQ\_fSA_N$$

Si $SEQ\_fSD_k = SEQ\_fSA_l$ alors MPI_fPD_k a pour adjoint MPI_fPA_l. Soit la routine MPI_fPD_k, nous calculons SEQ_fSA_k puis nous cherchons dans

les fonctions sequentielles sa correspondance SEQ_fSD_l, son adjoint est alors MPI_fPA_l.

Notre méthode de séquentialisation, empirique, est similaire à l'approche "Partitioned Global Address Space" PGAS utilisée pour les preuves de propriétés de codes parallèles, voir [2, 4, 3]. Un ensemble de process exécutent en parallèles des portions de code (2 ou 3 dans notre cas pour alléger les notations). En distinguant les variables de même nom sur des process différents au moyen d'un indice, et en introduisant des variables globales pour représenter les canaux de communication, on peut écrire un programme séquentiel équivalent. Dans le cas de deux processeurs nous pouvons schématiser la situation en

$$\begin{array}{c|c} P_1 & P_2 \\ x = 12 & x = 0 \end{array}$$

Dans l'écriture séquentielle de ce code nous distinguons la variable suivant qu'elle soit dans le processeur 1 ou le processeur 2. Dans le cas précédent x_i représenterait la variable x dans le processeur i, cela donne alors le code séquentiel suivant :

$$\begin{array}{c} x_1 = 12 \\ x_2 = 0 \end{array}$$

En parallèle notre nouvelle représentation donne

$$\begin{array}{c|c} P_1 & P_2 \\ x_1 = 12 & x_2 = 0 \end{array}$$

Nous allons maintenant introduire une variable globale fictive l_i qui représente le canal de communication du processeur i avec les autres processeurs. Nous obtenons :

$$\begin{array}{c|c} P_1 & P_2 \\ x_1 = 12 & x_2 = 0 \\ l_1 = x_1 & \end{array}$$

Ce cheminement explique l'étape $MPI\_fPD \rightarrow SEQ\_fSD$. Pour passer à l'étape $SEQ\_fSD \Rightarrow SEQ\_fSA$ nous différentions.

Calculons maintenant l'adjoint et des routines parallèles classiques.

Le couple MPI_SEND/MPI_RECV Pour passer du parallèle au séquentiel, nous distinguons les variables suivant le processus donné. Par exemple supposons que nous somme en SPMD (Single Program, Multiple Data) et que nous avons l'instruction suivante :

```
if (P1) then
mpi_recv(x, P2)
else
mpi_recv(x, P1)
```

c'est à dire le couple : mpi_send(y)/ MPI_recv(y). Pour des raisons de clarté considérons juste les paramètres essentiels à la compréhension du passage en mode reverse. Supposons que nous avons maintenant deux processeurs :

$$\begin{array}{c|c} P_1 & P_2 \\ l = x_1 & y_2 = l \\ & \end{array}$$

Ce qui donne séquentiellement

$$\begin{array}{c} l = x_1 \\ y_2 = l \end{array}$$

Le code inverse donne

$$\begin{array}{c} \bar{l} = 0 \\ \bar{l} = \bar{l} + \bar{y_2} \\ \bar{y_1} = 0 \\ \bar{x_1} = \bar{x_1} + \bar{l} \\ \bar{l} = 0 \end{array}$$

Ce qui donne :

$$\begin{array}{c|c} P_1 & P_2 \\ & \bar{l} = \bar{y_2} \\ \bar{x_1} = \bar{x_1} + \bar{l} & \bar{y_2} = 0 \end{array}$$

Collecte de données réparties : MPI_GATHER Cette fonction permet au processus récepteur de collecter les données provenant de tous les processus. Dans notre exemple le processus P_1 collecte les données x_2 et x_3 provenant des processus P_2 et P_3.

Nous simplifions l'écriture des paramètres de la primitive MPI_GATHER,pour des raisons de clarté considérons juste les paramètres essentiels à la compréhension du passage en mode reverse ; nous noterons MPI_GATHER(x,y,P_1), x représente l'entrée, y la sortie et P_1 le processeur récepteur :
Le code en parallèle est traduit de la manière suivant :

$$\begin{array}{c|c|c} P_1 & P_2 & P_3 \\ l_1 = x_1 & l_2 = x_2 & l_3 = x_3 \\ y_1[1] = l_1 & & \\ y_1[2] = l_2 & & \\ y_1[3] = l_3 & & \end{array}$$

Qui se traduit séquentiellement :

$$
\begin{aligned}
l_1 &= x_1 \\
l_2 &= x_2 \\
l_3 &= x_3 \\
y_1[1] &= l_1 \\
y_1[2] &= l_2 \\
y_1[2] &= l_2
\end{aligned}
$$

Nous pouvons alors appliquer le mode inverse :

$$
\begin{aligned}
\bar{l_3} &= \bar{y_1}[3] \\
\bar{l_2} &= \bar{y_1}[2] \\
\bar{l_1} &= \bar{y_1}[1] \\
\bar{y_1}[:] &= 0 \\
\bar{x_3} &= \bar{x_3} + \bar{l_3} \\
\bar{x_2} &= \bar{x_2} + \bar{l_2} \\
\bar{x_1} &= \bar{x_1} + \bar{l_1}
\end{aligned}
$$

Ce qui correspond au code parallèle :

P_1	P_2	P_3
$\bar{l_1} = \bar{y_1}[1]$	$\bar{l_2} = \bar{y_1}[2]$	$\bar{l_3} = \bar{y_1}[3]$
$\bar{y_1}[:] = 0$	$\bar{y_1}[:] = 0$	$\bar{y_1}[:] = 0$
$\bar{x_1} = \bar{x_1} + \bar{l_1}$	$\bar{x_2} = \bar{x_2} + \bar{l_2}$	$\bar{x_3} = \bar{x_3} + \bar{l_3}$

Diffusion sélective de donnée réparties : MPI_SCATTER Cette fonction permet à un processus de diffuser des données aux processus indiqués. Dans notre exemple le processus P_1 diffuse ses données à tous les processus (P_2 et P_3). mpi_scatter(x,y,P_1) est une écriture simplifiée, x représente le paramètre d'entrée, y le paramètre de sorties et P_1 est le processus qui diffuse ses données. Le code en parallèle est traduit de la manière suivant :

P_1	P_2	P_3
$l_1 = x_1[1]$		
$l_2 = x_1[2]$		
$l_3 = x_1[3]$		
$y_1 = l_1$	$y_2 = l_2$	$y_2 = l_2$

Ce qui donne en séquentiel

$$
\begin{aligned}
&P_1 \\
l_1 &= x_1[1] \\
l_2 &= x_2[1] \\
l_3 &= x_3[1] \\
y_1 &= l_1 \\
y_2 &= l_2 \\
y_3 &= l_3
\end{aligned}
$$

Le mode inverse donne

$$
\begin{array}{c}
P_1 \\
\bar{l_3} = \bar{y_3} \\
\bar{l_2} = \bar{y_2} \\
\bar{l_1} = \bar{y_1} \\
y[:] = 0 \\
\bar{x_1}[3] = \bar{x_1}[3] + \bar{l_3} \\
\bar{x_1}[2] = \bar{x_1}[2] + \bar{l_2} \\
\bar{x_1}[1] = \bar{x_1}[1] + \bar{l_1}
\end{array}
$$

Ce qui donne

$$
\begin{array}{c|c|c}
P_1 & P_2 & P_3 \\
\bar{l_1} = \bar{y_1} & \bar{l_2} = \bar{y_2} & \bar{l_3} = \bar{y_3} \\
\bar{y_1} = 0 & \bar{y_2} = 0 & \bar{y_3} = 0 \\
\bar{x_1}[3] = \bar{x_1}[3] + \bar{l_3} & & \\
\bar{x_1}[2] = \bar{x_1}[2] + \bar{l_2} & & \\
\bar{x_1}[1] = \bar{x_1}[1] + \bar{l_1} & &
\end{array}
$$

Diffusion générale : MPI_BCAST Cette fonction permet à un processus de diffuser un message à tous les autres processus. Dans notre exemple le processus P_1 envoie y aux processus P_2 et P_3.

Dans la primitive MPI_BCAST(y,P_1), y représente le paramètre d'entrée et de sortie, P_1 est le processus qui envoie à tous les autres processus.

$$
\begin{array}{c|c|c}
P_1 & P_2 & P_3 \\
l_1 = y_1 & & \\
 & y_2 = l_1 & y_3 = l_1
\end{array}
$$

Ce qui se traduit séquentiellement par :

$$
\begin{array}{c}
l_1 = y_1 \\
y_2 = l_1 \\
y_3 = l_1
\end{array}
$$

Le code inverse donne

$$
\begin{array}{c}
\bar{l_1} = \bar{y_3} \\
\bar{y_3} = 0 \\
\bar{l_1} = \bar{l_1} + \bar{y_2} \\
\bar{y_2} = 0 \\
\bar{y_1} = \bar{y_1} + \bar{l_1}
\end{array}
$$

ce qui donne

$$\begin{array}{c|c|c} P_1 & P_2 & P_3 \\ & & \bar{l_1} = \bar{y_3} \\ & & \bar{y_3} = 0 \\ & \bar{l_1} = \bar{l_1} + \bar{y_2} & \\ & \bar{y_2} = 0 & \\ & & \\ \bar{y_1} = \bar{y_1} + \bar{l_1} & & \end{array}$$

MPI_REDUCE Cette fonction permet de faire des opération de réduction en plus des communications collectives. Le résultat de l'opération est récupéré sur un seul processus. Dans notre exemple L'opération considérée est une somme qui est récupéré par le processus P_1.

On écrira MPI_ALL_REDUCE(x,y,SUM,P_1) où y est la valeur reçcue, x est la valeur de sortie et SUM l'opération que l'on considère. Supposons que nous travaillons sur 3 processeurs P_1 P_2 *et* P_3. Le calcul fait sur chaque processeur est le suivant:

$$\begin{array}{c|c|c} P_1 & P_2 & P_3 \\ l_1 = x_1 & l_2 = x_2 & l_3 = x_3 \\ y_1 = l_1 + l_2 + l_3 & & \end{array}$$

Ce qui donne séquentiellement :

$$\begin{aligned} l_1 &= x_1 \\ l_2 &= x_2 \\ l_3 &= x_3 \\ y_1 &= l_1 + l_2 + l_3 \end{aligned}$$

Nous obtenons le code inverse

$$\begin{aligned} \bar{l_1} &= \bar{y_1} \\ \bar{l_2} &= \bar{y_1} \\ \bar{l_3} &= \bar{y_1} \\ \bar{y_1} &= 0 \\ \bar{x_3} &= \bar{x_3} + \bar{l_3} \\ \bar{x_2} &= \bar{x_2} + \bar{l_2} \\ \bar{x_1} &= \bar{x_1} + \bar{l_1} \end{aligned}$$

Qui se traduit sous forme parallèle :

$$\begin{array}{c|c|c} P_1 & P_2 & P_3 \\ \bar{l_1} = \bar{y_1} & & \\ \bar{l_2} = \bar{y_1} & & \\ \bar{y_1} = 0 & & \\ \bar{l_3} = \bar{y_1} & & \\ \bar{x_1} = \bar{x_1} + \bar{l_1} & \bar{x_2} = \bar{x_2} + \bar{l_2} & \bar{x_3} = \bar{x_3} + \bar{l_3} \end{array}$$

MPI_ALLREDUCE Cette fonction fait la même chose que *mpi_reduce* à la différence que le résultat de l'opération est récupéré par tous les processus.

On écrira MPI_ALL_REDUCE(x,y,SUM,P_1) où y est la valeur reçu, x est la valeur de sortie et SUM l'opération que l'on considère. Supposons que nous travaillons sur 3 processeurs P_1 P_2 *et* P_3. Le calcul fait sur chaque processeur est le suivant:

P_1	P_2	P_3
$l_1 = x_1$	$l_2 = x_2$	$l_3 = x_3$
$y_1 = l_1 + l_2 + l_3$	$y_2 = l_1 + l_2 + l_3$	$y_3 = l_1 + l_2 + l_3$

Le code précédent peut être représenté séquentiellement de la manière suivante:

$$
\begin{aligned}
l_1 &= x_1 \\
l_2 &= x_2 \\
l_3 &= x_3 \\
y_1 &= l_1 + l_2 + l_3 \\
y_2 &= l_1 + l_2 + l_3 \\
y_3 &= l_1 + l_2 + l_3
\end{aligned}
$$

Nous sommes en mesure maintenant de donner le code différentié inverse du problème. où, après avoir remarqué que les valeurs de $\bar{l_1}$, $\bar{l_2}$, et $\bar{l_3}$ sont identiques:

$$
\begin{aligned}
\bar{l_3} &= \bar{y_3} \\
\bar{y_3} &= 0 \\
\bar{l_2} &= \bar{y_2} \\
\bar{y_2} &= 0 \\
\bar{l_1} &= \bar{y_1} \\
\bar{y_1} &= 0 \\
\bar{x_1} &= \bar{x_1} + \bar{l_1} + \bar{l_2} + \bar{l_3} \\
\bar{x_2} &= \bar{x_2} + \bar{l_1} + \bar{l_2} + \bar{l_3} \\
\bar{x_3} &= \bar{x_3} + \bar{l_1} + \bar{l_2} + \bar{l_3}
\end{aligned}
$$

Ce qui donne en parallèle :

P_1	P_2	P_3
$\bar{l_1} = \bar{y_1}$	$\bar{l_2} = \bar{y_2}$	$\bar{l_3} = \bar{y_3}$
$\bar{y_1} = 0$	$\bar{y_2} = 0$	$\bar{y_3} = 0$
$\bar{tmp} = \bar{l_1} + \bar{l_2} + \bar{l3}$	$\bar{tmp} = \bar{l_1} + \bar{l_2} + \bar{l3}$	$\bar{tmp} = \bar{l_1} + \bar{l_2} + \bar{l3}$
$\bar{x_1} = \bar{x_1} + \bar{tmp}$	$\bar{x_2} = \bar{x_2} + \bar{tmp}$	$\bar{x_3} = \bar{x_3} + \bar{tmp}$

Détermination des adjoints Si nous nous intéressons à MPI_REDUCE son adjoint séquentiel est :

$$
\begin{aligned}
\bar{l_1} &= \bar{l_1} + \bar{y_1} \\
\bar{l_2} &= \bar{l_2} + \bar{y_1} \\
\bar{l_3} &= \bar{l_3} + \bar{y_1} \\
\bar{y_1} &= 0 \\
\bar{x_3} &= \bar{x_3} + \bar{l_3} \\
\bar{x_2} &= \bar{x_2} + \bar{l_2} \\
\bar{x_1} &= \bar{x_1} + \bar{l_1}
\end{aligned}
$$

Qui se traduit par

P_1	P_2	P_3
$\bar{l_1} = \bar{l_1} + \bar{y_1}$	$\bar{l_2} = \bar{l_2} + \bar{y_1}$	$\bar{l_3} = \bar{l_3} + \bar{y_1}$
$\bar{y_1} = 0$	$\bar{y_1} = 0$	$\bar{y_1} = 0$
$\bar{x_1} = \bar{x_1} + \bar{l_1}$	$\bar{x_2} = \bar{x_2} + \bar{l_2}$	$\bar{x_3} = \bar{x_3} + \bar{l_3}$

P_1	P_2	P_3
$l_1 = x_1$		
$y_1 = l_1$	$y_2 = l_1$	$y_3 = l_1$

Qui peut s'écrire sous la forme

$$
\begin{aligned}
l_1 &= x_1 \\
y_1 &= l_1 \\
y_2 &= l_1 \\
y_3 &= l_1
\end{aligned}
$$

Si nous regardons les routines directes séquentielles nous pouvons remarquer que ces deux fonctions avec les mêmes valeurs en paramètres d'entrée donnent les même résultats dans les paramètres de sortie. On a donc que l'adjoint de MPI_REDUCE est MPI_BCAST.

Ecrivons maintenant nos résultats en terme de pseudo code dans le tableau

récapitulatif suivant.

$MPI\_GATHER(x, y, P_1)$	$MPI\_SCATTER(\bar{tmp}, \bar{y}, P_1)$ $if \ \ (P1) \ \ \bar{y_1}[:] = 0$ $\bar{x} = \bar{x} + \bar{tmp}$
$MPI\_SCATTER(x, y, P_1)$	$MPI\_GATHER(\bar{tmp}, \bar{y}, P_1)$ $\bar{y} = 0$ $\bar{x} = \bar{x} + \bar{tmp}$
$MPI\_BCAST(y, P_1)$	$MPI\_REDUCE(\bar{temp}, \bar{y}, SUM, P_1)$ $\bar{y} = 0$ $\bar{x} = \bar{x} + \bar{temp}$
$MPI\_REDUCE(x, y, SUM, P_1)$	$if \ \ (P1) \ \ \{\bar{tmp} = \bar{y}; \bar{y} = 0\}$ $\bar{x} = \bar{x} + \bar{y}$
$MPI\_ALL\_REDUCE(x, y, SUM)$	$MPI\_ALL\_REDUCE(\bar{y}, \bar{tmp}, SUM, P_1)$ $\bar{y} = 0$ $\bar{x} = \bar{x} + \bar{tmp}$
$MPI\_RECV(y)/MPI\_SEND(y)$	$MPI\_SEND(\bar{y})/, MPI\_RECV(\bar{y})$
$MPI\_SEND(y)/mpi\_recv(y)$	$MPI\_RECV(\bar{y})/MPI\_SEND(\bar{y})$

4.4.3 Problèmes des flots de données non déterministes

Le mode inverse a besoin de savoir dans quelle branche est passé le code. Illustrons nos propos par un exemple.

```
Initialise V avec X
 pour chaque processeur Pi
 qui n'est pas maitre faire
V := f1(V, Pi)
V = f2(V)
...
V = fn(V)
call mpi_send(V, processeur_maitre)
fin faire

if (processeurmaitre)
call mpi_recv(V, mpi_any_source)
V := fN+1(V)
...
V = fp(V)
Recuperer la variable Y la valeur de V
endif
```

Pour pouvoir construire le code adjoint, il nous faut savoir de quel processeur "mpi_rec " reçoit sa valeur. En effet si par exemple "mpi_rec " reçoit sa valeur de P_1 nous pouvons construire le code adjoint suivant :

```
Initialise V avec X
 pour chaque processeur P_i
 qui n'est pas maitre faire
push(V)
V := f_1(V, P_i)
push(V)
V = f_2(V)
...
push(V)
V = f_n(V)
push(V)
call mpi_send(V, processeur_maitre)
fin faire
if (processeurmaitre)
push(V)
call mpi_rec(V, mpi_any_source)
push(V)
V := f_{N+1}(V)
...
push(V)
V = f_p(V)
endif

if(processeurmaitre)
pop(V)
V̄ := V̄ × f'_p(V)
...
pop(V)
V̄ := V̄ × f'_{n+1}(V)
call mpi_send(V, P_1)
endif
pop(V)

 pour chaque processeur P_i
 qui n'est pas maitre faire
callmpi_rec(V, processeur_maitre)
pop(V)
V̄ := V̄ × f'_n(V) ···
pop(V)
V̄ := V̄ × f'_2(V) pop(V)
V̄ := V̄ × f'_1(V, P_i)
fin faire
```

Pour utiliser la différentiation dans ce cas il faut modifier le code pour le rendre déterministe. De manière générale le code utilisé doit être entièrement déterministe. Le code AIRONUM est presque que totalement déterministe. Quelques cas sont répertoriés, ces cas se situent dans les routines MPI. Quelques modifications des communications ont donc été nécessaire pour utiliser la différentiation.

4.5 Conclusion

Les méthodes que nous appliquons ne sont pas encore intégrées dans l'outil de differentiation. Elles nécessitent des modifications du code générés à l'aide de script. Au fil du travail nous avons rencontrés et mis en évidence de nouveaux problèmes notamment sur l'allocation dynamique. Ce qui a permis une riche interaction avec les concepteurs du logiciel Tapenade.

La différentiation des codes parallèles MPI avait fait il y a quelques années l'objet de travaux de simple transposition sans modification de l'analyse de dépendance, travaux qui produisent des derivées fausses dans de nombreux contextes. L'étude réalisée ici a permis d'étendre l'analyse de dépendances au contexte MPI. Elle nous a permis de vérifier que les méthodes employées pour les routines MPI étaient justes, nous avons aussi proposé une illustration mettant en évidence les adjoints de certaines routines MPI. Nous avons aussi résolu des problèmes d'allocation dynamique empêchant l'exécution automatique du code. Au total, une différentiation directe est validée et celle inverse est en cours.