Université Côte d'Azur - UFR Sciences

École Doctorale de Sciences Fondamentales et Appliquées

Thèse

pour obtenir le titre de

Docteur en Sciences

de l'Université Côte d'Azur

Discipline : Mathématiques

présentée et soutenue par

Ala Taftaf

Développements du Modèle Adjoint de la Différentiation Algorithmique destinés aux Applications Intensives en Calcul

---

Extensions of Algorithmic Differentiation by Source Transformation inspired by Modern Scientific Computing

---

Thèse dirigée par: Laurent Hascoët

Soutenue le 17 Janvier 2017

Jury :

Rapporteurs :	Bruce Christianson	-	Université Hertfordshire, Royaume-Uni
	Uwe Naumann	-	Université RWTH Aachen, Allemagne
Examinateurs :	Jens-Dominik Mueller	-	Université Queen Mary, Royaume-Uni
	Didier Auroux	-	Université Nice Sophia Antipolis, France
Directeur :	Laurent Hascoët	-	INRIA Sophia-Antipolis, France

Résumé. Le mode adjoint de la Différentiation Algorithmique (DA) est particulièrement intéressant pour le calcul des gradients. Cependant, ce mode utilise les valeurs intermédiaires de la simulation d'origine dans l'ordre inverse à un coût qui augmente avec la longueur de la simulation. La DA cherche des stratégies pour réduire ce coût, par exemple en profitant de la structure du programme donné.
Dans ce travail, nous considérons d'une part le cas des boucles à point-fixe pour lesquels plusieurs auteurs ont proposé des stratégies adjointes adaptées. Parmi ces stratégies, nous choisissons celle de B. Christianson. Nous spécifions la méthode choisie et nous décrivons la manière dont nous l'avons implémentée dans l'outil de DA Tapenade. Les expériences sur une application de taille moyenne montrent une réduction importante de la consommation de mémoire.
D'autre part, nous étudions le checkpointing dans le cas de programmes parallèles MPI avec des communications point-à-point. Nous proposons des techniques pour appliquer le checkpointing à ces programmes. Nous fournissons des éléments de preuve de correction de nos techniques et nous les expérimentons sur des codes représentatifs. Ce travail a été effectué dans le cadre du projet européen "AboutFlow".

Mots-clés: Différentiation Algorithmique, Méthode Adjointe, Algorithmes Point-Fixe, Checkpointing, Communication par Passage de Messages, MPI

Abstract. The adjoint mode of Algorithmic Differentiation (AD) is particularly attractive for computing gradients. However, this mode needs to use the intermediate values of the original simulation in reverse order at a cost that increases with the length of the simulation. AD research looks for strategies to reduce this cost, for instance by taking advantage of the structure of the given program.
In this work, we consider on one hand the frequent case of Fixed-Point loops for which several authors have proposed adapted adjoint strategies. Among these strategies, we select the one introduced by B. Christianson. We specify further the selected method and we describe the way we implemented it inside the AD tool Tapenade. Experiments on a medium-size application shows a major reduction of the memory needed to store trajectories.
On the other hand, we study checkpointing in the case of MPI parallel programs with point-to-point communications. We propose techniques to apply checkpointing to these programs. We provide proof of correctness of our techniques and we experiment them on representative CFD codes. This work was sponsored by the European project "AboutFlow".

Keywords: Algorithmic Differentiation, Adjoint Methods, Fixed-Point Algorithms, Checkpointing, Message Passing, MPI

Résumé étendu:

Le projet “AboutFlow” [15] se concentre sur les méthodes d’optimisation basées sur le gradient. Le mode adjoint de la Différentiation Algorithmique (DA) [25], [40] est particulièrement intéressant pour le calcul des gradients. Cependant, ce mode doit utiliser les valeurs intermédiaires de la simulation d’origine dans l’ordre inverse de leur calcul. Quelle que soit la stratégie choisie pour réaliser cette inversion, le coût de cette opération augmente avec la durée de la simulation.

Dans le domaine de la DA, nous recherchons des stratégies afin d’atténuer ce coût, par exemple en tirant parti de la structure du programme donné. Une telle structure fréquente est celle des boucles à point fixe. Les boucles à point fixe (PF) sont des algorithmes qui affinent itérativement une valeur jusqu’à ce qu’elle devienne stationnaire. Nous appelons ”état” la variable qui contient cette valeur et ”paramètres” les variables utilisées pour calculer cette valeur. Comme les boucles PF partent d’une estimation initiale de l’état, a priori fausse, une intuition est qu’au moins les premières itérations de la boucle ont une influence très faible sur le résultat final. Par conséquent, stocker ces iterations pour le calcul d’adjoint est relativement inutile et consomme de la mémoire. De plus, les boucles PF qui commencent à partir d’une estimation initiale très proche de résultat final convergent en seulement quelques iterations. Comme la boucle adjointe de la méthode adjointe standard suit exactement le même nombre des iterations que la boucle originale, celle-ci peut retourner un gradient qui n’est pas suffisamment convergé.

Dans ce travail, nous recherchons un adjoint spécifique pour les boucles PF. Parmi les stratégies documentées dans la littérature, nous avons sélectionné les approches Piggyback [23], Delayed Piggyback [23], Blurred Piggyback [4], Deux phases [10] et Deux-Phases raffinée [10]. Ces adjoints spéciaux parviennent à éviter l’inversion naïve de la séquence d’itérations originale, économisant ainsi le coût d’inversion du flux des données.

La différence entre ces approches est principalement le moment de démarrage des calculs adjoints. Certaines de ces approches commencent à calculer l’adjoint depuis les premières itérations de la boucle originale, comme dans le cas de l’approche Piggyback, certaines d’entre elles attendent jusqu’à ce que l’état soit suffisamment convergé, comme dans le cas de Delayed Piggyback et Blurred Piggyback et d’autres calculent l’adjoint seulement lorsque l’état est totalement convergé, comme dans le cas des approches Deux-Phases et Deux-Phases raffinée. Parmi ces stratégies, nous avons sélectionné l’approche Deux-Phases raffinée pour être implémentée dans notre outil de DA “Tapenade” [31]. Notre choix est motivé par le fait que cette méthode est générale, c’est-à-dire qu’elle ne fait pas d’hypothèses sur la forme de la boucle PF, et aussi qu’elle est relativement facile à implementer vu qu’elle nécessite peu de modifications sur la méthode adjointe standard.

Dans cette méthode, la boucle adjointe est une nouvelle boucle PF qui utilise les valeurs intermédiaires de la dernière itération seulement.

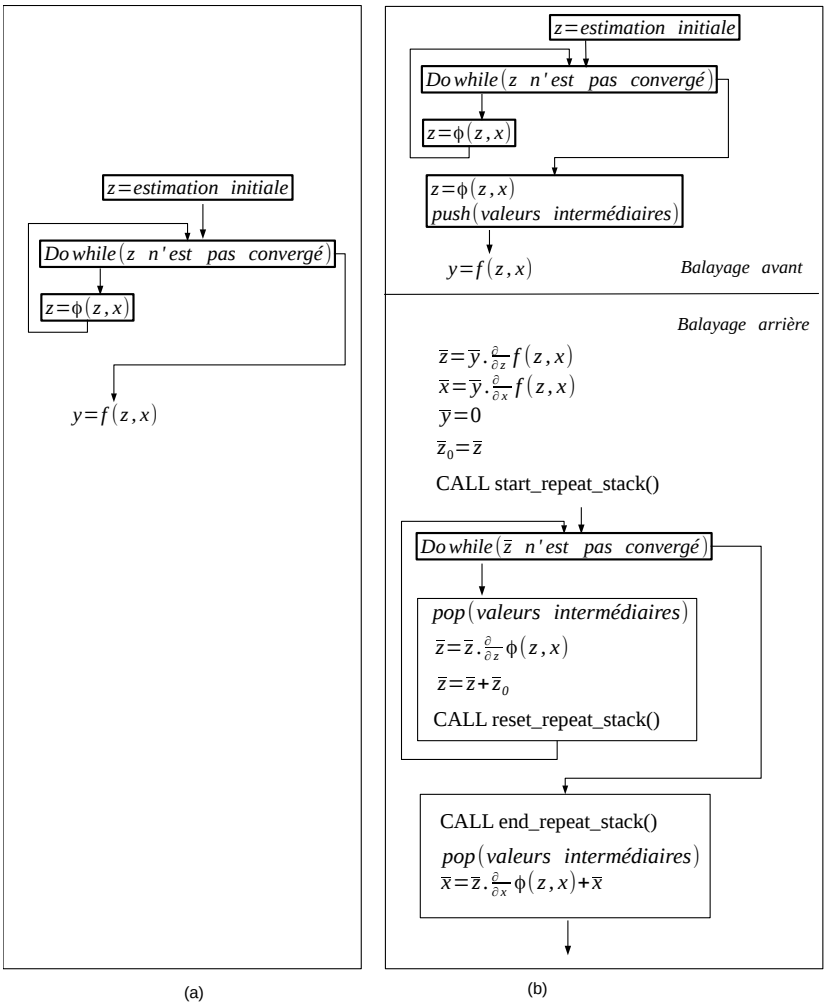

FIGURE 1: (a) Une boucle à point fixe. (b) L'adjoint Deux-Phases raffiné appliqué à cette boucle.

Un exemple des boucles PF est illustré par la figure 1 (a). La boucle initialise l'état z avec une certaine estimation initiale, puis itérativement appelle

$$z = \phi(z, x) \tag{1}$$

jusqu'à ce que z atteigne une valeur stationnaire z_* qui est le point fixe de la fonction $\phi(z, x)$. Ce point fixe est utilisé par la suite pour calculer un résultat final $y = f(z_*, x)$. La figure 1 (b) montre l'application de l'approche Deux-Phases raffinée à cette boucle PF. Cette approche maintient la structure standard des codes adjoints pour tout ce qui est avant et après la boucle PF. Dans le balayage avant, l'approche Deux-Phases raffinée copie la boucle PF du programme d'origine et insère après celle-ci un balayage avant du corps de la boucle PF, dans lequel elle stocke les valeurs intermédiaires de la dernière

itération. Dans le balayage arrière, cette méthode introduit une nouvelle boucle PF qui a son propre variable d'état $\overline{z}$. La variable $\overline{z}$ ne correspond pas ici à l'adjoint de l'état z mais elle est plutôt une variable intermédiaire qui contient les calculs adjoints. La boucle adjointe résout l'équation PF adjointe

$$\overline{z}_* = \overline{z}_* . \frac{\partial}{\partial z} \phi(z_*, x) + \overline{z}_0 \tag{2}$$

qui définit $\overline{z}_*$ en fonction de $\overline{z}_0$ retourné par l'adjoint de la fonction f. L'adjoint Deux-Phases raffiné termine en calculant la valeur de $\overline{x}$ requise, en utilisant $\overline{z}_*$. Nous remarquons ici que l'adjoint Deux-Phases raffiné différentie deux fois la fonction $\phi(z, x)$: une fois par rapport à l'état z à l'intérieur de la boucle PF adjointe et une autre fois par rapport aux paramètres x en dehors de la boucle PF adjointe.

Dans ce travail, nous spécifions plus en détail la méthode Deux-Phases raffinée afin de prendre en compte les particularités des codes réels. En effet, les travaux théoriques sur les boucles PF présentent souvent ces boucles schématiquement comme une boucle while autour d'un seul appel à une fonction ϕ qui implémente l'itération PF. Cependant, les codes réels ne suivent presque jamais cette structure. Même en obéissant à une structure de boucle "while" classique, les boucles PF peuvent contenir par exemple plusieurs sorties. Dans de nombreux cas, l'application de Deux-Phases raffinée à ces structures retourne des codes adjoints erronés. Ceci est dû au fait que les sorties alternatives peuvent empêcher la dernière itération de la boucle de balayer toute la fonction ϕ. Comme l'approche Deux-Phases ne calcule que l'adjoint de la dernière iteration, celle-ci peut dans ce cas ne calculer que l'adjoint d'une partie de ϕ et non l'adjoint de la fonction entière. Pour pouvoir appliquer l'approche Deux-Phases raffinée, nous avons donc besoin de définir un ensemble de conditions suffisantes. En particulier:

- Chaque variable écrite par le corps de la boucle PF doit faire partie de l'état.
- Les variables d'état doivent atteindre des valeurs stationnaires.
- Le flux de contrôle du corps de la boucle PF doit être stationnaire à la convergence de la boucle.

Avant d'implementer l'approche Deux-Phases, une question importante se pose: comment peut-on détecter les boucles PF dans un code donné ? Statiquement, il est très difficile ou même impossible de détecter une boucle PF dans un code donné. Même lorsque cette dernière a une structure simple avec une seule sortie, un outil de DA ne peut pas déterminer statiquement si le flux de contrôle de cette boucle converge ni si chaque variable écrite par la boucle atteindra un point fixe. Par conséquent, nous comptons sur l'utilisateur final pour fournir cette information, par exemple à l'aide

d'une directive. En revanche, contrairement à l'emplacement de la boucle PF, l'état et les paramètres peuvent être détectés automatiquement grâce aux analyses de flux de données. Étant donné l'ensemble **use** des variables lues par la boucle PF, l'ensemble **out** des variables écrites par la boucle PF et l'ensemble **live** des variables utilisées par la suite de La boucle PF, nous pouvons définir:

$$\begin{aligned} \text{état} &= \mathbf{out}(\text{boucle PF}) \cap \mathbf{live} \\ \text{paramètres} &= \mathbf{use}(\text{boucle PF}) \backslash \mathbf{out}(\text{boucle PF}) \end{aligned}$$

Dans l'approche Deux-Phases raffinée, les valeurs calculées par le programme d'origine ne sont stockées que lors de la dernière itération de la boucle PF. Ensuite, elles sont lues à plusieurs reprises dans la boucle adjointe. Malheureusement, notre mécanisme de pile standard ne permet pas ce comportement. Pour implémenter la méthode Deux-Phases raffinée dans notre outil de DA, nous devons définir une extension pour spécifier qu'une certaine zone dans la pile (une "zone à accès répétitif") sera lue à plusieurs reprises. Pour faire ceci, nous avons ajouté trois nouvelles primitives à notre pile, voir la figure 1 (b):

- `start _repeat _stack ()` appelée au début de la boucle PF adjointe. Elle indique que la position actuelle de la pile est le sommet d'une zone à accès répétitif.
- `reset _repeat _stack ()` appelée à la fin du corps de la boucle PF adjointe. Elle indique que le pointeur de la pile doit revenir au sommet de la zone à accès répétitif
- `end _repeat _stack ()` appelée à la fin de la boucle PF adjointe. Elle indique qu'il n'y aura pas d'autre lecture de la zone à accès répétitif.

Nos extensions du mécanisme de pile doivent de plus permettent l'application du compromis stockage-recalcul classique nommé "checkpointing". Ce mécanisme entraine en particulier une alternance complexe de balayages avant (qui empilent des valeurs) et de balayages arrière (qui dépilent des valeurs). En particulier, le checkpointing peut entrainer le démarrage d'un balayage avant au milieu d'une phase d'accès répétitif à la pile. Dans ce cas, il faut protéger la zone d'accès répétitif en empêchant les nouvelles valeurs empilées d'écraser cette zone. Notre solution est de forcer l'ajout des nouvelles valeurs au-dessus de la zone à accès répétitif. Pour faire ceci, nous avons ajouté deux primitives supplémentaires à notre pile:

- `freeze _repeat _stack ()` appelée juste avant la partie balayage vers l'avant (FW sweep) de checkpointing. Elle enregistre la position actuelle du pointeur de

la pile et indique que tous les `push`s de checkpointing doivent sauvegarder leurs valeurs au-dessus du sommet de la zone à accès répétitif.

- `unfreeze _repeat _stack ()` appelée après la partie balayage vers l'arrière (BW sweep) de checkpointing. Elle indique que les `pop`s de checkpointing ont renvoyé le pointeur de la pile au sommet de la zone à accès répétitif. Cette primitive retourne le pointeur de la pile à son ancien emplacement avant le checkpointing de telle sorte que les prochains `pop`s peuvent lire à nouveau les valeurs de la zone à accès répétitif.

Pour implementer la méthode Deux-Phases raffinée, nous avons spécifié la transformation de l'adjoint, de telle façon qu'elle peut être appliquée à toute structure de boucles PF, éventuellement imbriquée. L'idée principale est de définir cette opération comme étant une transformation récursive sur les graphes de contrôle du programme original.

Pour la validation, nous avons expérimenté l'adjoint Deux-Phases sur un vrai code de taille moyenne et quantifié ses avantages, qui sont marginaux en termes d'exécution et significatifs en termes de consommation de mémoire. Nous avons également expérimenté l'adjoint Deux-Phases raffiné sur une structure imbriquée de boucles PF. La structure imbriquée a été exécutée une fois avec une estimation initiale pour la boucle interne qui reste constante à travers les itérations externes, nous l'appelons "estimation initiale constante", et une autre fois avec une estimation initiale qui dépend des résultats de la boucle interne à l'itération externe précédente, nous l'appelons "estimation initiale intelligente". La structure imbriquée avec une estimation initiale intelligente pour la boucle interne effectue moins d'itérations que dans le cas où elle a une estimation initiale constante.

L'application de l'adjoint standard à la structure imbriquée avec une estimation initiale intelligente pour la boucle PF interne retourne un adjoint qui lui aussi a une estimation initiale intelligente pour la boucle adjointe interne. Nous disons que dans ce cas, l'adjoint standard a hérité l'intelligence de l'estimation initiale de la boucle interne d'origine. Contrairement à la méthode standard, l'adjoint Deux-Phases raffiné n'a pas hérité l'intelligence de l'estimation initiale de la boucle interne d'origine. Ceci peut être expliqué par le fait que l'adjoint Deux-Phases ne calcule pas le vrai adjoint de l'état z mais plutôt la valeur d'une variable intermédiaire qui lui est semblable.

En s'inspirant de l'estimation initiale intelligente de l'adjoint standard, nous avons défini une estimation initiale intelligente pour la boucle interne adjointe de la méthode Deux-Phases raffinée. Cette nouvelle estimation depend des résultats obtenus par la boucle interne adjointe à l'itération extérieure précédente. La nouvelle estimation initiale réduit le nombre d'itérations de l'adjoint Deux-Phases de presque moitié.

La stratégie classique pour réduire le coût d'inversion du flux de données du mode adjoint de la DA est un compromis stockage-recalcul nommé "checkpointing", cette strategie sera expliquée en détails dans le chapitre 1. Le checkpointing entraine la répétition, dans un ordre complexe, de certaines parties du programme choisies par l'utilisateur final. Dans la suite, nous écrirons que ces parties sont "checkpointées". Le checkpointing a été largement étudié dans le cas de programmes séquentiels. Cependant, la plupart des codes industriels sont maintenant parallélisés, le plus souvent à l'aide de la bibliothèque MPI [52]. Dans ce cas, la duplication de parties du code risque d'entrainer des incohérences dans la communication des messages. Dans les travaux précédents (l'approche "populaire"), le checkpointing a été appliqué de telle sorte que le morceau de code checkpointé contient toujours les deux extrémités de chaque communication. En d'autres termes, aucun appel MPI à l'intérieur de la partie de code checkpointée ne peut communiquer avec un appel MPI qui est à l'extérieur de cette partie. De plus, les appels de communication non bloquants et leurs *wait*s correspondants doivent être tous à l'intérieur ou à l'extérieur de la partie checkpointée. Dans les travaux antérieurs, cette restriction est le plus souvent tacite. Toutefois, si une seule extrémité d'une communication point à point se trouve dans la partie checkpointée, la méthode ci-dessus produira un code erroné.

Nous proposons des techniques pour pouvoir appliquer le checkpointing aux codes MPI adjoints [41], [54] avec des communications point à point, qui ou bien n'imposent pas de restrictions, ou bien les explicitent afin que les utilisateurs finaux puissent vérifier leur applicabilité.
Une technique est appelée " receive-logging ". Cette technique, illustrée par la figure 2, s'appuie sur l'enregistrement de chaque message au moment où il est reçu, de telle sorte que les communications dupliquées n'ont pas besoin d'avoir lieu. Dans la suite nous omettrons le préfixe `MPI_` des appels de communication.

- Lors de la première exécution de la partie checkpointée, chaque appel de communication est exécuté normalement. Cependant, chaque opération de réception (en fait son *wait* dans le cas d'une communication non bloquante) stocke la valeur qu'elle reçoit dans un emplacement local au processus. Les opérations d'envoi ne sont pas modifiées.

- Pendant l'exécution dupliquée de la partie checkpointée, chaque opération d'envoi ne fait rien (elle est "désactivée"). Chaque opération de réception, au lieu d'appeler `recv`, lit la valeur précédemment stockée pendant la première exécution.

Bien que cette technique lève complètement les restrictions sur le checkpointing des codes MPI, l'enregistrement des messages la rend plus coûteuse que l'approche populaire.

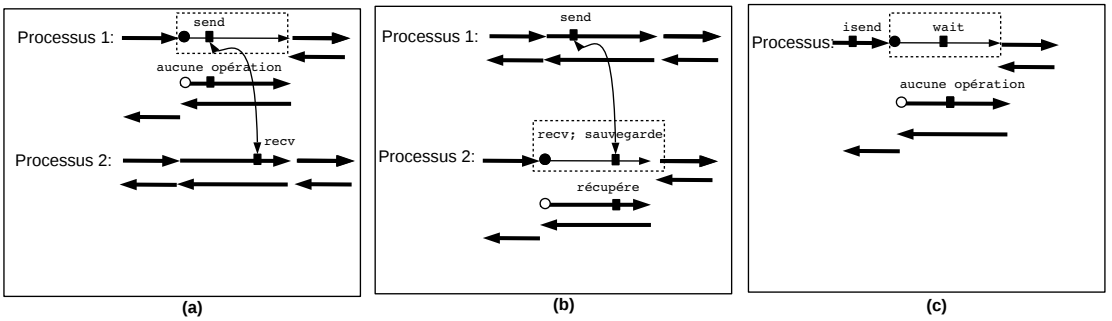

FIGURE 2: Trois exemples dans lesquels nous appliquons du receive-logging. Pour plus de clarté, nous avons séparé les processus: processus 1 en haut et processus 2 en bas. (a) Un programme adjoint après le checkpointing d'un morceau de code ne contenant que la partie *send* d'une communication point à point. (b) Un programme adjoint après le checkpointing d'un morceau de code ne contenant que la partie *recv* d'une communication point à point. (c) Un programme adjoint après le checkpointing d'un morceau de code contenant seulement la partie `wait` d'une communication non bloquante.

Nous pouvons raffiner la technique receive-logging en remplaçant l'enregistrement des valeurs par la duplication des communications à chaque fois que c'est possible, de telle façon que la technique raffinée englobe maintenant l'approche populaire. Ce raffinement est appelé "message-resending". Le principe est d'identifier une paire `send` - `recv` dont les extrémités appartiennent à la même partie de code checkpointée, et de réexécuter cette paire de communications de façon identique pendant la partie dupliquée du checkpointing, effectuant ainsi la communication deux fois. Les communications dont une extrémité n'appartient pas à la partie de code checkpointée sont toujours traitées par receive-logging.

Figure 3 (b) montre l'application du checkpointing couplé avec le receive-logging à un morceau de code. Dans ce morceau de code, nous sélectionnons une paire `send-recv` et nous appliquons du message-resending à cette paire. Comme résultat, voir la figure 3 (c), cette paire est ré-exécutée lors de la duplication de la partie checkpointée et la valeur reçue n'est plus enregistrée lors de la première instance de la partie checkpointée.

Cependant, pour pouvoir appliquer le message-resending, la partie de code checkpointée doit obéir à une contrainte supplémentaire que nous appellerons "étanche à droite". Une partie checkpointée est "étanche à droite" si aucune dépendance de communication ne va de l'aval de (c'est-à-dire après) la partie checkpointée retournant vers la partie checkpointée. Par exemple, la partie checkpointée sur la figure 3 (a) est étanche à droite. Sur cette figure, si on change la partie checkpointée pour qu'elle ne contient plus la partie `recv` de processus 2, cette partie ne deviendra plus étanche à droite car nous allons avoir une dependance allant de processus 2 situé à l'extérieur de la partie checkpointée vers le `send` du processus 1 situé à l'intérieur de la partie checkpointée.

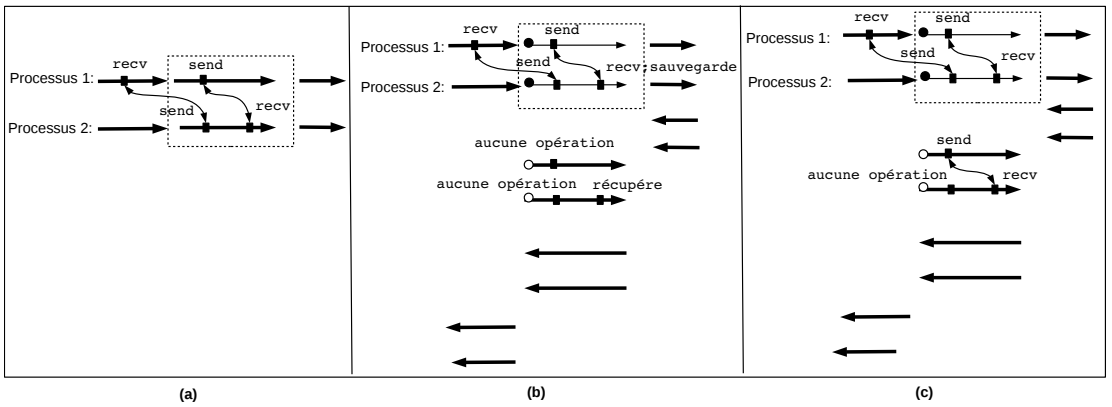

FIGURE 3: (a) Un programme parallèle MPI. (b) L'adjoint correspondant à ce programme après le checkpointing d'un morceau de code en appliquant du receive-logging. (c) L'adjoint correspondant après le checkpointing d'un morceau de code en appliquant du receive-logging couplé avec le message-resending

Une extrémité de la communication est dite "orpheline" par rapport à une partie checkpointée, si elle appartient à cette partie tandis que son partenaire n'appartient pas, par exemple un `send` qui appartient à une partie checkpointé alors que son `recv` n'appartient pas.

Dans le cas général:

- Losque la partie checkpointée n'est pas étanche à droite, nous ne pouvons appliquer que du receive-logging à toutes les extrémités des communications appartenant à la partie checkpointée.
- Dans le cas inverse, c'est-à-dire lorsque la partie checkpointée est étanche à droite, nous recommandons l'application du message-resending à toutes les extrémités de communications non orphelines appartenant à cete partie checkpointée. Pour les extrémités orphelines, nous ne pouvons appliquer que du receive-logging. L'intérêt de combiner les deux techniques est de reduire la consommation de mémoire vu que nous ne sauvegardons maintenant que les `recv` qui sont orphelins.

Dans ce travail, nous fournissons des éléments de preuve de correction de nos techniques receive-logging et message-resending, à savoir qu'elles préservent la sémantique du code adjoint et qu'elles n'introduisent pas des deadlocks. Nous discutons des questions pratiques concernant le choix de morceau de code checkpointé. Nous expérimentons nos techniques sur des codes représentatifs dans lesquels nous effectuons des différents choix de morceaux checkpointés. Nous quantifions les dépenses en termes de mémoire et de nombre de communications pour chaque code adjoint résultant.

Contents

List of Figures

List of Tables

Citations to Previously Published Work

Some portions of Chapters 2 have appeared in the following two papers:

- Taftaf, A. , Pascual,V. and Hascoët, L. "Adjoints of Fixed-Point iterations". 11th World Congress on Computational Mechanics (WCCM XI). 20-25 July 2014, Barcelona, Spain.
- Taftaf, A., Hascoët, L. and Pascual, V. "Implementation and measurements of an efficient Fixed Point Adjoint". International Conference on Evolutionary and Deterministic Methods for Design, Optimisation and Control with Applications to Industrial and Societal Problems. EUROGEN 2015, 14-16 September 2015, Glasgow, UK.

Parts of Chapter 3 has been published as:

- Taftaf, A. and Hascoët, L. On The Correct Application Of AD Checkpointing To Adjoint MPI-Parallel Programs. ECCOMAS Congress 2016, VII European Congress on Computational Methods in Applied Sciences and Engineering. Crete Island, Greece, 5–10 June 2016.
- Taftaf, A. and Hascoët, L. Experiments on Checkpointing Adjoint MPI Programs. International Conference on Numerical Optimisation Methods for Engineering Design. 11th ASMO UK/ISSMO/NOED2016. Munich, Germany, 18-20 July 2016.

Acknowledgements

I am deeply indebted to my supervisor, Dr. Laurent Hascoët for offering me an opportunity within the ECUADOR team. I would like to thank you for all the guiding, cooperation, encouragements, and a lasting support throughout the research. You definitely provided me with the tools that I needed to choose the right direction and successfully complete my thesis.

To the members of the jury Prof. Bruce Christianson, Prof. Uwe Naumann, Prof. Jens-Dominik Mueller and Prof. Didier Auroux, I am grateful for all the time you devoted to reading this. It is my honor and I thank you for the advice and the constructive criticism that contributed to bringing the original draft to this final stage.

I am in debt to all members of ECUADOR team for the useful discussions, in particular, I wish to thank Valérie Pascual, Alain Dervieux and Stephen Wornom. I want to thank Christine Claux for all the support that she gave me.

I would like to express my gratitude to the About Flow project that supported this work. Working within this project was a great chance for learning and professional development. Therefore, I consider myself as a very lucky individual as I was provided with an opportunity to be a part of this project. I am also grateful for having a chance to meet so many wonderful people and professionals who led me through the period of my thesis. I also gratefully acknowledge the financial support I received from the European Commission.

The sacrifices that this work has required, have been felt most strongly by my family and friends. Thus, I would like to express my gratitude to all of them.

Dedicated to my husband Ramzi, my daughter Abrar, my mothers Abla, Rabiaa and Habiba, my fathers Zouhair, Hedi, Mohamed and Ahmed and to all my relatives and friends

Chapter 1

Introduction (français)

Les dérivées de fonctions sont nécessaires dans de nombreux domaines de l'informatique. Elles jouent un rôle central dans les problèmes inverses, l'assimilation de données [46], l'analyse de sensibilité [35], l'optimisation de forme [37] et de nombreux autres domaines. Par exemple, dans l'optimisation de forme, les dérivées et en particulier les gradients sont utilisés pour trouver une amélioration possible du design courant.

Pour les modèles complexes, en particulier ceux implémentés sous forme de programmes informatiques, le développement à la main des codes qui calculent les dérivées de ces fonctions est à la fois extrêmement long et en plus une source d'erreurs. D'un autre côté, le calcul de ces dérivées par des différences divisées sur les programmes qui implémentent ces fonctions renvoie des dérivées inexactes. Les raisons de cette inexactitude sont la troncature des dérivées d'ordre supérieur et les erreurs numériques dues au choix de la perturbation. Contrairement à ces deux approches, une méthode connue sous le nom de " Différentiation Algorithmique" (DA) [25], [40] produit des codes qui calculent des dérivées précises et peut être appliquée à des programmes arbitrairement complexes.

Cette méthode repose sur le fait que le programme d'origine P, calculant une fonction différentiable $F : X \in I\!R^N \rightarrow Y \in I\!R^M$, peut être exprimé comme étant une suite d'instructions élémentaires $\{I_1; I_2; ...I_p; \}$, chacune calculant une fonction différentiable élémentaire $f_1, f_2, ...f_p$, de telle sorte que la fonction $F(X)$ est la composition de ces fonctions élémentaires,

$$F(X) = f_p(...(f_2(f_1(X))...) \tag{1.1}$$

L'application de la règle de différentiation des fonctions composées à $F(X)$ donne une nouvelle fonction $F'(X)$ qui calcule la dérivée première (la Jacobienne):

$$F'(X) = f'_p(X_{p-1}) * ... * f'_2(X_1) * f'_1(X) \tag{1.2}$$

où X_1 est la sortie de la fonction $f_1(X)$, X_2 est la sortie de la fonction $f_2(f_1(X))$, etc. En théorie, nous pouvons donc étendre chaque instruction I_k, de telle façon qu'elle calcule en plus de la fonction élémentaire f_k, sa dérivée f'_k. L'ensemble de ces nouvelles instructions forme un nouveau programme que nous appelons le "programme différentié", P'. Cependant, dans la pratique, la Jacobienne $F'(X)$ peut être une matrice énorme dont la hauteur et la largeur sont de l'ordre du nombre de variables dans le programme d'origine. Le calcul et le stockage de tous les éléments de la Jacobienne peuvent exiger donc une consommation importante de temps et de mémoire. Pour cette raison, dans la pratique on ne calcule pas tous les éléments de la Jacobienne mais plutôt une de ces deux projections:

$$F'(X) * \dot{X} \qquad ou \qquad \overline{Y} * F'(X)$$

où $\dot{X}$ est un vecteur dans $I\!R^N$ et $\overline{Y}$ est un vecteur ligne dans $I\!R^M$. Les modes particuliers de la DA qui calculent ces projections sont appelés respectivement les modes tangent et adjoint.

Alors que la formule $F'(X) * \dot{X}$ calcule (une combinaison linéaire) des colonnes de la Jacobienne, la formule $\overline{Y} * F'(X)$ fournit (une combinaison linéaire) des lignes de la Jacobienne. Le coût du calcul de la Jacobienne complète est donc proportionnel dans le premier cas au nombre d'entrées du programme, et dans le deuxième cas au nombre de sorties du programme. Dans ce travail, l'objectif final est de calculer des gradients. Dans ce cas, le résultat de la fonction à différentier est un scalaire et le mode adjoint est donc le plus efficace pour calculer le gradient. Nous nous concentrerons donc dans ce qui suit sur ce mode.

Le mode adjoint de la DA calcule $\overline{X} = \overline{Y} * F'(X)$. Rappelant l'équation 1.2, on obtient:

$$\overline{X} = \overline{Y} * f'_p(X_{p-1}) * ... * f'_2(X_1) * f'_1(X) \tag{1.3}$$

Comme le produit matrice par vecteur est beaucoup moins cher que le produit matrice par matrice, cette équation est mieux évaluée de gauche à droite. Pour alléger les notations, on appellera $\overline{X}_{p-1}$ le produit du vecteur $\overline{Y}$ par la Jacobienne $f'_p(X_{p-1})$, c'est-à-dire
$\overline{X}_{p-1} = \overline{Y} * f'_p(X_{p-1})$, $\overline{X}_{p-2}$ le produit du vecteur $\overline{X}_{p-1}$ par la Jacobienne $f'_{p-1}(X_{p-2})$, c'est-à-dire $\overline{X}_{p-2} = \overline{X}_{p-1} * f'_{p-1}(X_{p-2})$ et ainsi de suite jusqu'à ce que nous définissions à la fin $\overline{X}_0 = \overline{X}_1 * f'_1(X)$. Par définition $\overline{X}$ est $\overline{X}_0$.
Du point de vue des programmes, toutes les valeurs intermédiaires X_k du programme d'origine sont contenues dans des variables. De même, les valeurs des dérivées $\overline{X}_k$ seront placées dans des nouvelles variables du programme différentié. Par conséquent, le programme différentié, en plus des variables w du programme d'origine, devra déclarer autant des variables différentiées $\overline{w}$, de mêmes formes et de mêmes dimensions que

w. Ces nouvelles variables sont appelées : "les variables adjointes". Cependant, nous notons que dans l'équation 1.3, les valeurs du programme d'origine sont utilisées dans l'ordre inverse de leur calcul, c'est-à-dire X_{p-1} est utilisée en premier, puis X_{p-2}, puis X_{p-3}, etc. On nomme cette question "l'inversion de flux de données". Deux stratégies sont couramment utilisées pour pouvoir utiliser les valeurs du programme d'origine dans l'ordre inverse:

- l'approche Recompute-All [16] recalcule les valeurs du programme d'origine à chaque fois que ceci est nécessaire, en redémarrant le programme à partir d'un état initial mémorisé,
- l'approche Store-All [7] [31] stocke les valeurs intermédiaires dans une pile, ou au moins celles qui seront nécessaires, lors d'une exécution préliminaire du programme original connu sous le nom de balayage avant (FW sweep). Ensuite, cette exécution est suivie par un balayage dit arrière (BW sweep) qui calcule les dérivées adjointes en utilisant les valeurs originales mémorisées. Les primitives `push` et `pop` nécessaires sont fournies par une bibliothèque séparée.

Les deux approches Recompute-All et Store-All se révèlent impraticables sur des grandes applications réelles à cause de leurs coûts en termes de temps ou d'espace mémoire respectivement. Des compromis stockage-recalcule sont nécessaires. Une stratégie classique est appelée "checkpointing". Dans notre contexte (Store-All), le checkpointing [28] consiste à sélectionner une partie C d'un programme P et à ne pas stocker les valeurs intermédiaires de C au cours du balayage avant sur P. Au lieu de cela, on stocke le minimum de données nécessaires (un "snapshot") pour pouvoir exécuter C à nouveau. Il s'agit donc d'une exécution simple de C et non pas un balayage avant sur C. Pendant le balayage arrière sur P, lorsque l'on atteint à nouveau la partie C, on remet en place le contenu du snapshot ce qui permet d'exécuter C à nouveau, cette fois sous la forme d'un balayage avant standard.

L'objectif de cette thèse est d'étudier plus en détail les techniques qui aident à limiter le coût de l'inversion du flux de données de l'adjoint. Deux de ces techniques liées au mécanisme de checkpointing ont été sélectionnées.

- Dans le chapitre 3, nous considérons l'adjoint des boucles à point fixe, pour lesquelles plusieurs auteurs ont proposé des stratégies adjointes adaptées. Parmi ces stratégies, nous choisissons la méthode "Deux-Phases raffinée" de B. Christianson [10]. Cette méthode exige des mécanismes originaux tels que l'accès répétitif à la pile ou encore la différenciation dupliquée de corps de la boucle par rapport à des différentes variables indépendantes. Nous décrivons comment cette méthode

doit être spécifiée afin de prendre en compte les structures particulières présentes dans les codes réels tels que les boucles avec des sorties multiples. Nous décrivons comment les différentes variables (état et paramètres) requises par l'adjoint peuvent être détectées automatiquement grâce à l'analyse du flux de données d'un outil de DA. Nous décrivons la façon dont nous avons étendu le mécanisme standard de la pile et la façon dont nous avons implémenté la méthode Deux-Phases dans notre outil de DA Tapenade. Pour la validation, nous avons expérimenté la méthode Deux-Phases sur un code réel de taille moyenne et nous avons quantifié ses avantages qui sont marginaux en termes d'exécution et significatifs en termes de consommation de mémoire. Nous avons étudié la question connexe de l'estimation initiale dans le cas des boucles à point fixe imbriquées.

- Dans le chapitre 4, nous abordons la question du checkpointing appliqué aux programmes parallèles MPI adjoints [41], [54]. D'une part, nous proposons une extension du checkpointing dans le cas de programmes parallèles MPI avec des communications point à point, de telle sorte que la sémantique du programme adjoint est préservée pour tout choix du fragment de code checkpointé. D'autre part, nous proposons une technique alternative, plus économique mais qui requiert un certain nombre de restrictions sur le choix de la partie de code checkpointée. Nous fournissons des éléments de preuve de correction de nos techniques, à savoir qu'elles préservent la sémantique du code adjoint et qu'elles n'introduisent pas des deadlocks. Nous discutons des questions pratiques concernant le choix de la combinaison de techniques à appliquer étant donné un morceau de code checkpointé ainsi que le choix de morceau de code checkpointé lui-même. Nous expérimentons nos techniques sur des codes représentatifs dans lesquels nous effectuons différents choix de fragments checkpointés. Nous quantifions les coûts en termes de mémoire et de nombre de communications pour chaque code adjoint résultant.

Chapter 2

Introduction (english)

2.1 Introduction

Derivatives of functions are required in many areas of computational science. They play a central role in inverse problems, e.g. data assimilation [46], sensitivity analysis [35], design optimization [37] and many other domains. For instance, in design optimization, derivatives and in particular gradients are used to find a possible improvement of a current design.

For complex models, especially those implemented as computer programs, developing by hand codes that compute the derivatives of these functions is error-prone and extremely time-consuming. From the other side, computing these derivatives by divided differences on the programs that implement these functions returns inaccurate derivatives. The reasons for this inaccuracy being the cancellation in floating-point arithmetic due a too small perturbation ϵ and the truncation error due a too large perturbation. In contrast to these two approaches, the method known as Algorithmic Differentiation (AD) produces codes that compute accurate derivatives and can be applied to arbitrarily complex programs. This method relies on the fact that the original program can be expressed as a sequence of elementary instructions, each of them computing an elementary differentiable function. Applying the chain rule of calculus to this program produces a new program that includes instructions that compute the derivatives.

There exist two fundamental modes of AD: tangent and adjoint. When the number of inputs is much larger then the number of outputs, the adjoint mode is recommended. This work was supported by an European Project called "AboutFlow" [15]. This project focuses on methods of optimization that are gradient-based. Since the adjoint mode is particularly attractive for computing gradients, since the number of outputs is one, this thesis concentrates on this mode.

The adjoint mode, however, needs to use the intermediate values of the original simulation in reverse order. Whatever strategy is chosen to achieve this reversal, the cost for doing this increases with the length of the simulation. AD research looks for strategies to mitigate this cost, for instance by taking advantage of the structure of the given program. In this thesis, we consider the frequent case of Fixed-point loops, for which several authors have proposed adapted adjoint strategies. We explain why we consider the strategy initially proposed by B. Christianson as the best suited for our needs. We describe the way we implemented this strategy in our AD tool Tapenade. Experiments on a medium-size application shows a major reduction of the memory needed to store trajectories.

Another way to reduce the cost of data flow reversal is to employ a trade-off between storage and recomputation of the intermediate values. This trade-off is called checkpointing. Checkpointing has been largely studied in the case of sequential programs. However, most industrial-size codes are now parallelized. In the case of parallel programs implemented by using the MPI library, the presence of communications seriously restricts application of checkpointing. In most attempts to apply checkpointing to the adjoint MPI codes, a number of restrictions apply on the form of communications that occur in the checkpointed pieces of code. In many works, these restrictions are not explicit, and an application that does not respect these restrictions may lead to an erroneous derivative code. In this thesis, we propose techniques to apply checkpointing to adjoint MPI codes with point-to-point communications, that either do not impose these restrictions, or explicit them so that the end users can verify their applicability. These techniques rely on both adapting the snapshot mechanism of checkpointing and on modifying the behavior of communication calls. We provide proof of correctness of these strategies, and we demonstrate in particular that they cannot introduce deadlocks. We experiment these strategies on representative CFD codes.

This introduction chapter presents the basics of AD, or at least those that are useful to understand the sequel. In section 2.2, we present briefly the principal modes and techniques of AD. In section 2.3, we show how AD differentiated codes can benefit from static data-flow analysis. In section 2.4, we present the AD tool "Tapenade" that we have used for implementation and validation.

2.2 Elements of Algorithmic Differentiation

Algorithmic Differentiation (called also Automatic Differentiation) [25], [40] is a set of techniques that, given a program P that computes some differentiable function $F : X \in \mathbb{R}^N \rightarrow Y \in \mathbb{R}^M$, builds a new program P' that computes the derivatives of F. The

main idea can be described in two steps. In a first step, we focus on one particular run-time trace of the code execution, i.e. we consider control as fixed and the program becomes one large sequence of simple instructions. Control will be re-introduced in the differentiated code in the final stage. In a second step, P being now a sequence of instructions $\{I_1; I_2; ...I_p; \}$ each of those computing one differentiable elementary function $\{f_1, f_2, ...f_p\}$, the function $F(X)$ is the composition of these elementary functions, i.e.

$$F(X) = f_p(...(f_2(f_1(X))...) \tag{2.1}$$

Applying the chain rule to $F(X)$ gives a new function $F'(X)$ that computes the first order full derivatives, i.e. the Jacobian:

$$F'(X) = f'_p(X_{p-1}) * ... * f'_2(X_1) * f'_1(X) \tag{2.2}$$

In which X_1 is the output of the function $f_1(X)$, X_2 is the output of the function $f_2(f_1(X))$, etc.
In theory, we may extend every instruction I_k, so that it computes in addition to the elementary function f_k, its derivative f'_k. The set of these new instructions forms a new program that we call the "differentiated program", i.e. P'. However, in practice, the Jacobian $F'(X)$ may be a huge matrix whose height and width are of the order of the number of variables in the original program. Computing the whole Jacobian may require too much time and memory. To deal with this difficulty, one may not compute all the Jacobian elements but rather one of these two projections:

$$F'(X) * \dot{X} \qquad or \qquad \overline{Y} * F'(X)$$

where $\dot{X}$ is a vector in $\mathbb{R}^N$ and $\overline{Y}$ is a row-vector in $\mathbb{R}^M$. The particular modes of AD that compute these projections are called respectively the tangent and adjoint modes.

2.2.1 The Tangent Mode of Algorithmic Differentiation

The tangent mode computes $\dot{Y} = F'(X) * \dot{X}$. Recalling equation 2.2, we get:

$$\dot{Y} = f'_p(X_{p-1}) * ... * f'_2(X_1) * f'_1(X) * \dot{X} \tag{2.3}$$

Since the product matrix by vector is much cheaper than the product matrix by matrix, this equation is best evaluated from right to left. For short, we will call $\dot{X}_1$ the product of the vector $\dot{X}$ by the Jacobian $f'_1(X)$, i.e. $\dot{X}_1 = f'_1(X) * \dot{X}$, then $\dot{X}_2$ the product of the vector $\dot{X}_1$ by the Jacobian $f'_2(X_1)$, i.e. $\dot{X}_2 = f'_2(X_1) * \dot{X}_1$, and so on until we define at the end $\dot{X}_p = f'_p(X_{p-1}) * \dot{X}_{p-1}$. By definition, $\dot{Y}$ is $\dot{X}_p$.

In equation 2.3, the primal values are used in the derivative computations in the same order they are computed by the original program., i.e. X is needed first, then X_1, etc. Therefore, an algorithm that computes $\dot{Y}$ is relatively easy to implement: just keep the original variables w that compute the successive X_k and introduce new program variables $\dot{w}$ of the same shape as w that compute the successive mathematical variables $\dot{X}_k$ that we have just defined. These new variables are called "tangent variables".
The tangent code is thus a copy of the original program, in which we insert new instructions that compute the tangent derivatives $\dot{X}_k$. Since an instruction may use a variable and immediately overwrite it, its differentiated instruction must be inserted *before* it, so that the derivatives rightfully use the value of the variable before it is overwritten. Figure 2.1 illustrates the application of the AD tangent mode on a piece of code. By

(a)	(b)
subroutine $F(x, y)$	*subroutine* $\dot{F}(x, \dot{x}, y, \dot{y})$
	$\dot{w}_1 = 2 * x * \dot{x}$
$w_1 = x^2$	$w_1 = x^2$
$w_2 = 2 * w_1^3$	$\dot{w}_2 = 6 * w_1^2 * \dot{w}_1$
$w_3 = \sin(w_2)$	$w_2 = 2 * w_1^3$
$y = 3 * w_3$	$\dot{w}_3 = \cos(w_2) * \dot{w}_2$
	$w_3 = \sin(w_2)$
	$\dot{y} = 3 * \dot{w}_3$
	$y = 3 * w_3$

FIGURE 2.1: (a) Example of code. (b) The tangent mode applied to this code

convention we represent the derivatives variables with a dot above the name of their primal variable. Note that each derivative variable must be declared with the same type and size as its primal.

2.2.2 The Adjoint Mode of Algorithmic Differentiation

The adjoint mode of AD, called also "Reverse mode", computes $\overline{X} = \overline{Y} * F'(X)$. Recalling equation 2.2, we get:

$$\overline{X} = \overline{Y} * f'_p(X_{p-1}) * ... * f'_2(X_1) * f'_1(X) \tag{2.4}$$

Here also, we prefer the product vector by matrix to the product matrix by matrix because it is cheaper. Therefore, this equation is best evaluated from left to right. For short, we will call $\overline{X}_{p-1}$ the product of the vector $\overline{Y}$ by the Jacobian $f'_p(X_{p-1})$, i.e. $\overline{X}_{p-1} = \overline{Y} * f'_p(X_{p-1})$, then $\overline{X}_{p-2}$ the product of the vector $\overline{X}_{p-1}$ by the Jacobian $f'_{p-1}(X_{p-2})$, i.e. $\overline{X}_{p-2} = \overline{X}_{p-1} * f'_{p-1}(X_{p-2})$ and so on until that we define at the end

$\overline{X}_0 = \overline{X}_1 * f_1'(X)$. By definition $\overline{X}$ is $\overline{X}_0$.
Similarly to the case of the tangent mode, we construct here an algorithm that keeps the original variables w that compute the successive X_k and introduces new program variables $\overline{w}$ of the same shape as w that hold the successive mathematical variables $\overline{X}_k$ that we have just defined. These new variables are called "adjoint variables".

Notice that in equation 2.4, the primal values are needed in the opposite of their computation order, i.e. X_{p-1} is needed first, then X_{p-2}, then X_{p-3}, etc. Therefore, unlike the tangent code, where the differentiated instructions are computed together with the original ones, the adjoint code must consist of two sweeps. The first sweep (the forward sweep (FW)) runs the original program and computes the intermediate values. The second sweep (the backward sweep (BW)) computes the adjoint derivatives, using the intermediate primal values computed in the first sweep. Figure 2.2 illustrates the application of the AD adjoint mode on a piece of code. In figure 2.2, we name the new derivative variables $\overline{w}_k$, of the same type and size as the original ones w_k so that these intermediate variables hold the adjoint derivatives.

(a)	(b)
subroutine $F(x, y)$	*subroutine* $\overline{F}(x, \overline{x}, y, \overline{y})$
$w_1 = x^2$	$w_1 = x^2$
$w_2 = 2 * w_1^3$	$w_2 = 2 * w_1^3$
$w_3 = \sin(w_2)$	$w_3 = \sin(w_2)$
$y = 3 * w_3$	$y = 3 * w_3$ *Forward Sweep*
	$\overline{w}_3 = 3 * \overline{y}$ *Backward sweep*
	$\overline{y} = 0$
	$\overline{w}_2 = \overline{w}_3 * \cos(w_2)$
	$\overline{w}_3 = 0$
	$\overline{w}_1 = 6 * \overline{w}_2 * w_1^2$
	$\overline{w}_2 = 0$
	$\overline{x} = 2 * \overline{w}_1 * x$
	$\overline{w}_1 = 0$

FIGURE 2.2: (a) Example of code. (b) The adjoint mode applied to this code

However, the example in figure 2.2 is oversimplified in that no variable is overwritten (Single Assignment code). This is almost never the case in real codes, where memory space is limited. Since these values are needed to compute the derivatives, strategies must be designed to retrieve the values in the reverse order. We will see later how this problem is solved, but let us keep in mind that there is a penalty attached to the adjoint mode coming from the need of a data-flow reversal.

2.2.3 Comparison of the merits of Tangent and Adjoint modes

To compare the merits of Tangent and Adjoint AD, consider the $M * N$ Jacobian of a function $f : I\!R^N \rightarrow I\!R^M$, see figure 2.3. Let us compare the run-time cost of computing this Jacobian by using the tangent mode with the cost by using the adjoint mode. Each run of the tangent code costs only a small multiple of the run-time of the program P that computes f. This ratio, we call it C_t, typically ranges between 1 and 3. Since the tangent code returns only one column at once, computing the whole Jacobian requires running the tangent code N times. At the end, the total run-time cost of computing the Jacobian by using the tangent mode is $C_t * N * runtime(P)$. Similarly, each run of the adjoint code costs only a small multiple of the run-time of P. However, this ratio, we call it C_a, is slightly greater than C_t because of the data-flow reversal in the case of the adjoint mode. The ratio C_a typically ranges between 5 and 10. Since, the adjoint code returns only one row at once, computing the whole Jacobian requires running the adjoint code M times. At the end, the total run-time cost of computing the Jacobian by using the adjoint mode is $C_a * M * runtime(P)$. When N is much larger than M, the adjoint mode is recommended. Therefore, this mode is particularly attractive for computing gradients (i.e. $M = 1$).

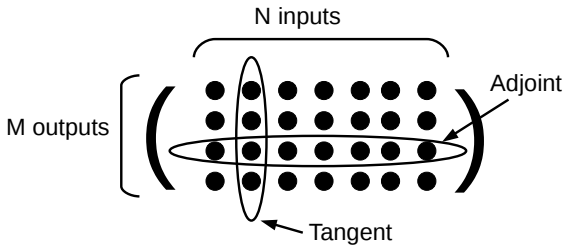

FIGURE 2.3: Computing the Jacobian elements by tangent and adjoint modes

2.2.4 Dealing with the data-flow reversal of the adjoint mode

Applying the adjoint has an extra cost coming from the reversal of the data-flow. We saw in subsection 2.2.2 that in the forward sweep of the adjoint, each instruction may overwrite the values computed by previous instructions. Since these values are needed to compute the adjoint derivatives, strategies must be designed to retrieve these values in reverse order. Two strategies are commonly used in AD tools:

`Recompute-All:` recomputes the intermediate values needed by the derivative of each instruction I_k, by restarting (a slice of) the original program from the stored initial state X_0 until instruction I_{k-1}. Figure 2.4 (a) illustrates this approach. Left-to-right arrows

represent the execution of the original instructions I_k and right-to-left arrows represent the execution of the derivatives instructions $\overline{I}_k$. The black dot represents the storage of the initial state X_0 and each of the white dots represents the restoration of this state. The memory cost is the storage of X_0, which is a constant cost. The run-time cost is quadratic in the number of instructions p. This approach is used for instance by the AD tool TAF [16].

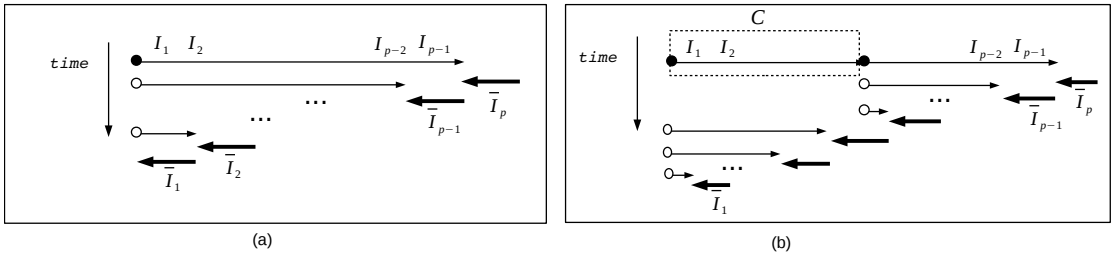

FIGURE 2.4: (a) Data-flow reversal with Recompute-All. (b) Checkpointing a piece of code C with Recompute-All

`Store-All:` stores each intermediate value X_k that is overwritten during the forward sweep onto a stack, then retrieves these values before they are needed by the derivative instructions during the backward sweep. Figure 2.5 (a) illustrates this approach. Left-to-right arrows represent the execution of the original instructions I_k. These arrows are drawn thicker to reflect the fact that the original instructions store the overwritten values. Right-to-left arrows represent the execution of the derivative instructions $\overline{I}_k$. These instructions restore the stored intermediate values and use them to compute the derivatives. The memory cost is proportional to the number of instructions p. In contrast, there is no repeated computation of the original instructions, so there is no extra run-time cost. Admittedly, there is a small run-time penalty associated to the push/pop stack operations, but it is a fixed cost per original instruction, so it has a negligible effect on the complexity measurements with respect to p. This approach is used for instance by the AD tools Adifor [7] and Tapenade [31].

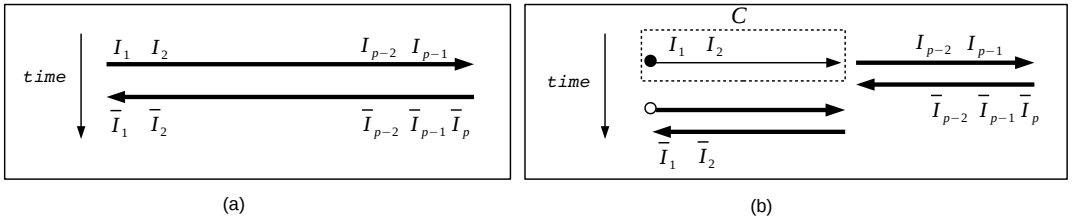

FIGURE 2.5: (a) Data-flow reversal with Store-All approach. (b) Checkpointing a piece of code C with Store-All

In the sequel, we use Store-All because this is the approach of our application tool Tapenade. Suppose for instance that we change the example of figure 2.2 (a) so that

only one variable, w, holds the intermediate computations. We apply the adjoint mode together with the Store-All approach to this new code. The resulting program is sketched in figure 2.6 (b). We see that the value of w is saved into the stack each time it is overwritten by an instruction.

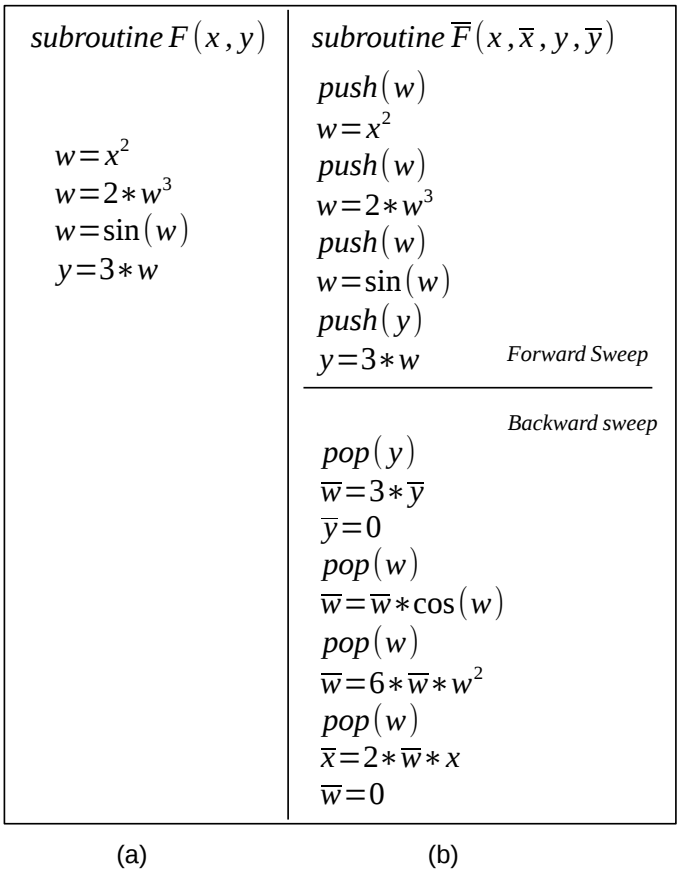

FIGURE 2.6: (a) Example of code. (b) The adjoint mode coupled with Store-All applied to this code

On large real applications both Recompute-All and Store-All approaches turn out to be impracticable due to their cost in time or memory space respectively. Trade-offs are needed, and a classical one is called "checkpointing".

- In the Recompute-All approach, checkpointing means selecting a part of code C and storing the state just after exit from this part. Recomputing the needed values can then start from this state instead of the initial state X_0. The result of checkpointing C is shown on figure 2.4 (b). At the cost of storing one extra state, the run-time cost has been divided almost by two.

- In the Store-All approach, checkpointing means selecting a part of code C and in *not* storing its intermediate values, but rather storing the minimum amount of data needed to run this part again later ("a snapshot"). After taking the

snapshot, the original program is run with no storage of intermediate values. Then, before computing the derivatives of C, C is run again this time with storing the intermediate values. The result of checkpointing C is shown on figure 2.5 (b). The thin left-to-right arrow represents the first execution of C, in which no storage of intermediate values is performed. The black dot reflects the storage of the snapshot and the white dot reflects its retrieval. At the cost of storing the snapshot and of running C twice, the peak stack size is divided almost by two.

A good choice of checkpointed parts is vital for efficient adjoint differentiation of large programs. One may allow the AD-tool user to select the checkpointed parts in order to reduce the overall memory requirement. This approach is provided for instance by the AD tools TAF [17] and Tapenade [28]. As an alternative, one may exploit the call graph structure of the program and choose the calls to subroutines as checkpointed parts. This approach is used for instance by the AD tools OpenAD [42] and Tapenade [31]. There is in general no systematic method to organize an "optimal" choice of checkpointed parts, e.g. a choice that, given a fixed maximal memory space for checkpointed parts, would minimize the number of extra recomputations (for store-all). However, there might be optimal checkpointing schemes for some particular code structures. This is in particular the case for time-stepping procedures. If the number of time steps is known, if the computational costs of the time steps are almost constant, and the maximum number of checkpointed parts is fixed (e.g. by memory limitations of the machine), then an optimal checkpointing schedule can be computed in advance to achieve an optimal run time increase. This optimal scheme is called binomial. It was proved in [22] that this scheme achieves a logarithmic growth of memory and run-time, with respect to p. This approach is used for instance by the AD tool ADOL-C [36]. Other checkpointing schemes may have an optimal configuration which is extremely expensive to find. For instance, if checkpointed parts may be placed at subroutine calls only, the question of finding the optimal set of calls that must be checkpointed is an NP-complete problem [38], [39], for which only approximate solutions can be found in a reasonable time.

2.2.5 Implementation methods of Algorithmic Differentiation

There are two basic approaches for applying Algorithmic Differentiation to a program:

`Operator overloading`: It consists in overloading the arithmetic operations so that these operations propagate the derivative information along the differentiated code. The main idea is to replace the types of the floating-point variables with a new type that holds the derivative in addition to the primal information. This depends on the language of the original code. The AD tool boils down to a library that defines both the overloaded

type and the arithmetic operations that operate on this type. One great strength of this approach is that it can be easily re-defined to compute higher-order derivatives. Also, the original program is barely changed as the code that computes the derivatives is defined inside the libraries. However, since the execution of the overloaded operations follow the order of the original program, this approach requires specific strategies for the adjoint mode which reduce the performance of the differentiated program. Examples of AD tools based on operator overloading are ADOL-C [1], DCO/Fortran and DCO/C++ [40].

`Program Transformation`: It consists in building a new source program that computes the derivatives [26]. The AD tool must feature a compiler that after parsing the original program and building an internal representation of it, generates a new program that computes the derivatives of the original one. This approach allows the AD tool to perform static analysis on the original program. These analysis are useful to produce an efficient code that consumes less in term of time and memory. This makes the program transformation approach, thus, the best choice for the adjoint mode, specially because this mode requires reversing the data-flow of the original program. This approach, however, is hard to implement which may explain why the operator overloading AD tools appeared earlier than the ones of the program transformation. Examples of source transformation based AD tools are Tapenade [31], TAF [16] and OpenAD/F [2].

A possible combination of Operator Overloading and Program Transformation has been studied in [13]. The combinatorial method exploits the advantages of each of these two approaches: the flexibility and robustness of Operator Overloading and the efficiency of source transformation. The main idea is to identify the parts of codes that are the most expensive in terms of number of operations and apply Program Transformation to them. For the rest of the code, Operator Overloading is applied. The resulting code shows a significant reduction in terms of time and memory in comparison with the one on which only Operator Overloading is applied.

2.3 Improving the differentiated code : static Data-flow analyses

Ideally, AD tools should produce differentiated programs as efficient as the best hand-coded versions. To this end, a set of techniques have been developed in order to improve the performance of codes generated by AD tools. One of these techniques is static data flow analyses [27] run on the original program. These analyses statically gather information that is useful to produce an efficient differentiated program.
Data-Flow analyses depend on the internal representation of programs [55]. The most appropriate program representation appears to be a call graph of flow graphs:

- The call graph is a directed graph whose nodes are the subroutines or the functions of the original program and the edges are the calls between these nodes. An arrow from a node A to a node B reflects that A possibly calls B. Recursions are cycles in the call graph.
- Each subroutine or function is represented by a flow graph. The flow graph is a directed graph with one node for each basic block. Arrows between these basic blocks represent the flow of control. Flow graphs may be cyclic, due to loops and other cyclic control.

The individual instructions are represented as abstract syntax trees. A symbol table is associated to each basic block. It saves variable names, function names, type names and so on.

The classical problem of static code analysis is known as undecidability. This means that in many situations, the answer to a given data-flow question can be not only "yes" or "no", but also "maybe". For example a data flow question can be: "at this point in the code, is the value of V greater than 5 ?", or in the context of AD: "at this point in the code, does the value of V depend on the independent inputs ? ". In general, no data-flow analysis can guarantee to reply only "yes" and "no" to these questions on any code. The theoretical reason for that is the undecidability of the termination problem whose consequence is that for any data-flow analysis, one can exhibit a code on which the analysis cannot decide between "yes" and "no". In practice, a far more frequent reason is that the static information available on the inputs is limited and the methods to propagate this information through the code are approximate.
So, whatever effort we put into the development of our tools, we must be prepared to uncertainty on the data-flow information. In other words all data-flow analyses, and all the program transformations that follow must be conservative, i.e. take safe decisions when the data-flow information is uncertain, so as to produce correct code. Conservativity obviously depends on the particular analysis and transformation. For example, when a code simplification is triggered by the fact that V is greater than 5, then the simplification is forbidden in case of uncertainty so that in this case, "maybe" is treated as "no". For other analyses "maybe" may have to be treated as "yes".

Data-flow analyses must be carefully designed to avoid combinatorial explosion. A classic solution is to choose a hierarchical model. In this kind of model, two sweeps through the program are performed: a first sweep that computes local synthesized information for instance on each subroutine or on each basic block. This sweep is performed bottom-up, starting from the smallest levels of program representation and propagating the synthesized information up to the larger levels. Consequently, the synthesized information

must be independent from the rest of the program, i.e. the calling context. Then, a second sweep uses the synthesized information. This sweep is performed top-down on the program and it is context dependent.

2.3.1 Data-flow analyses for Algorithmic Differentiation

Naive application of the tangent or of the adjoint mode to a given program produces a differentiated code that computes the derivatives of all the output variables with respect to all the input variables. However, in practice we may need only the derivatives of some selected outputs (called "dependent") of the original program with respect to some selected inputs of this program (called "independent"). We call "`active`" each variable that belongs to one computational path that relates the independent variables to the dependent ones.

To improve the run time of the differentiated program, one should at least:

1. eliminate all derivative instructions that compute derivatives of variables that are not `active`, and simplify out all occurrences of these variables in derivative expressions.
2. eliminate all instructions that compute primal values that are not used in the (remaining) derivative instructions.

Instructions 1 and 2 can be eliminated from the differentiated code by using a set of analyses run on the original program. These analyses are respectively "`Activity`" and "`Diff-liveness`". `Activity` analysis detects all variables that are `active` and thus need to be differentiated. `Diff-liveness` analysis detects all variables, called "`diffLive`", whose values are needed in the computation of derivatives and thus need to be computed in the differentiated code.
Notice that the `Activity` analysis is useful even when all the inputs are independents and all the outputs are dependents. A variable can also be detected as inactive after it receives a constant value, and likewise when we can prove that its derivative is not used in the sequel of the differentiated code for the computation of the final derivatives.
Figure 2.7 illustrates the benefits of both `Activity` and `Diff-liveness` analyses on an adjoint code. In figure 2.7 (a), we set the variable x as independent and the variable y as dependent. The intermediate variables w_2 and w_3 do not belong to the computational path that relates x to y, i.e. w_2 and w_3 do not depend on x and do not influence y. Consequently, `Activity` analysis detects these two variable as `non-active` and thus no instruction that differentiates these variables has to appear in the differentiated code,

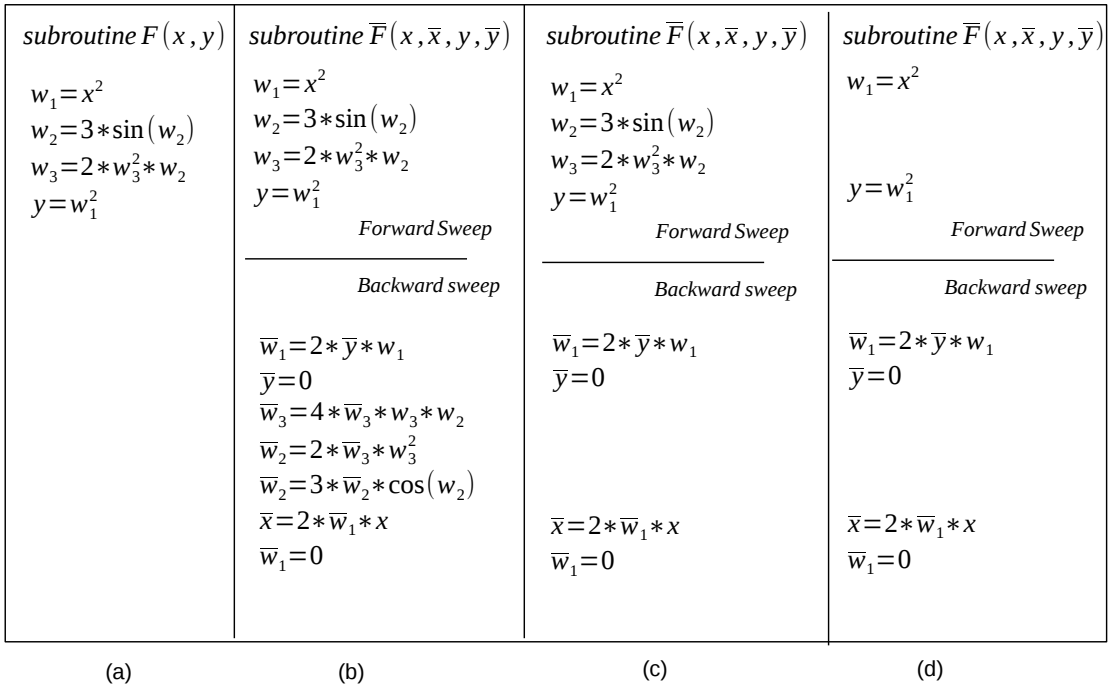

FIGURE 2.7: Effect of `Activity` and `Diff-liveness` analyses on an adjoint AD code. (a) Example of code. (b) Naive adjoint mode applied to this code. (c) The adjoint code after running the `Activity` analysis. (d) The adjoint code after running the `Activity` and `Diff-liveness` analyses.

see figure 2.7 (c). We see in figure 2.7 (b), that w_2 and w_3 are used only to compute the two adjoint variables $\overline{w}_2$ and $\overline{w}_3$. Since every instruction that computes $\overline{w}_2$ and $\overline{w}_3$ has been removed by the `Activity` analysis, there is no reason to keep the computations of w_2 and w_3 in the forward sweep of the adjoint. Consequently, `Diff-liveness activity` detects these computations as `non-diffLive` and remove them from the differentiated code, see figure 2.7 (d).

We saw in subsection 2.2.2, that the main drawback of the adjoint code is memory consumption since it requires the storage of all intermediate variables before they are overwritten during the Forward sweep. To reduce this cost, a possible way is to store only the needed values to compute the derivatives. For instance the adjoint instructions corresponding to assignment $y = 3 * x$ are: $\overline{x} = 3 * \overline{y} + \overline{x}$; $\overline{y} = 0$. We observe that the primal variable x is not used in the adjoint instructions. There is thus no need to save its value in the case it get overwritten in the forward sweep. This is the purpose of `TBR` analysis [30], which analyses the original program to find every variable that is really used in the computation of derivatives and thus its value needs `To Be Recorded` in the case it get overwritten.

For further efficiency, `TBR` analysis must take advantage of `Activity` analysis, so as to store only the values needed to compute the derivatives of variables that are `active`.

Figure 2.8 shows the interest of `TBR` analysis. We use the same example as the one of

figure 2.6. The value of variable w overwritten by the first instruction $w = x^2$ and the value of variable y overwritten by the last instruction $y = 3 * w$ are not used in the derivatives computations. Consequently, `TBR` analysis detects that it is not necessary to store neither w before the first instruction nor y before the last instruction and thus no `PUSH/POP` instructions for these two variables have to appear in the differentiated code, see figure 2.8 (c).

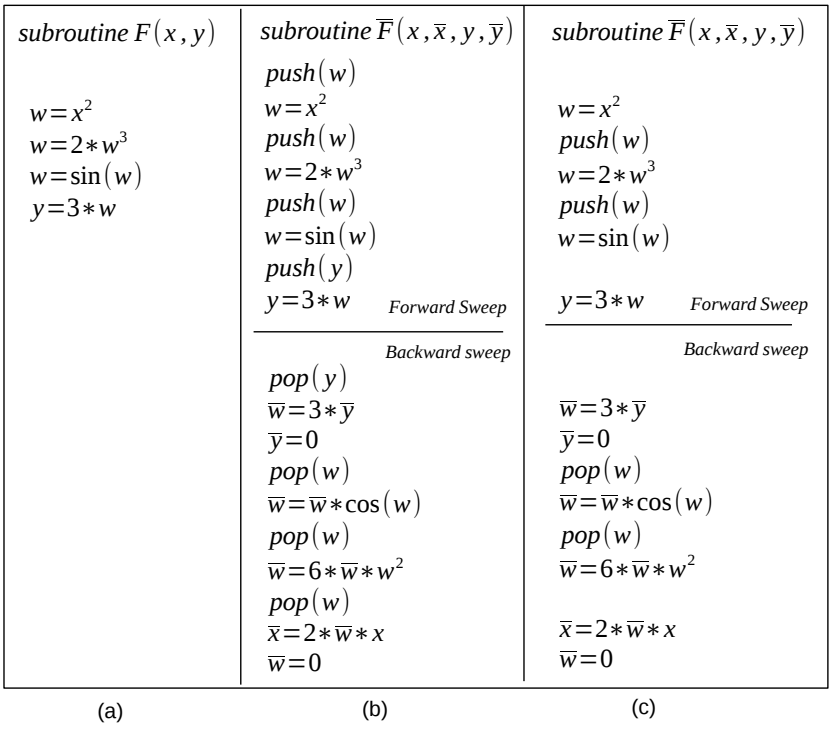

FIGURE 2.8: (a) Example of code. (b) A naive application of the adjoint mode to this code. (c) The adjoint mode applied after running `TBR` analysis

To improve the efficiency of differentiated codes, there are other AD specific analyses such as the analysis that aims to reduce the memory consumption of snapshots in the case of checkpointing [14] or the analysis run by AD tools based on Recompute-All, that applies slicing to shorten the repeated recomputation sequences [18]. The latter analysis, called `ERA`, is the counterpart of our `TBR` analysis for the Recompute-All approach.
Here we concentrate mainly on `Activity`, `Diff-liveness` and `TBR` analyses. We show how these analyses can be formalized and implemented on programs represented by Flow graphs by using the so-called "data-flow equations"[55]. Regarding notation, we write $\texttt{Info}^{-}(\texttt{I})$ each time a data-flow information `Info` is defined immediately before an instruction `I` and $\texttt{Info}^{+}(\texttt{I})$ each time `Info` is defined immediately after the instruction `I`.

2.3.2 Activity analysis

This analysis detects from the original code the set of variables that are `active`. We say that a variable v depends in a differentiable way on w when the derivative of w with respect to v is not trivially null. `Activity` analysis is a combination of forward and backward analysis. It propagates:

- forward from the beginning of the program, the set of variables that depend in a differentiable way on some independent input. These variables are called "`varied`".
- backward from the end of the program, the set of variables that influence in a differentiable way some dependent output. These variables are called "`useful`".

The variables are `active` when they are at the same time `varied` and `useful`.
In tangent mode, when a variable is not `varied` at some location in the original program, then its derivative at this location is certainly null. Conversely, when a variable is not `useful`, then its derivative does not influence the final result. Symmetrically, in the adjoint code, when a variable is not `useful` at some location in the original program, then its derivative is certainly null at this location. Conversely, when a variable is not `varied`, then its derivative does not influence the final result.
In the general case of multi-procedure codes, we must avoid combinatorial explosion of data-flow analysis in the way we have explained in section 2.3. Therefore, we must identify the summarized data-flow information that will be precomputed for each subroutine (by a bottom-up call graph sweep) and later used during the following top-down call graph sweep, at every occurrence of a subroutine call.

`Bottom-up Activity analysis`

The bottom-up analysis needed by the `Activity` analysis is called `Diff-dependency` analysis. It determines for each output of a procedure `P`, the set of inputs on which it depends in a differentiable way, noted `diffDep(P)`. To this end, this analysis propagates forward from the beginning of the program a matrix-like piece of information `diffDep` that tells, for each variable at the current location, the subset of the input variables on which it depends in a differentiable way. Given an instruction `I`, a variable v overwritten by this instruction depends in a differentiable way on some input if it depends in a differentiable way on some input of `I`, w, which depends itself in a differentiable way on some input. If the variable v is partially overwritten as in the case of arrays it will still depend on whatever it depended on before `I`. Calling $\texttt{I}_0$ and $\texttt{I}_\infty$, respectively the entry and exit instructions of the program and `Id` the identity dependence relation, the

data-flow equations are:

$$\begin{aligned} \texttt{diffDep}^{+}(\texttt{I}_0) &= \texttt{Id} \\ \texttt{diffDep}^{+}(\texttt{I}) &= \texttt{diffDep}^{-}(\texttt{I}) \otimes \texttt{diffDep}(\texttt{I}) \end{aligned}$$

in which the composition of `diffDep` objects $\otimes$ is defined as :

$$(\texttt{v}, \texttt{input}) \in \texttt{diffDep}(\texttt{A}) \otimes \texttt{diffDep}(\texttt{B}) \Longleftrightarrow$$
$$\exists \texttt{w} | (\texttt{v.w}) \in \texttt{diffDep}(\texttt{A}) \& (\texttt{w}, \texttt{input}) \in \texttt{diffDep}(\texttt{B})$$

and the result of $\texttt{diffDep}(\texttt{P})$ is found in $\texttt{diffDep}(\texttt{I}_\infty)$.
Since the `Diff-dependency` analysis is performed bottom-up on the call graph, the `diffDep` set of each subroutine is computed after that all the subroutines possibly called inside this subroutine have been computed.

`Top-down Activity analysis`

After that $\texttt{diffDep}(\texttt{P})$ set is synthesized for each procedure `P`, `Activity` analysis propagates two data-flow sets through the program:

- The `varied` variables. Given an instruction `I`, a variable resulting from this instruction is considered as `varied`, either if it depends in a differentiable way on some variable that is `varied` before `I`, or it was `varied` before `I` and it is not totally overwritten by `I`. Formally, we write:

$$\texttt{varied}^{+}(\texttt{I}) = \texttt{varied}^{-}(\texttt{I}) \otimes \texttt{diffDep}(\texttt{I})$$

- The `useful` variables. Given an instruction `I`, a variable is considered as `useful` before this instruction, either if it influences in a differentiable way some variable that is `useful` after `I`, or it was `useful` after `I` and it is not totally overwritten by `I`. Formally, we write:

$$\texttt{useful}^{-}(\texttt{I}) = \texttt{diffDep}(\texttt{I}) \otimes \texttt{useful}^{+}(\texttt{I})$$

In both data-flow equations, the composition $\otimes$ is defined as:

$$\texttt{w} \in \texttt{S} \otimes \texttt{diffDep}(\texttt{I}) \Longleftrightarrow \exists \texttt{v} \in \texttt{S} | (\texttt{v.w}) \in \texttt{diffDep}(\texttt{I})$$

and likewise in the opposite direction, i.e. $\texttt{diffDep}(\texttt{I}) \otimes \texttt{S}$.
When `I` is an assignment, $\texttt{diffDep}(\texttt{I})$ can be computed easily: the left-hand-side variable depends in a differentiable way on all the right-hand-side variables except those with

non differentiable types, e.g. integers. When the left-hand-side variable `v` is not totally overwritten by `I`, then (`v`, `v`) is set to belong to `diffDep(I)`. When `I` is a call to subroutine `P`, we use the synthesized information computed for `P`, i.e. `diffDep(P)`.
The final activity is the intersection of the two data-flow sets:

$$\begin{aligned} \texttt{active}^-(\texttt{I}) &= \texttt{varied}^-(\texttt{I}) \cap \texttt{useful}^-(\texttt{I}) \\ \texttt{active}^+(\texttt{I}) &= \texttt{varied}^+(\texttt{I}) \cap \texttt{useful}^+(\texttt{I}) \end{aligned}$$

2.3.3 Diff-liveness analysis

In the differentiated code, we may find primal instructions that are useful to compute the primal results but are not needed to compute the derivatives. Therefore, removing these instructions from the differentiated code will not affect its desired results, which are the derivatives. `Diff-liveness` analysis analyses the original program to detect which primal variables are needed in the computation of derivatives, i.e. `diffLive` variables. To avoid combinatorial explosion, we need to perform first a bottom up analysis.

`Bottom-up Diff-liveness analysis`

The bottom-up analysis needed by the `Diff-liveness` analysis is called `Dependency` analysis. It detects for each procedure `P` any dependency (including those that are not differentiable) between the outputs and inputs, i.e. `dep(P)`. The main idea is to propagate forward from the beginning of the procedure the set of variables that depend on the inputs. Given an instruction `I`, a variable v overwritten by this instruction depends on some input if it depends on some input of `I`, w, which depends itself on some input. Calling respectively $\texttt{I}_0$ and $\texttt{I}_\infty$ the entry and exit instructions of the program and `Id` the identity dependence relation, the data-flow equations are:

$$\begin{aligned} \texttt{dep}^+(\texttt{I}_0) &= \texttt{Id} \\ \texttt{dep}^+(\texttt{I}) &= \texttt{dep}^-(\texttt{I}) \otimes \texttt{dep}(\texttt{I}) \end{aligned}$$

in which the composition operation $\otimes$ is the same as the one defined in the `Diff-dependency` analysis. Only the elementary dependencies through an instruction are different. For instance, in the case of a simple assignment, the left-hand side of the assignment depends on all the variables situated at the right-hand side of this assignment. The result of `dep(P)` is found in $\texttt{dep}(\texttt{I}_\infty)$.

`Top-down Diff-liveness analysis`

After that `dep(P)` set is synthesized for each procedure P, `Diff-liveness` analysis propagates backward from the end of the program, the set of variables that are `diffLive`.

Given an instruction I, a variable is diffLive before this instruction, either because it influences a variable which is difflive after this instruction or because it is used in the derivative instruction I′. Formally, we will write:

$$\texttt{diffLive}^{-}(\texttt{I}) = \texttt{use}(\texttt{I}') \cup (\texttt{dep}(\texttt{I}) \otimes \texttt{diffLive}^{-}(\texttt{I}))$$

When I is an assignment, dep(I) can be computed easily: the left-hand-side variable depends on all the right-hand-side variables. When I is a call to subroutine P, we use the synthesized information dep(P) computed for P.
For simplicity reasons, we will not detail the data-flow equations that describe use(I′).

2.3.4 TBR analysis

This analysis aims to reduce the memory consumption of the adjoint code with store-all approach. Instead of storing the value of every variable overwritten during the forward sweep of the adjoint code, we store only those that are needed in the computation of derivatives. To this end, TBR analysis propagates forward from the beginning of the program the set of variables that are required in the adjoint code, req and flags assignments that overwrite these variables, so that their values will be recorded. After the overwriting statement, the overwritten variable is in general removed from the req set. Like before, to avoid combinatorial explosion we must identify a bottom-up analysis.

Bottom-up TBR analysis

TBR performs a bottom-up analysis called“ Killed analysis”. This analysis detects for each procedure P the set of variables that have been totally ovewritten, “killed”, by this procedure, kill(P). It propagates forward from the beginning of the procedure the set kill of all the killed variables. Given an instruction I, a variable is killed after I if it is totally overwritten by I or by one of the instructions preceding I. Formally, we write:

$$\texttt{kill}^{+}(\texttt{I}) = \texttt{kill}^{-}(\texttt{I}) \cup \texttt{kill}(\texttt{I})$$

Top-down TBR analysis

After kill(P) set is synthesized for each procedure P, TBR analysis propagates forward from the beginning of the program the set of variables that are required in the computation of derivatives req. Given an instruction I, a variable v is a part of req after I if it is used in the derivative instruction I′ or it is was a part of req before I, but it is not

totally overwritten by I. Formally, we write:

$$\mathtt{req}^{+}(\mathtt{I}) = (\mathtt{req}^{-}(\mathtt{I}) \cup \mathtt{use}(\mathtt{I}')) \setminus \mathtt{kill}(\mathtt{I})$$

When I is a call to subroutine P, we use the synthesized information computed for P, i.e. kill(P). For simplicity reasons, we will not detail the data-flow equations that describe use(I′).
While propagating the set of required variables, each time an individual instruction overwrites a required variable, we flag the overwritten variable as "To Be Recorded", and a PUSH/POP pair will be inserted.

2.3.5 Termination issue

In all these analyses, solutions must be obtained iteratively if the call graph or the flow graph contains cycles. Therefore, it is necessary to make sure that the Fixed-Point resolution terminates. Abstract interpretation [12] gives the general framework for this. The idea of the proof is that at each location in the code, the currently detected value of the analyzed property is growing during the successive iterations of the iterative process. Since this property value belongs to a set of possible values which is finite and forms a lattice, the iterative process must reach a global fixed point in a finite number of steps.

2.4 Algorithmic differentiation tool: Tapenade

Tapenade [31] is an Automatic Differentiation tool developed by our research team. Given a FORTRAN or C source program, it generates the derivatives of this program, in tangent or adjoint mode. Tapenade has two principal objectives:

- To serve as a platform to experiment and validate refinements on the adjoint AD mode.
- To be used on real-size applications, yet providing the benefits of the latest AD refinements.

To meet these objectives, several design choices have been made, in particular:

- **Source transformation:** The main focus of our research is on the adjoint mode. Since source transformation is the best choice for this mode, see subsection 2.2.2, we decided to use this approach.

- **Store-All in adjoint mode:** For the adjoint mode, Tapenade uses the Store-All approach to recover intermediate values.

- **Context-sensitive and Flow-sensitive Data-Flow analysis:** Accurate data-flow analyses must be context-sensitive and Flow-sensitive. Therefore, instead of syntax trees, the internal representation is a Call graph of flow graphs.

- **Source language independence:** The internal representation must concentrate on the semantics of the program and thus be independent of its particular programming language. Therefore, the original program has to be expressed in terms of an intermediate language, called IL for Imperative Language. IL is an abstract language with no concrete syntax, i.e. no textual form. This language must be rich enough to represent all imperative programming constructs, such as procedure definition, variable declaration, procedure calls . . . , including Object Oriented constructs for future extension of the tool. Whenever possible, an IL operator must represent similar constructs of different languages.
  Thanks to this design choice, Tapenade is able not only to differentiate codes that are written in C or Fortran but also codes that mix both languages [45].

- **Readability:** In the differentiated code, the end-user should be able to recognize the structure of the original code. Therefore, the internal representation should keep information that a classic compiler may discard, for instance the order of instructions in the original code. This information is used during the generation of the new code.

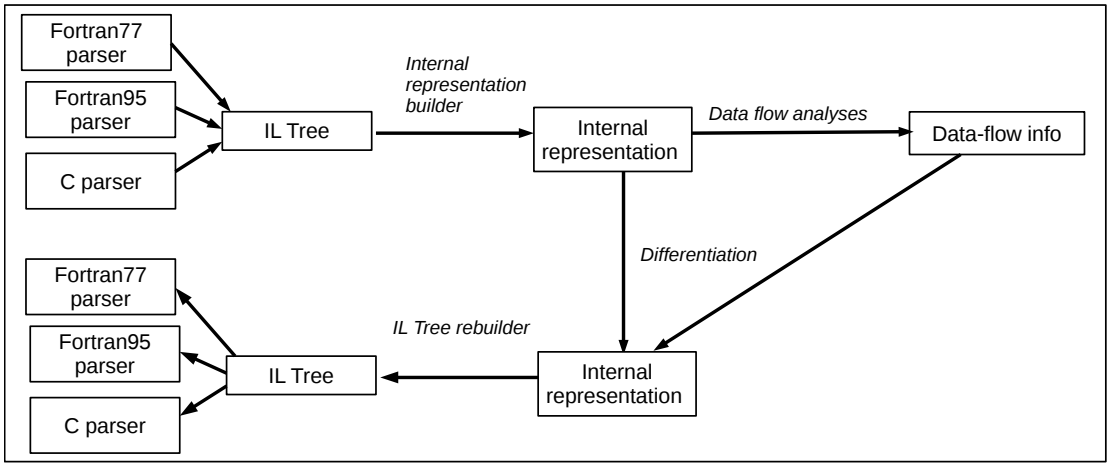

FIGURE 2.9: General architecture of Tapenade

The architecture of Tapenade resembles that of a classical compiler, building an internal representation of the original program and performing data-flow analyses on it, see figure 2.9. A big difference, however, is that Tapenade produces its results in the language

of the original program instead of machine code. This imposes additional constraints to keep some degree of resemblance between the original code and the differentiated one. For instance, Tapenade saves the order of declarations inside the original program so that it can regenerate these declarations in the same order in the differentiated code. Another important difference is that the data-flow analyses performed by Tapenade have to be global and thus no separate compilation has to be made, i.e. all the original code has to be parsed and then analyzed jointly. Examples of data-flow analyses run by Tapenade on the original program [3] are: `Read-Written` analysis, `Activity` analysis, `TBR` analysis and `Diff-liveness` analysis.

2.5 Organization

The adjoint algorithms obtained through the adjoint mode of AD are probably the most efficient way to obtain the gradient of a numerical simulation. This however needs to use the data-flow of the original simulation in reverse order, at a cost that increases with the length of the simulation. In the context of the AboutFlow project that has funded this research, our industrial partners have submitted several large application codes for which this data-flow reversal may have a prohibitive cost. The goal of this thesis is to further study the techniques that help keep this cost acceptable. In collaboration with the partners, two such techniques, related to the checkpointing mechanism, have been selected.

- In chapter 3, we consider the adjoint of Fixed-Point loops, for which several authors have proposed adapted adjoint strategies. Among these strategies, we select the one introduced by B. Christianson. This method features original mechanisms such as repeated access to the trajectory stack or duplicated differentiation of the loop body with respect to different independent variables. We describe how the method must be further specified to take into account the particularities of real codes, and how data flow information can be used to automate detection of relevant sets of variables. We describe the way we proceeded to implement this strategy in our AD tool. Experiments on a medium-size application demonstrate a minor, but non negligible improvement of the accuracy of the result, and more importantly a major reduction of the memory needed to store the trajectories.

- In chapter 4, we address the question of checkpointing applied to adjoint MPI parallel programs. On one hand we propose an extension of checkpointing in the case of MPI parallel programs with point-to-point communications, so that the semantics of an adjoint program is preserved for any choice of the checkpointed

piece of code. On the other hand, we propose an alternative extension of checkpointing, more efficient but that requires a number of restrictions on the choice of the checkpointed piece. We try to provide proof of correctness of these strategies, and in particular demonstrate that they cannot introduce deadlocks. Trade-offs between the two extensions should be investigated. We propose an implementation of these strategies inside the AMPI library. We discuss practical questions about the choice of strategy to be applied within a checkpointed piece and the choice of the checkpointed piece itself. At the end, we validate our theoretical results on representative CFD codes.

Chapter 3

An efficient Adjoint of Fixed-Point Loops

3.1 Introduction

Exploiting knowledge of the algorithm and of the structure of the given simulation code can yield a huge performance improvement in the adjoint code. In Tapenade, special strategies are already available for parallel loops [29], long unsteady iterative loops, etc. We focus here on the case of Fixed-Point loops which are loops that iteratively refine a value until it becomes stationary. We call "state" the variable that holds this value and "parameters" the set of variables used to compute it.

As Fixed-Point algorithms start from some initial guess for the state, one intuition is that at least the first iterations are almost meaningless. Therefore, storing them for the adjoint computation is a waste of memory. Furthermore, Fixed-Point loops that start with an initial guess almost equal to the final result converge only in a few iterations. As the adjoint loop of the standard AD adjoint code runs for exactly the same number of iterations, it may return a gradient that is not converged enough. For these reasons we looked for a specific adjoint strategy for Fixed-Point loops. Among the strategies documented in literature, we selected the Piggyback, Delayed Piggyback, Blurred Piggyback, Two-Phases and Refined Two-Phases approaches. These special adjoints manage to avoid naive inversion of the original sequence of iterations, therefore saving the cost of data-flow reversal. The difference between these approaches is mainly when starting the adjoint computations. Some of these approaches start adjoining since the first iterations of the original loop, as in the case of Piggyback approach, some of them wait until that the state becomes sufficiently converged, as in the case of Delayed Piggyback and Blurred Piggyback and some others compute the adjoint only when

the state has fully converged, as in the case of Two-Phases and refined Two-Phases approaches. Among these strategies, we select the one we find the best suited to be implemented in our AD tool.

In section 3.2, we examine in more detail these methods, their strengths and their weaknesses. We introduce also a method that combines the Black Box and Two-Phases approaches. We call this method "Refined Black Box". In section 3.3, we compare between some of the special FP adjoints and we select the one that will be implemented in our AD tool. In section 3.4, we specify further the selected method in order to take into account particular structures that occur in real codes such as loops with multiple exits. In section 3.5, we focus on the practical implementation of the selected adjoint strategy. We describe how the various variables needed by the adjoint can be automatically detected by using the data flow analysis of our AD tool. We describe the way we extended the standard stack mechanism and the way we implemented the special selected strategy in our tool Tapenade. In section 3.6 we show how checkpointing may reduce the efficiency of the selected strategy. Finally, in section 3.7 we experiment our implemented strategy on a real medium size code as well as a representative code that contains nested structure of Fixed-Point loops.

3.2 Existing methods

Many equations having the form $F(z_*, x) = 0$ may be solved by using iterative methods [34] that satisfy some Fixed-Point equation having the form $z_* = \phi(z_*, x)$ with x is some fixed parameter and z_* is an attractive Fixed Point of ϕ, i.e. $||\frac{\partial}{\partial z}\phi(z, x)|| < 1$ with z is in a neighborhood of z_*. These iterative methods, called Fixed-Point (FP) loops, initialize the state $\mathtt{z}$ with some value called "initial guess" $\mathtt{z_0}$, then iteratively call $\mathtt{z_{k+1}} = \phi(\mathtt{z_k}, \mathtt{x})$ until meeting some stopping criterion that expresses that z has reached the fixed point of the function $\phi(z, x)$, i.e. z is almost equal to z_*. This fixed point is used after that to compute some final result $y = f(z_*, x)$. An example of FP loops is sketched in figure 3.1 (a).

The stopping criterion of a FP loop can be written in different ways. It can test for instance the stationarity of z, i.e. it tests if $||z_{k+1} - z_k|| \leq \epsilon$ or it can simply check that z is the desired solution for $F(z, x)$, i.e. it tests if $||F(z, x)|| \leq \epsilon$.

In general, the choice of the initial guess z_0 is made in arbitrary way. However, z_0 has to be in the contraction basin of $\phi(z, x)$, i.e. z_0 verifies:
$||z_* - \phi(z_0, x)|| < ||z_* - z_0||$.

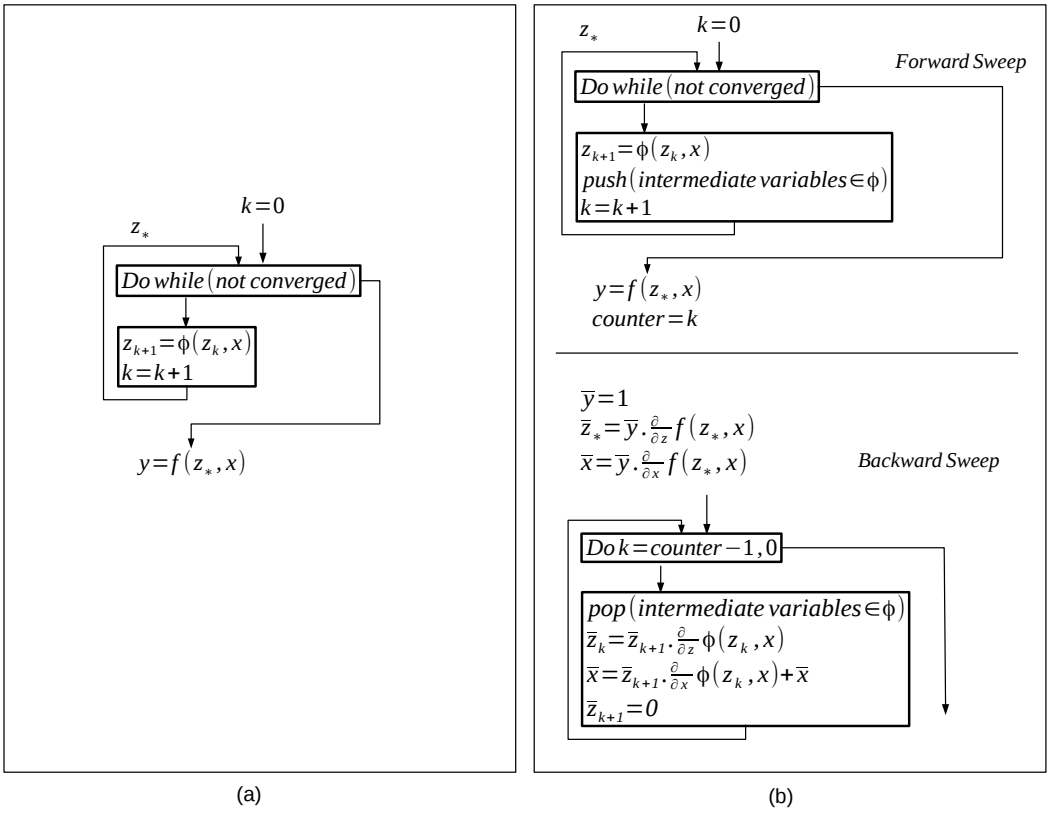

FIGURE 3.1: (a) An example of code containing a FP loop. (b) The Black Box approach applied to this code

3.2.1 Black Box approach

The Black Box approach (called also the "Brute Force") is the standard adjoint mode applied to the FP loop in a mechanical fashion, i.e. without taking into account its specific structure. In the Store-All approach, the Black Box adjoint consists in two successive sweeps (see figure 3.1 (b)): A forward sweep that contains a copy of the original loop, i.e. a loop that initiates $\mathtt{z}$ with some initial guess $\mathtt{z_0}$, then, iteratively calls:

$$\mathtt{z_{k+1}} = \phi(\mathtt{z_k}, \mathtt{x})$$

until reaching some fixed point z_*. A backward sweep that contains another loop (called the "adjoint loop") that follows exactly the same number of iterations as the original loop. The adjoint loop iteratively calls the adjoint of $\mathtt{z_{k+1}} = \phi(\mathtt{z_k}, \mathtt{x})$, which can be written as:

$$\overline{\mathtt{z}}_\mathtt{k} = \overline{\mathtt{z}}_{\mathtt{k+1}}.\frac{\partial}{\partial \mathtt{z}}\phi(\mathtt{z_k}, \mathtt{x})$$

$$\overline{\mathtt{x}} = \overline{\mathtt{z}}_{\mathtt{k+1}}.\frac{\partial}{\partial \mathtt{x}}\phi(\mathtt{z_k}, \mathtt{x}) + \overline{\mathtt{x}}$$

We may note that the value of $\overline{\mathtt{x}}$ and $\overline{\mathtt{z}}$ depend on the value of `z` at each iteration. As every iteration of the FP loop overwrites the intermediate values computed at the previous iteration, a mechanism has to be used in order to retrieve the values of `z` in reverse order. In the Store-All approach, we store each intermediate value `z` onto a stack during the forward sweep of the adjoint and then retrieve this value when needed during the backward sweep, see figure 3.1 (b). The needed push and pop primitives are provided by a separate library.
Assuming that we need n iterations to converge the original FP loop, the $\overline{x}$ returned by the Black Box adjoint is actually equal to:

$$\overline{x} = \overline{x}_0 + \overline{z}_0.\frac{\partial}{\partial x}\phi(z_*, x) + \overline{z}_0.\frac{\partial}{\partial z}\phi(z_{n-1}, x).\frac{\partial}{\partial x}\phi(z_{n-1}, x) +$$
$$\overline{z}_0.[\frac{\partial}{\partial z}\phi(z_{n-2}, x)]^2.\frac{\partial}{\partial x}\phi(z_{n-2}, x) + ... + \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_0, x)]^n.\frac{\partial}{\partial x}\phi(z_0, x)$$

with $\overline{x}_0$ and $\overline{z}_0$ resulting from the adjoint of the function f, i.e. $\overline{x}_0 = \overline{y}.\frac{\partial}{\partial x}f(z_*, x)$ and $\overline{z}_0 = \overline{y}.\frac{\partial}{\partial z}f(z_*, x)$.

`Strengths and weaknesses:`

The main advantage of this approach is its generality since it can be applied to any structure of FP loops. Also, this approach is relatively easy to apply as it does not require a big understanding of the mathematical background of a given code.
However, this approach is memory costly, i.e. it saves the intermediate values of z at every iteration of the FP loop. Moreover, this approach does not take into account the convergence of the adjoint. The adjoint follows exactly the same number of iterations as the original loop. It was shown in [20] that whenever the original FP loop converges, its adjoint with the Black Box approach will converge as well. However, the convergence of the derivatives will not reach the same tolerance as the one of the original values. In the case where the original loop needs only a few iterations to converge, for instance when the initial guess of the original loop is very close to the final solution, the adjoint loop may return derivatives that are not well converged.
This approach is not efficient (in terms of time and memory) in the case of FP loops with superlinear convergence, e.g. Newton, since the adjoint of these loops amounts to solution of a single linear system [43].

3.2.2 Piggyback approach

Unlike the Black Box method, the Piggyback approach developed by Griewank [23, 24] observes that the adjoint loop needs not follow exactly the same number of iterations as

the original loop. The adjoint can be a fixed-point loop itself with its own initial guess as well as its own stopping criterion.

Let us consider the system:

$$w_* = F(z_*(x), x) = 0 \tag{3.1}$$

$$y = f(z_*(x), x) \tag{3.2}$$

in which, x represents the parameters of this system and y is the desired solution.

`Tangent and Adjoint Sensitivity Equations:`

Applying the chain rule to this system gives us the Jacobian $\frac{dy}{dx}$ which can be expressed as:

$$\frac{dy}{dx} = \frac{\partial}{\partial x} f(z_*, x) - \frac{\partial}{\partial z} f(z_*, x).\frac{\partial}{\partial z} F(z_*, x)^{-1}.\frac{\partial}{\partial x} F(z_*, x) \tag{3.3}$$

It is too expensive to compute the whole Jacobian. Therefore, in practice we compute rather one of these two projections:

$$\frac{dy}{dx}.\dot{x} \qquad or \qquad \overline{y}.\frac{dy}{dx}$$

where $\dot{x}$ is a vector and $\overline{y}$ is a row-vector. These projections are computed by the so-called tangent and adjoint modes of AD, see chapter 2.

The tangent mode computes $\dot{y} = \frac{dy}{dx}.\dot{x}$. Recalling equation 3.3 we obtain:

$$\dot{y} = \frac{\partial}{\partial x} f(z_*, x).\dot{x} - \frac{\partial}{\partial z} f(z_*, x).\frac{\partial}{\partial z} F(z_*, x)^{-1}.\frac{\partial}{\partial x} F(z_*, x).\dot{x}$$

The tangent code of the Black Box approach evaluates this equation from right to left. First, this code computes an intermediate value $\dot{z}_*$ so that:

$$\dot{z}_* = -\frac{\partial}{\partial z} F(z_*, x)^{-1}.\frac{\partial}{\partial x} F(z_*, x).\dot{x} \tag{3.4}$$

Then, it uses the value of $\dot{z}_*$ to compute $\dot{y}$, i.e.

$$\dot{y} = \frac{\partial}{\partial x} f(z_*, x).\dot{x} + \frac{\partial}{\partial z} f(z_*, x).\dot{z}_*$$

Equation 3.4 can also be written as:

$$\dot{F}(z_*, x, \dot{z}_*, \dot{x}) = \dot{w}_* = \frac{\partial}{\partial z}F(z_*, x).\dot{z}_* + \frac{\partial}{\partial x}F(z_*, x).\dot{x} = 0 \tag{3.5}$$

This equation is called the *tangent sensitivity equation.*

Symmetrically, the adjoint mode computes $\overline{x} = \overline{y}.\frac{dy}{dx}$. Recalling equation 3.3 we obtain:

$$\overline{x} = \overline{y}.\frac{\partial}{\partial x}f(z_*, x) - \overline{y}.\frac{\partial}{\partial z}f(z_*, x).\frac{\partial}{\partial z}F(z_*, x)^{-1}.\frac{\partial}{\partial x}F(z_*, x)$$

The adjoint code of the Black Box approach evaluates this equation from left to right. First, this code computes an intermediate value $\overline{w}_*$ so that:

$$\overline{w}_* = -\overline{y}.\frac{\partial}{\partial z}f(z_*, x).\frac{\partial}{\partial z}F(z_*, x)^{-1} \tag{3.6}$$

Then, it uses the value of $\overline{w}_*$ to compute $\overline{x}$, i.e.

$$\overline{x} = \overline{y}.\frac{\partial}{\partial x}f(z_*, x) + \overline{w}_*.\frac{\partial}{\partial x}F(z_*, x) \tag{3.7}$$

Equation 3.6 can also be written as:

$$\overline{F}(z_*, x, \overline{w}_*, \overline{y}) = \overline{z}_* = \overline{w}_*.\frac{\partial}{\partial z}F(z_*, x) + \overline{y}.\frac{\partial}{\partial z}f(z_*, x) = 0 \tag{3.8}$$

This equation is called the *adjoint sensitivity equation.*

`Piggyback approach:`

Griewank observes that the majority of FP loops that solve the equation 3.1 satisfy a FP equation of the form:

$$z_* = z_* - P_k.F(z_*, x)$$

where P_k is some preconditioner that approximates the inverse of the Jacobian $F'(z_k, x)$ and that verifies $||I - P_k.\frac{\partial}{\partial z}F(z_*, x)|| \leq \rho_0 < 1$. The closer the preconditioner is to $F'(z_k, x)^{-1}$, the more the Fixed-Point equation resembles to Newton's method with its excellent local convergence properties, i.e. Newton's method is known to have a quadratic convergence to the solution z_*. In the following we assume that for all arguments (z, x) in some neighborhood of (z_*, x) we have :

$$||I - P_k.\frac{\partial}{\partial z}F(z, x)|| \leq \rho_0 < 1$$

Griewank observes also that $||F(z_k, x)||$ is equivalent to the norm of the solution error $||z_k - z_*||$. Therefore, a good stopping criterion for the FP loop will check at each iteration if $||F(z_k, x)||$ is sufficiently close to zero. In practice, the stopping criterion will test if $\frac{||F(z_k,x)||}{||F(z_0,x)||} \leq \epsilon$.

Applying the tangent mode of the Black Box approach to the FP loop leads to a loop that iteratively calls:

$$\mathtt{z_{k+1}} = \mathtt{z_k} - \mathtt{P_k.F(z_k, x)}$$

$$\dot{\mathtt{z}}_{\mathtt{k+1}} = \dot{\mathtt{z}}_{\mathtt{k}} - \mathtt{P_k}.\dot{\mathtt{F}}(\mathtt{z_k}, \mathtt{x}, \dot{\mathtt{z}}_{\mathtt{k}}, \dot{\mathtt{x}}) - \dot{\mathtt{P}}_{\mathtt{k}}.\mathtt{F}(\mathtt{z_k}, \mathtt{x})$$

until convergence of $||F(z_k, x)||$ to zero.

Let us define $\dot{\phi}$ as a function that satisfies for all (z_k, x) in some neighborhood of (z_*, x), the equation $\dot{\phi}(z_k, x, \dot{z}_k, \dot{x}) = \dot{z}_k - P_k.\dot{F}(z_k, x, \dot{z}_k, \dot{x}) - \dot{P}_k.F(z_k, x)$. We denote $\dot{z}_*$ the fixed point of the function $\dot{\phi}$, so that $\dot{z}_* = \dot{\phi}(z_k, x, \dot{z}_*, \dot{x})$.
We compute the derivative of $\dot{\phi}$ with respect to $\dot{z}$ at the fixed point $\dot{z}_*$ with the arguments (z, x) in some neighborhood of (z_*, x). We obtain:

$$\frac{d}{d\dot{z}}\dot{\phi}(z, x, \dot{z}_*, \dot{x}) = I - P_k.\frac{d}{d\dot{z}}\dot{F}(z, x, \dot{z}_*, \dot{x}) = I - P_k.\frac{\partial}{\partial z}F(z, x)$$

As $||\frac{d}{d\dot{z}}\dot{\phi}(z, x, \dot{z}_*, \dot{x})|| = ||I - P_k.\frac{\partial}{\partial z}F(z, x)|| \leq \rho_0 < 1$, $\dot{z}_*$ is an attractive fixed point of the function $\dot{\phi}$. Consequently, the equation $\dot{z}_* = \dot{\phi}(z_k, x, \dot{z}_*, \dot{x})$ may be solved by an iterative method that repeatedly calls :

$$\dot{\mathtt{z}}_{\mathtt{k+1}} = \dot{\mathtt{z}}_{\mathtt{k}} - \mathtt{P_k}.\dot{\mathtt{F}}(\mathtt{z_k}, \mathtt{x}, \dot{\mathtt{z}}_{\mathtt{k}}, \dot{\mathtt{x}}) - \dot{\mathtt{P}}_{\mathtt{k}}.\mathtt{F}(\mathtt{z_k}, \mathtt{x}) \tag{3.9}$$

until that $\dot{z}$ reaches the fixed point $\dot{z_*}$.
As the tangent loop already calls 3.9 repeatedly, it suffices to change the stopping criterion of the tangent loop to express the convergence of not only $||F(z_k, x)||$ but also $||\dot{F}(z_k, x)||$.
For simplicity, we choose to omit from the instruction 3.9 the term $\dot{\mathtt{P}}_{\mathtt{k}}.\mathtt{F}(\mathtt{z_k}, \mathtt{x})$ as it disappears gradually as $F(z_k, x)$ converges to zero. Hence, the tangent derivatives are computed now by a FP loop, we call it tangent loop, that iteratively calls:

$$\mathtt{z_{k+1}} = \mathtt{z_k} - \mathtt{P_k.F(z_k, x)}$$

$$\dot{\mathtt{z}}_{\mathtt{k+1}} = \dot{\mathtt{z}}_{\mathtt{k}} - \mathtt{P_k}.\dot{\mathtt{F}}(\mathtt{z_k}, \mathtt{x}, \dot{\mathtt{z}}_{\mathtt{k}}, \dot{\mathtt{x}})$$

until convergence of both $||F(z_k, x)||$ and $||\dot{F}(z_k, x, \dot{z}_k, \dot{x})||$ to zero.

The objective of the tangent loop is to compute the *tangent sensitivity equation* 3.5. Symmetrically, the objective of the adjoint FP loop is to compute the *adjoint sensitivity equation* 3.8. Transposing the *adjoint equation*, we obtain:

$$\overline{F}(z_*, x, \overline{w}_*, \overline{y})^T = \overline{z}_*^T = \frac{\partial}{\partial z}F(z_*, x)^T.\overline{w}_*^T + \frac{\partial}{\partial z}f(z_*, x).\overline{y}^T \tag{3.10}$$

The transposed Jacobian $\frac{\partial}{\partial z}F(z_*, x)^T$ has the same size, spectrum, and sparsity characteristics as $\frac{\partial}{\partial z}F(z_*, x)$ itself. Hence, solving the *adjoint sensitivity equation* 3.10 is almost equivalent to solving the *tangent sensitivity equation* 3.5. Furthermore, the square matrices $I - P_k.\frac{\partial}{\partial z}F(z, x)$ and $I - P_k^T.\frac{\partial}{\partial z}F(z, x)^T$ have the same spectrum, thus, for all arguments (z, x) in some neighborhood of (z_*, x) we have :

$$\rho(I - P_k^T.\frac{\partial}{\partial z}F(z, x)^T) \leq ||I - P_k.\frac{\partial}{\partial z}F(z, x)|| \leq \rho_0 < 1$$

where ρ is the spectrum radius.

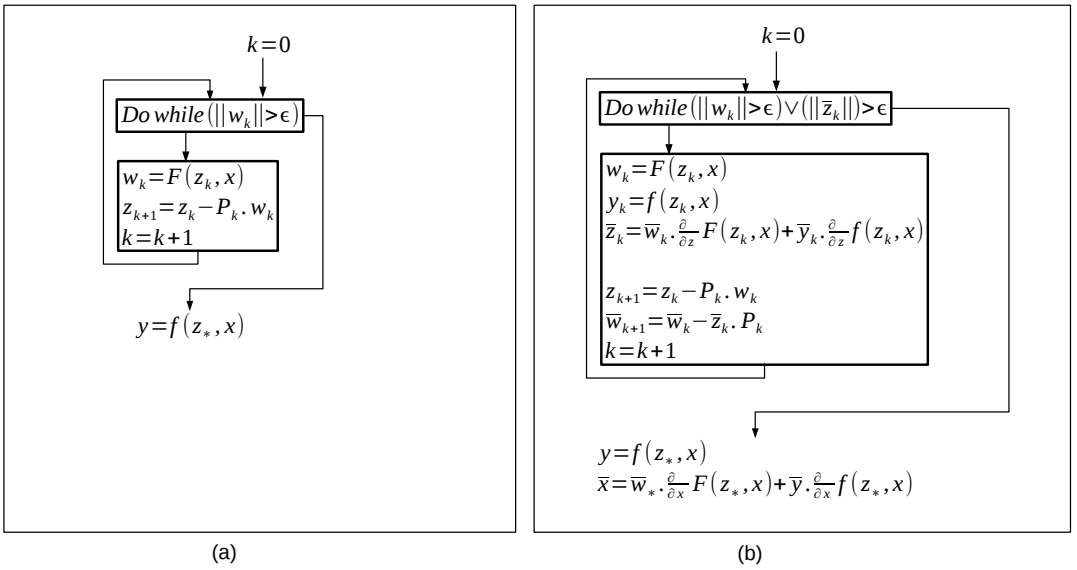

FIGURE 3.2: (a) An example of code containing a FP loop. (b) The Piggyback approach applied to this code

Therefore, by analogy with what has been done in the tangent, we may compute the adjoint derivatives by using a FP loop that iteratively calls:

$$\mathtt{z_{k+1}} = \mathtt{z_k} - \mathtt{P_k}.\mathtt{F}(\mathtt{z_k}, \mathtt{x})$$

$$\overline{\mathtt{w}}_{\mathtt{k+1}}^{\mathtt{T}} = \overline{\mathtt{w}}_{\mathtt{k}}^{\mathtt{T}} - \mathtt{P}_{\mathtt{k}}^{\mathtt{T}}.\overline{\mathtt{F}}(\mathtt{z_k}, \mathtt{x}, \overline{\mathtt{w}}_{\mathtt{k}}, \overline{\mathtt{y}})^{\mathtt{T}}$$

where $\overline{\mathtt{F}}(\mathtt{z_k}, \mathtt{x}, \overline{\mathtt{w}}_{\mathtt{k}}, \overline{\mathtt{y}})^{\mathtt{T}} = \frac{\partial}{\partial \mathtt{z}}\mathtt{F}(\mathtt{z_k}, \mathtt{x})^{\mathtt{T}}.\overline{\mathtt{w}}_{\mathtt{k}}^{\mathtt{T}} + \frac{\partial}{\partial \mathtt{z}}\mathtt{f}(\mathtt{z_k}, \mathtt{x}).\overline{\mathtt{y}}^{\mathtt{T}}$ until meeting some stopping criterion that expresses the convergence of both $||F(z_k, x)||$ and $||\overline{F}(z_k, x, \overline{w}_k, \overline{y})^T||$ to zero. Then, we use the converged value of $\overline{w}$, $\overline{w}_*$, to compute $\overline{x}$ by solving equation 3.7.

It was proven in [24] that the convergence rate of the adjoint FP loop is similar to the convergence rate of the FP loop itself. However, since Piggyback is computed by using values of z that are still not converged, the adjoint may require a few more iterations than the original FP loop.
Figure 3.2 shows an application of the Piggyback approach to a FP loop that satisfies a FP equation of the form $z_{k+1} = z_k - P_k.F(z_k, x)$. One may observe that the sequel of the FP loop `f` as well as its adjoint are computed inside the adjoint FP loop. Also, as the adjoint vectors $\overline{\mathtt{w}}_\mathtt{k}$ are computed in the same order as the original values, saving the values of `z` (and also the values of intermediate variables used to compute `z`) at each iteration is not needed any more.

`Stopping criterion:`

As the adjoint values are computed in the same order as the original ones, the induced norm used in the test of convergence of the adjoint values is the same as the one used to test the convergence of the original values.
Griewank observes that $||F(z,x)||$ is equivalent to the norm of the solution error $||z-z_*||$. Therefore, a good stopping criterion for the FP loop will check at each iteration the convergence of $||F(z,x)||$ to zero. Similarly, $||\overline{F}(z,x,\overline{w},\overline{y})^T|| + ||F(z,x)||$ is equivalent to the norm of the solution error $||\overline{w}^T - \overline{w}_*^T||$. Consequently, a good stopping criterion for the adjoint FP loop will check at each iteration the convergence of both $||\overline{F}(z,x,\overline{w},\overline{y})^T||$ and $||\overline{F}(z,x)||$ to zero.

`Strengths and weaknesses:`

The main advantage of Piggyback method is its efficiency in terms of memory consumption, i.e. it does not require the storage of z at each iteration of the FP loop as it is the case of the Black Box adjoint.
Also, the adjoint of the FP loop is a FP loop that takes into account the convergence of the adjoint values. In the case where the original values need only a few iterations to converge, e.g. when they start from a good initial guess, the adjoint loop will perform extra iterations to converge the adjoint values.
Another advantage is the fact that the derivative of ϕ with respect to the parameters x is calculated only once outside the adjoint loop which may reduce the computation time of the adjoint derivatives.
Since Piggyback computes the gradients in the same order as the original values, the resulting adjoint can be implemented as a parallel program, see subsection 3.2.4. Therefore, Piggyback can also be very efficient in terms of time.

As weaknesses of this approach, the stopping criterion of Piggyback combines the test of convergence of the original values with that of the adjoint values. Consequently, neither

the original values, nor the adjoint ones can take advantage from the fact that they may have a good initial guess. For instance, if the original values have an initial guess almost equal to the final solution and therefore they need only a few iterations to converge, the adjoint loop will perform extra iterations to converge the adjoint values. During these extra iterations, the adjoint loop will continue to compute the original values which may be considered as waste of execution time. Symmetrically, even if the adjoint values start from a good initial guess, the adjoint loop will iterate as the original values are not yet converged.
Piggyback makes an assumption on the shape of the FP loop, i.e. it requires that the original loop satisfies the FP equation of the form $z_{k+1} = z_k - P_k.F(z_k, x)$.
Also, it changes the two sweeps structure of the Black Box approach which makes the implementation of this method delicate inside an AD tool.
As the gradients are computed in the same order as the original values, this method needs to compute the sequel of the FP loop f as well as its adjoint inside the adjoint FP loop which may have a significant cost when the sequel is complex or when the FP loops are nested.
Also, one of the weaknesses of Piggyback approach is that it starts adjoining very early, i.e. it computes the adjoint by using the first computed values of the state z. This makes sometimes the adjoint diverge during the first iterations of the adjoint loop. We call this the "adjoint lag effect" of the Piggyback approach. We discuss this further in subsection 3.3.2.

3.2.3 Delayed Piggyback approach

Delayed Piggyback [25] approach is a refinement of Piggyback seen in subsection 3.2.2. It consists in applying Piggyback after that the original FP loop has been "sufficiently" converged to the solution z_*. Actually, it is not very beneficial to compute the adjoint as long as the values of z are still far from any particular solution. The very early values of the z may sometimes make the adjoint diverge, see subsection 3.3.2. Thus, it makes sense to wait for the original values z to gain some stationarity, before actually computing the derivatives values.

Algorithmically, Delayed Piggyback consists in two sweeps (see figure 3.3) . The first one copies the original loop with a small modification on the stopping criterion, i.e. it expresses that the values of z have "sufficiently" converged. Then, the second sweep applies the Piggyback method.

`Stopping criterion:`

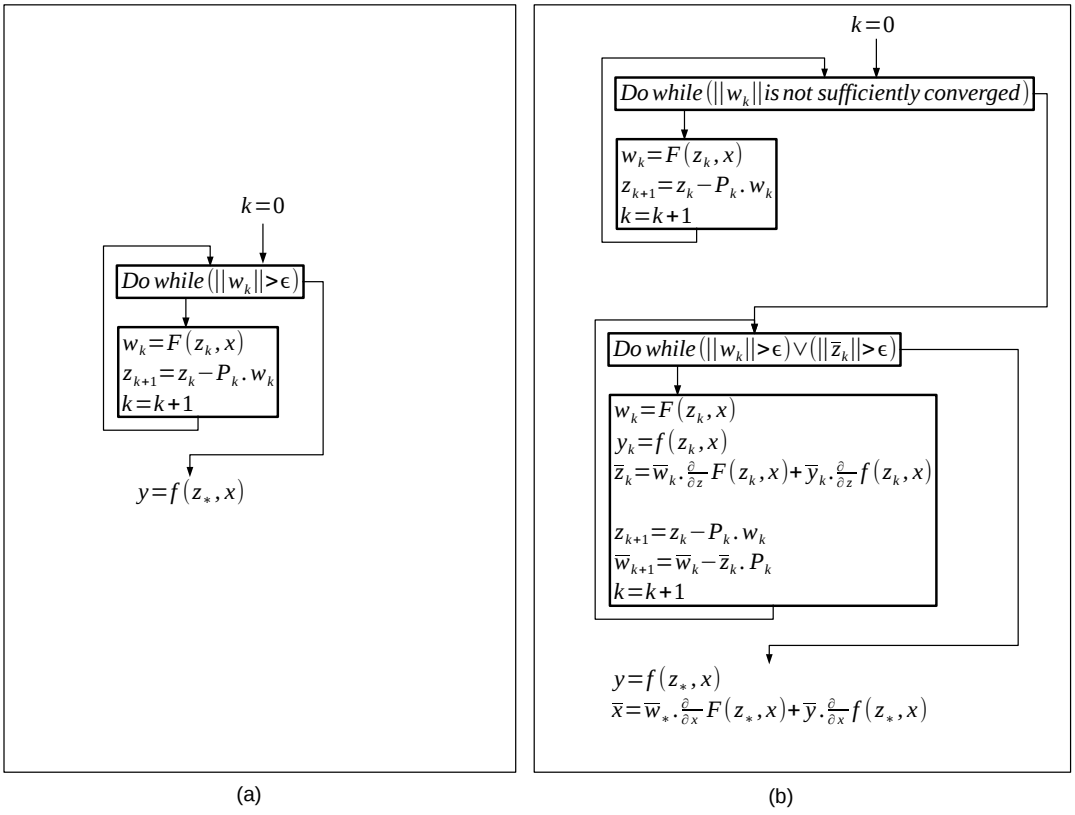

FIGURE 3.3: (a) An example of code containing a FP loop. (b) The Delayed Piggyback approach applied to this code

The stopping criterion of the first loop has to express that the values of z have "sufficiently" converged. To do so, one may use for instance the same stopping criterion as the original loop with a small modification on the value of ϵ, i.e. we use a new ϵ that is greater than the one of the original loop. Since, the second loop applies Piggyback, its stopping criterion checks at each iteration if $||\overline{F}(z, x, \overline{w}, \overline{y})^T||$ and $||F(z, x)||$ have converged to zero.

`Strengths and weaknesses:`

Delayed Piggyback has the same strengths and weaknesses as non-refined Piggyback. Since this approach computes the adjoint by using only values of the state that are sufficiently close to the solution, this will on one hand reduce the computation time of the adjoint derivatives and on the other hand reduce the adjoint lag effect in the first iterations (this will be described further in subsection 3.3.2).

3.2.4 Blurred Piggyback approach

Blurred Piggyback, originally proposed by T.Bosse [4], is another refinement of the Piggyback approach seen in subsection 3.2.2. This approach is mostly used in One-shot

optimization methods, where at each step of the iterative process, we converge the forward and reverse solutions and also adjust the design parameters. We saw in Piggyback that the adjoint values are computed in the same order as the original values. Since the adjoint values do not depend on the original values of the same iteration, i.e. they depend only on the original values of the previous iteration, one may implement the adjoint as a parallel program, i.e. runs two processes : one computes the original values (we call it "original process") and the other computes the adjoint values (we call it "adjoint process") . At the end of each iteration of the adjoint loop, the original process sends the computed original values to the adjoint process. In general the adjoint values require more computation time than the original ones. Therefore, at the end of each iteration, the original process has to wait for the adjoint process to receive the original value that have been sent before actually starting the computations of the next iteration.

To reduce further the computation time, Blurred Piggyback proposes to run the two processes in an asynchronous way (see figure 3.4). Instead of waiting for the adjoint process, the original process saves the computed values of the current iteration in some temporary storage and then starts the computations of the next iteration. The temporary storage holds each time the last computed values, i.e. at the end of each iteration, the values of the temporary storage are ovewritten by the new computed original ones. From its side, the adjoint process uses the values of the temporary storage to compute the derivatives.

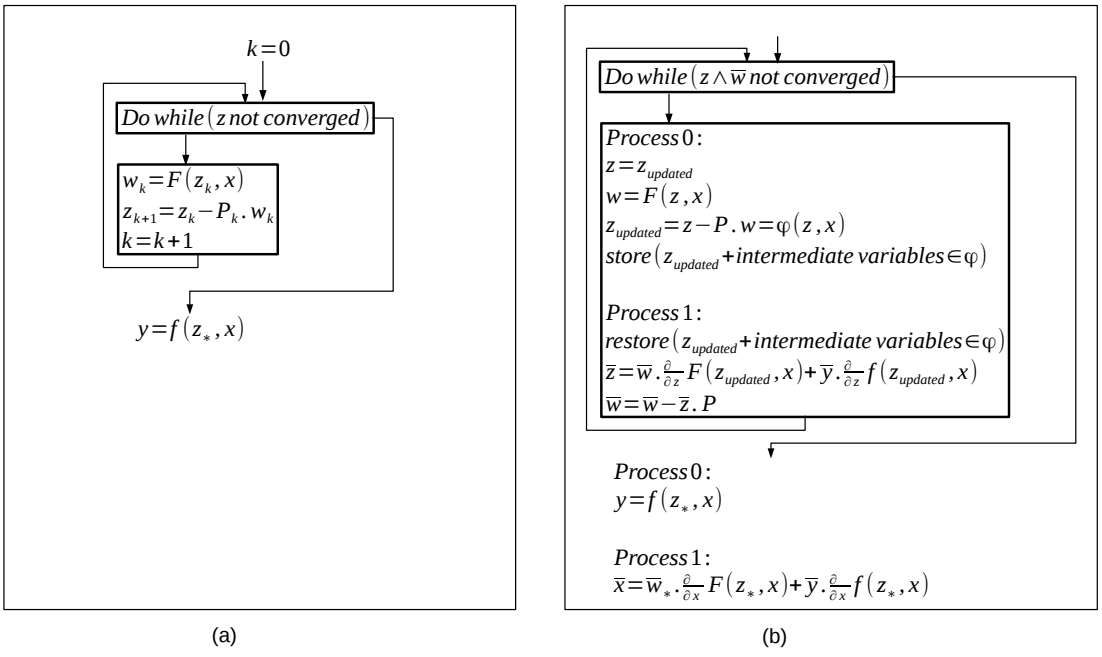

FIGURE 3.4: (a) An example of code containing a FP loop. (b) The Blurred Piggyback approach applied to this code

`Stopping criterion`

This approach is a refinement of Piggyback. Therefore, we have the same stopping criterion as in the case of Piggyback adjoint, i.e. the stopping criterion checks at each iteration if $||\overline{F}(z, x, \overline{w}, \overline{y})^T||$ and $||F(z, x)||$ have converged to zero.

`Strengths and weaknesses:`

Blurred Piggyback has the same strengths and weaknesses as the Piggyback approach. This approach is more efficient in terms of time than Piggyback. Actually, the adjoint and original process are run in asynchronous way. Furthermore, the adjoint values are computed always by using the last computed original values which may accelerate the convergence of the adjoint loop.
However, similarly to Piggyback this approach is hard to implement inside an AD tool. In fact, this approach requires that the adjoint be implemented as a parallel program. This can be fine when the original loop is a parallel program itself. However, when the original loop is a sequential program, implementing the blurred piggyback requires in addition a parallelization of the original program.

3.2.5 Two-Phases approach

The "Two Phases" method is a special adjoint for the FP loops developed by B. Christianson in [10, 11]. Unlike the Black Box approach, the adjoint loop does not follow the same number of iterations as the original one. Actually, the adjoint is FP loop itself that has its its own initial guess as well as its own stopping criterion. This method is implemented for instance in the AD tool ADOL-C [50].

Let us consider the FP system

$$z_*(x) = \phi(z_*(x), x) \tag{3.11}$$

$$y = f(z_*(x), x) \tag{3.12}$$

where x represents the parameters and y is the desired solution. Applying the chain rule of differentiation to the total derivative of the objective y with respect to parameters x gives:

$$\frac{dy}{dx} = \frac{\partial}{\partial z} f(z_*, x).\frac{dz_*}{dx} + \frac{\partial}{\partial x} f(z_*, x) \tag{3.13}$$

The objective of the adjoint is to compute the projection of the Jacobian $\frac{dy}{dx}$ through the adjoint vector $\overline{y}$, i.e. we want to compute the following $\overline{x}$:

$$\overline{x} = \overline{y}.\frac{dy}{dx} = \overline{y}.\frac{\partial}{\partial z}f(z_*, x).\frac{dz_*}{dx} + \overline{y}\frac{\partial}{\partial x}f(z_*, x) \tag{3.14}$$

Defining $\overline{z}_0 = \overline{y}.\frac{\partial}{\partial z}f(z_*, x)$ and $\overline{x}_0 = \overline{y}.\frac{\partial}{\partial x}f(z_*, x)$, equation 3.14 rewrites as:

$$\overline{x} = \overline{x}_0 + \overline{z}_0.\frac{dz_*}{dx} \tag{3.15}$$

Applying the chain rule of differentiation to the total derivative of z_* with respect to parameters x gives:

$$\frac{dz_*}{dx} = \frac{\partial}{\partial z}\phi(z_*, x).\frac{dz_*}{dx} + \frac{\partial}{\partial x}\phi(z_*, x).$$

This can be solved for $\frac{dz_*}{dx}$, giving:

$$\frac{dz_*}{dx} = (I - \frac{\partial}{\partial z}\phi(z_*, x))^{-1}.\frac{\partial}{\partial x}\phi(z_*, x)$$

The multiplication of the row vector $\overline{z}_0$ by the Jacobian $\frac{dz_*}{dx}$ gives :

$$\overline{z}_0.\frac{dz_*}{dx} = \overline{z}_0.(I - \frac{\partial}{\partial z}\phi(z_*, x))^{-1}.\frac{\partial}{\partial x}\phi(z_*, x)$$

As $||\frac{\partial}{\partial z}\phi(z_*, x)|| < 1$, we may apply Taylor series, leading to:

$$\overline{z}_0.\frac{dz_*}{dx} = \overline{z}_0.(I + \frac{\partial}{\partial z}\phi(z_*, x) + [\frac{\partial}{\partial z}\phi(z_*, x)]^2 + [\frac{\partial}{\partial z}\phi(z_*, x)]^3 + ...).\frac{\partial}{\partial x}\phi(z_*, x) \tag{3.16}$$

This equation rewrites as:

$$\overline{z}_0.\frac{dz_*}{dx} = \overline{w}_*.\frac{\partial}{\partial x}\phi(z_*, x), \tag{3.17}$$

where $\overline{w}_*$ is the fixed point of an iterative method that satisfies the FP equation:

$$\overline{w}_* = \overline{z}_0 + \overline{w}_*.\frac{\partial}{\partial z}\phi(z_*, x).$$

It was shown in [10], that the rate of convergence of $\overline{w}$ to the solution $\overline{w}_*$ is equal to the asymptotic rate of convergence of z to the solution z_* .

Recalling equation 3.17, $\overline{x}$ may be written as :

$$\begin{aligned}\overline{x} = \quad \overline{x}_0 + (\overline{z}_0 + \overline{z}_0.\frac{\partial}{\partial z}\phi(z_*, x) + \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_*, x)]^2 + ... \\ + \overline{z}_0[\frac{\partial}{\partial z}\phi(z_*, x)]^{nAdj}).\frac{\partial}{\partial x}\phi(z_*, x)\end{aligned} \tag{3.18}$$

where $nAdj$ is the number of iterations needed to converge $\overline{w}$, i.e. $\overline{w}$ reaches $\overline{w}_*$. We note here that the initial guess of the iterative method $\overline{w}_0$ is chosen so that it holds the value of $\overline{z}$ resulting from adjoining the function f, i.e. $\overline{w}_0 = \overline{z}_0$.

To compute $\overline{x}$, Christianson proposes modifications on the adjoint generated by the Black Box approach. We saw in subsection 3.2.1, that at the end of the adjoint FP loop resulting from the Black Box approach, $\overline{x}$ may be written as:

$$\begin{aligned}\overline{x} = \overline{x}_0 + \overline{z}_0.\frac{\partial}{\partial x}\phi(z_*, x) + \overline{z}_0.\frac{\partial}{\partial z}\phi(z_{n-1}, x).\frac{\partial}{\partial x}\phi(z_{n-1}, x) + \\ \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_{n-2}, x)]^2.\frac{\partial}{\partial x}\phi(z_{n-2}, x) + ... + \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_0, x)]^n.\frac{\partial}{\partial x}\phi(z_0, x)\end{aligned} \tag{3.19}$$

where n is the number of iterations needed to converge the original FP loop, i.e. the FP loop converges when z reaches z_*. The row-vectors $\overline{x}_0$ and $\overline{z}_0$ result from the adjoint of the downstream computation f.
If we replace z_k by z_* in equation 3.19, we obtain:

$$\begin{aligned}\overline{x} = \overline{x}_0 + \overline{z}_0.\frac{\partial}{\partial x}\phi(z_*, x) + \overline{z}_0.\frac{\partial}{\partial z}\phi(z_*, x).\frac{\partial}{\partial x}\phi(z_*, x) + \\ \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_*, x)]^2.\frac{\partial}{\partial x}\phi(z_*, x) + ... + \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_*, x)]^n.\frac{\partial}{\partial x}\phi(z_*, x)\end{aligned}$$

Rearranging the equation, we obtain:

$$\begin{aligned}\overline{x} = \overline{x}_0 + (\overline{z}_0 + \overline{z}_0.\frac{\partial}{\partial z}\phi(z_*, x) + \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_*, x)]^2 + ... \\ + \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_*, x)]^n).\frac{\partial}{\partial x}\phi(z_*, x)\end{aligned} \tag{3.20}$$

We see that if we change n by $nAdj$ in equation 3.20, this equation becomes equation 3.18. This means that if we change the adjoint FP loop of the Black Box approach, so that it computes the gradients using the converged values of z, i.e. z_* and also it iterates as many times as needed to converge $\overline{w}$, then the adjoint FP loop becomes a FP loop itself.

To this end, we compute first $\overline{w}$ inside the adjoint FP loop. Then, we change the stopping criterion of the adjoint, so that, instead of following the same number of iterations as the original FP loop, it expresses rather the convergence of $\overline{w}$ to the solution $\overline{w}_*$. To optimize the adjoint code, we omit the computation of $\overline{z}$ as its value is almost equal to zero at the final iterations of the adjoint. Consequently, $\overline{x}$ will be computed by using the values of $\overline{w}$ at each iteration of the adjoint loop. At the end, the adjoint FP loop becomes a loop that initiates $\overline{\mathtt{w}}$ with some initial guess, e.g. $\overline{\mathtt{z}}_0$ and then iteratively calls:

$$\begin{aligned}\overline{\mathtt{w}}_{\mathtt{k+1}} &= \overline{\mathtt{z}}_0 + \overline{\mathtt{w}}_{\mathtt{k}}.\frac{\partial}{\partial \mathtt{z}}\phi(\mathtt{z}_*, \mathtt{x}) \\ \overline{\mathtt{x}} &= \overline{\mathtt{x}}_0 + \overline{\mathtt{w}}_{\mathtt{k+1}}.\frac{\partial}{\partial \mathtt{x}}\phi(\mathtt{z}_*, \mathtt{x})\end{aligned}$$

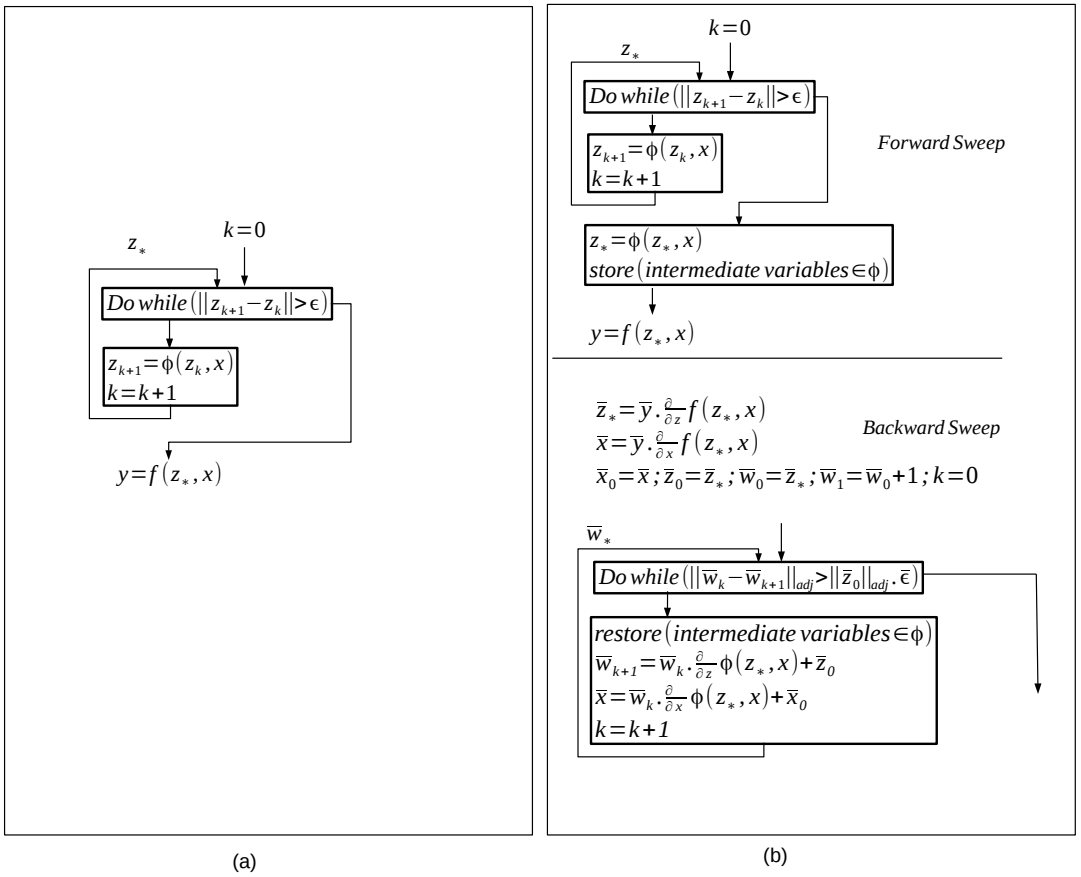

FIGURE 3.5: (a) An example of code containing a FP loop. (b) The Two-Phases approach applied to this code

until convergence of $\overline{w}$, i.e. $\overline{w}$ reaches $\overline{w}_*$.

Figure 3.5 shows an application of the Two-Phases approach to a FP loop. We see that the values of intermediate variables are saved only once during the last iteration of the original FP loop. Then, these values are restored many times during the iterations of the adjoint FP loop.

`Stopping criterion and initial guess`

We note by $||..||_{adj}$ the norm of the adjoint vectors. As the adjoint vectors are essentially row vectors, we may write : $||A||_{adj} = ||A^T||$, where A is a row vector and A^T is the transpose of A. Consequently, when we use the $1-norm$ in the stopping criterion of the original loop, we have to use the $\infty-norm$ in the stopping criterion of the adjoint loop. Symmetrically, when we use the $\infty-norm$ in in the stopping criterion of the original loop, we have to use the $1-norm$ in the stopping criterion of the adjoint loop. In the case of Euclidean norm, the norm used in the stopping criterion of the adjoint is the same as the one used in the stopping criterion of the original loop, i.e. $||A||_2 = ||A^T||_2$. In the light of error analysis, B. Christianson observes that if the desired accuracy is that $||\overline{x}_* - \overline{x}|| < \xi.||\overline{z}_0||$ with $\xi < 1$, then, the stopping criterion of the FP loop has to

check at each iteration if $||z_{k+1} - z_k|| \leq \epsilon$ and the stopping criterion of the adjoint FP loop has to check at each iteration if $||\overline{w}_k - \overline{w}_{k+1}|| \leq ||\overline{z}_0||.\overline{\epsilon}$ with ϵ and $\overline{\epsilon}$ are computed by using the value of ξ and estimations of the value of ρ and few other constants.
B.Christianson observes that $\overline{z}_0$ is a good initial guess for the adjoint FP loop, i.e. setting $\overline{w}_0 = \overline{z}_0$ may reduce the number of iterations needed to converge the adjoint values.

`Strengths and weaknesses:`

The main advantage of Two-Phases is its efficiency in terms of memory, i.e. it saves intermediate values of z of the last iteration only. Also, this method is general, i.e. it does not make assumptions on the structure of the FP loop. Furthermore, the adjoint of the FP loop is a FP loop that takes into account the convergence of the adjoint values. This method guarantees that whenever the original FP loop converges to the correct value, the adjoint FP loop will converge to the correct value too. In the case where the original values need only a few iterations to converge, e.g. when they start from a good initial guess, the adjoint loop will still perform enough iterations to converge the adjoint values. Symmetrically, when the adjoint needs only few iterations to converge e.g. in the case of Newton method the adjoint needs only one iteration, the adjoint loop will perform only the needed iterations.
From a practical point of view, this method is relatively easy to implement, i.e. it requires only a few modifications on the adjoint generated by the Black Box approach. As weaknesses, this method computes the value of $\overline{x}$ inside the adjoint loop which may slow down the computation time of derivatives. Also, this method can not be implemented as a parallel program.

3.2.6 Refined Two-Phases approach

The refined Two-Phases approach [10], as the name says, is a refinement of the Two-Phases approach. This method is implemented for instance in the AD tools TAF [17, 19] and OpenAD [21]. We saw in subsection 3.2.5 that the adjoint resulting from the Two-Phases approach initiates $\overline{w}$ with some initial guess and then iteratively calls:

$$\overline{\mathtt{w}}_{\mathtt{k+1}} = \overline{\mathtt{z}}_0 + \overline{\mathtt{w}}_{\mathtt{k}}.\frac{\partial}{\partial \mathtt{z}}\phi(\mathtt{z}_*, \mathtt{x})$$
$$\overline{\mathtt{x}} = \overline{\mathtt{x}}_0 + \overline{\mathtt{w}}_{\mathtt{k+1}}.\frac{\partial}{\partial \mathtt{x}}\phi(\mathtt{z}_*, \mathtt{x})$$

until convergence of $\overline{w}$, i.e. $\overline{w}$ reaches $\overline{w}_*$.
One can observe that at each iteration of the adjoint FP loop, $\overline{x}$ does not use the values of $\overline{x}$ computed at previous iterations. Therefore, it is common wisdom to place the computation of $\overline{x}$ outside the adjoint loop so that $\overline{x}$ uses only the converged value of $\overline{w}$,

$\overline{w}_*$. Figure 3.6 shows an application of the Refined Two-Phases approach to a FP loop. We notice here that the function $\phi(z, x)$ is differentiated twice: once with respect to the state z inside the adjoint FP loop and once with respect to parameters x outside the adjoint FP loop.

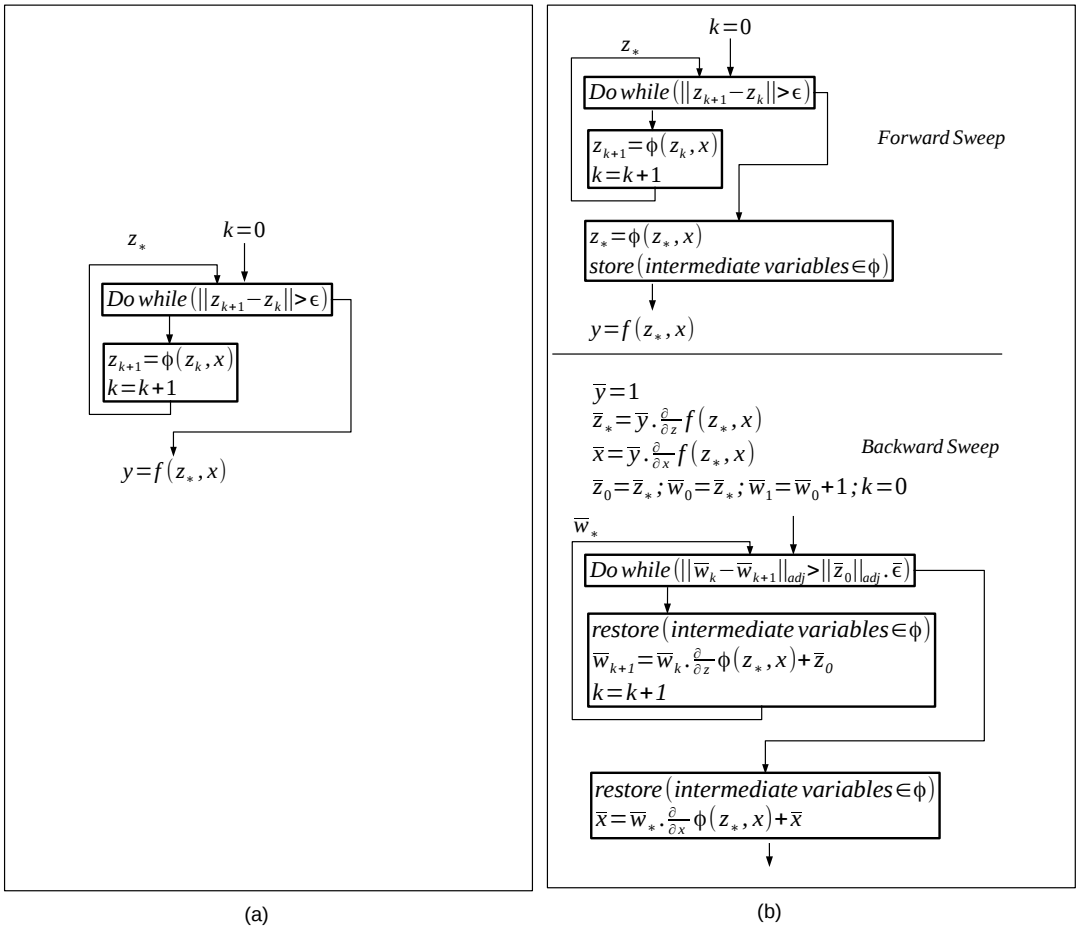

FIGURE 3.6: (a) An example of code containing a FP loop. (b) The Refined Two-Phases approach applied to this code

`Stopping criterion:`

We use here the same stopping criterion as in the case of non-refined Two-Phases approach, i.e. the stopping criterion checks at each iteration if $||\overline{w}_k - \overline{w}_{k+1}|| \leq ||\overline{z}_0||.\bar{\epsilon}$ with ϵ and $\bar{\epsilon}$ are computed by using the value of ξ and estimations of the value of ρ and few other constants.

`Strengths and weaknesses:`

The refined Two-Phases has not only the strengths of the Two-Phases approach but also the fact that the derivative of ϕ with respect to parameters x is computed only once outside the adjoint FP loop. This may reduce, consequently, the execution time of the adjoint.

However, to apply the refined Two-Phases method, one has to differentiate ϕ twice, once with respect to the state z inside the adjoint FP loop and once with respect to parameters x outside the adjoint loop. As the majority of AD tools can only perform the differentiation of ϕ with respect to all its independent variables at the same time, i.e. they can only generate a code that computes the derivative of ϕ with respect to z and x, one can either:

- use an AD tool to differentiate ϕ with respect to z and x and then split by hand the derivative of ϕ with respect to z from the derivative of ϕ with respect to x.
- use an AD tool to differentiate ϕ with respect to z and x and then set $\overline{x}$ to zero before $\frac{\partial}{\partial z}\phi(z_*, x)$; $\frac{\partial}{\partial x}\phi(z_*, x)$ during all the adjoint FP loop iterations except the last iteration. This guarantees that $\overline{x}$ will be computed only during the last iteration of the adjoint. This method is used for instance by the AD tool TAF [17].
- call the AD tool twice: once by specifying that the independents are z and the dependents are z, for $\frac{\partial}{\partial z}\phi(z_*, x)$, and another time by specifying that the independents are x and the dependents are z, for $\frac{\partial}{\partial x}\phi(z_*, x)$.
- improve the AD tool so that it differentiates ϕ in two different contexts on the same adjoint code.

3.2.7 Refined Black Box approach

We may imagine a method that combines the simplicity of the Black Box approach, seen in subsection 3.2.1 and the memory efficiency of the Two-Phases approach, seen in subsection 3.2.5. We call this method Refined Black Box. Although we didn't find references that define such a method in literature, we think that it might be interesting to look closely at this method and discover its advantages as well its weaknesses.
We saw in subsection 3.2.1 that the Black Box approach consists of two sweeps. The first one copies the original FP loop with saving the values of the state z (and the values of intermediate variables used to compute z) at each iteration. The second sweep contains the adjoint loop that computes the adjoint values by using the values of z (and the values of intermediate variables used to compute z) already stored. As the first iterations of the FP loop are meaningless for the adjoint, saving their intermediate values can be considered as a waste of memory. Along the lines of the Two-Phases approach, we may refine the Black Box approach by saving intermediate values of the last iteration only. Then, these values are read repeatedly by the adjoint loop.
The adjoint loop is thus a loop that follows exactly the same number of iterations as

the original FP loop and that iteratively calls:

$$\overline{\mathtt{z}}_{\mathtt{k}} = \overline{\mathtt{z}}_{\mathtt{k+1}}.\frac{\partial}{\partial \mathtt{z}}\phi(\mathtt{z}_{\mathtt{k}}, \mathtt{x})$$

$$\overline{\mathtt{x}} = \overline{\mathtt{z}}_{\mathtt{k+1}}.\frac{\partial}{\partial \mathtt{x}}\phi(\mathtt{z}_{\mathtt{k}}, \mathtt{x}) + \overline{\mathtt{x}}$$

Figure 3.7 shows the application of the Refined Black Box approach to a FP loop. The forward sweep consists of a copy of the original loop followed by one extra iteration in which the intermediate values are stored. The backward sweep is a loop that follows the same number of iterations as the original one and that uses the stored intermediate values to compute the derivatives.

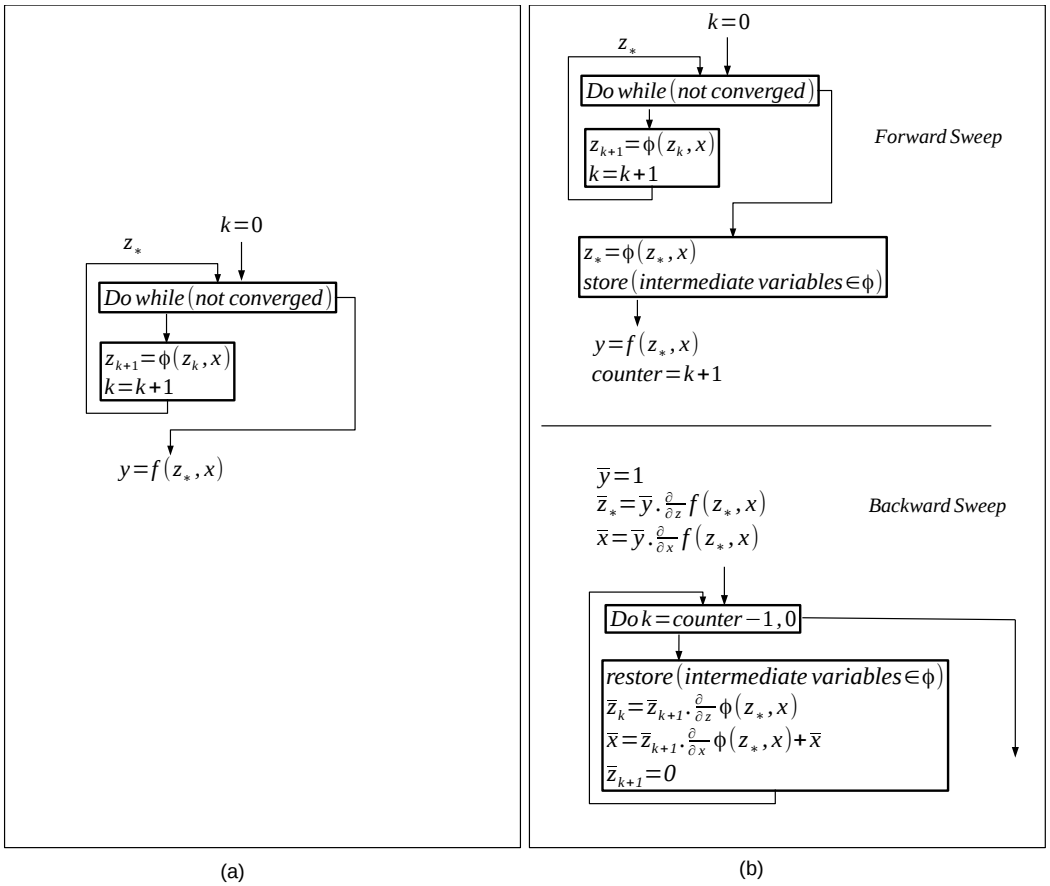

FIGURE 3.7: (a) An example of code containing a FP loop. (b) The Refined Black Box approach applied to this code

Mathematically, the resulting $\overline{x}$ may be written as :

$$\overline{x} = \overline{x}_0 + (\overline{z}_0 + \overline{z}_0.\frac{\partial}{\partial z}\phi(z_*, x) + \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_*, x)]^2 + ... + \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_*, x)]^n).\frac{\partial}{\partial x}\phi(z_*, x)$$

where n is the number of iterations of the original FP loop.

Assuming that the adjoint loop needs $nAdj$ iterations to converge to the same tolerance as the original loop. Refined Black Box approach returns a well converged $\overline{z}$ only when $nAdj \leq n$. In the opposite case, this method returns a non-converged $\overline{z}$. In any case, since the adjoint is based on the converged values of the state, we expect that the adjoint resulting from the refined Black Box needs fewer iterations to converge to the correct solution than in the case of the non-refined Black Box.

`Strengths and weaknesses:`

The main advantage of this approach is its efficiency in terms of memory, i.e. it saves the intermediate values of z only once. This approach has also the advantages of the Black Box approach such as the generality, i.e. it can be applied on any structure of FP loops, and the simplicity, i.e. it requires a minimal effort from the user. It is also relatively easy to implement inside an AD tool, i.e. it requires only a small modification in the stack mechanism.
However, similarly to the Black Box approach, this method does not take into account the convergence of the adjoint values which is dangerous in the cases where $nAdj > n$. Also, this method computes $\overline{x}$ inside the adjoint loop which may slow down the computation time of the derivatives.

3.3 Selecting the method to implement

In this subsection, we focus on the refined versions of the Black Box, Piggyback and Two-Phases approaches. We compare between some of these refined approaches and we select the one we find the best suited to be implemented in our AD tool.

3.3.1 Comparison between the Refined Black Box and Refined Two-Phases approaches

We saw in subsection 3.2.7, that the adjoint of the Refined Black box approach implements the equation:

$$\overline{x} = \overline{x}_0 + (\overline{z}_0 + \overline{z}_0.\frac{\partial}{\partial z}\phi(z_*, x) + \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_*, x)]^2 + ... + \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_*, x))]^n.\frac{\partial}{\partial x}\phi(z_*, x)$$

where n is the number of iterations of the original FP loop. From the other side, we saw in subsection 3.2.6 that when the initial guess is $\overline{z}_0$, the adjoint of the Refined Two Phases approach implements this equation:

$$\overline{x} = \overline{x}_0 + (\overline{z}_0 + \overline{z}_0.\frac{\partial}{\partial z}\phi(z_*, x) + \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_*, x)]^2 + ... + \overline{z}_0.[\frac{\partial}{\partial z}\phi(z_*, x)]^{nAdj}).\frac{\partial}{\partial x}\phi(z_*, x)$$

where $nAdj$ is the number of iterations needed to converge the adjoint values. One may observe that the two equations are similar and that the only difference is the number of iterations of the adjoint loop. This means that along the iterations, the adjoint resulting from the Refined Black Box approach gives exactly the same value of $\overline{x}$ as the adjoint resulting from the Refined Two Phases approach. Furthermore, the adjoint loop of the Refined Two Phases approach has the same convergence rate as the original FP loop. This means that in the majority of cases, the adjoint loop of the Refined Two Phases approach has the same number of iterations as the original loop and therefore the same number of iterations as the adjoint of the Refined Black Box approach. Consequently, in the majority of cases we have $n = nAdj$.
Thus, one may wonder: if the two approaches give exactly the same value of $\overline{x}$ at each iteration and also there is a very high probability that the two approaches give also exactly the same final result $\overline{x}$, i.e. $n = nAdj$ in the majority of cases, why do we need to apply the Refined Two Phases method especially that this approach requires specific modifications on the adjoint of the non-refined Black Box approach ?
We think that the main advantage of the Refined Two-Phases approach is that it protects the adjoint from the cases where the adjoint loop needs more iterations than the original loop, i.e. when $nAdj > n$. Also in the Refined Two Phases approach, we are able to define a good initial guess for the adjoint loop which may reduce its number of iterations. Finally, in the Refined Two-Phases approach, the partial derivative of ϕ with respect to the parameters x is computed only once outside the adjoint loop which may accelerate the computation time of the adjoint.

3.3.2 General weaknesses of the Piggyback class of methods

Piggyback class of methods includes Piggyback, Delayed Piggyback and Blurred Piggyback. These methods have generally two main weaknesses: The first one is that in the case of iterative methods with a superlinear convergence rate, e.g. Newton, these methods return an adjoint which is not efficient in terms of time.

We take for instance an iterative method that satisfies a Newton equation of the form:

$$z_* = z_* - (\frac{\partial}{\partial z}F(z_*, x))^{-1}.F(z_*, x)$$

We compute the derivative of $F(z, x) = 0$ at the solution $z = z_*$ with respect to x :

$$\frac{d}{dx}F(z, x) = \frac{\partial}{\partial x}F(z, x) + \frac{\partial}{\partial z}F(z, x).\frac{dz}{dx} = 0$$

Simplifying further the equation we obtain:

$$\frac{dz}{dx} = -\frac{\partial}{\partial z}F(z,x)^{-1}.\frac{\partial}{\partial x}F(z,x)$$

Applying the transpose we get:

$$(\frac{dz}{dx})^T = -(\frac{\partial}{\partial x}F(z,x))^T.(\frac{\partial}{\partial z}F(z,x)^{-1})^T.$$

Therefore, the adjoint of parameters $\overline{x}$ may be expressed as:

$$\begin{aligned}\overline{x} &= \overline{x}_0 + (\frac{dz}{dx})^T.\overline{z} \\ &= \overline{x}_0 - (\frac{\partial}{\partial x}F(z,x))^T.(\frac{\partial}{\partial z}F(z,x)^{-1})^T.\overline{z}\end{aligned}$$

where $\overline{x}_0$ is $\overline{x}$ computed before the the FP loop adjoint.

Obviously, an efficient adjoint of Newton method [43] needs to wait until that the state converges to the solution z_*. Then, it solves the linear system:

$$(\frac{\partial}{\partial z}F(z,x)^{-1})^T.g = -\overline{z}$$

followed by a call to the adjoint of F with respect to x:

$$\overline{x} = \overline{x} + (\frac{\partial}{\partial x}F(z,x))^T.g$$

Unfortunately Piggyback methods do not behave this way, i.e. these methods compute the adjoint derivatives together with the original values inside a loop until the convergence of both of them. Therefore, generally Piggyback methods are not recommended in this case of iterative methods.

The second weakness is that Piggyback methods compute the adjoint by using values of z that are very far from the solution z_*. These values of z may sometimes not respect the inequality:

$$||\frac{\partial}{\partial z}\phi(z,x))|| < 1$$

and therefore may cause the divergence of the adjoint. We take for instance the equation:

$$F(z_*,x) = z_*^2 - x = 0.$$

Given a parameter x, we try to find its square root z_* by using an iterative method that satisfies the Newton equation:

$$z_* = \phi(z_*,x) = \frac{1}{2}.(z_* + \frac{x}{z}).$$

Consider the case where $x = 4$ and therefore $z_* = 2$. The function $\phi(z, 4)$ is contractive for all z that satisfy the inequality $|2 - \phi(z, 4)| < |2 - z|$. Thus, for all $z > 0.67$.
However, $|\frac{\partial}{\partial z}\phi(z, 4)| < 1$ only for $z > 1.15$. Consequently, for all z such that $0.67 < z < 1.15$, the function $\phi(z, x)$ is contractive but its corresponding adjoint $\frac{\partial}{\partial z}\phi(z, x)$ is diverging.
This issue, known as the "lag effect" of the adjoint, is significant in the case of the non-refined Piggyback approach and less significant in the case of Delayed and Blurred Piggyback approaches. This is because these latter wait that the value of z become sufficiently close to z_* before starting the adjoint computations.

3.3.3 Comparison between the Delayed Piggyback and Refined Two-Phases approaches

Both Refined Two-Phases and Delayed Piggyback methods yield an adjoint convergence rate similar to original Fixed-Point loop. Derivatives convergence may lag behind by a few iterations, but will eventually converge at the same rate. Both methods achieve to differentiate only the last or the few last iterations i.e. those who operate on physically meaningful values. Both manage also to avoid naïve inversion of the original sequence of iterations, therefore saving the cost of data-flow reversal. Consequently the adjoint, which is itself a fixed point, must have a distinct, specific stopping criterion.

Because of its setting, Delayed Piggyback method makes some additional assumptions on the shape of the iteration step and on the structure of the surrounding program whereas Refined Two-Phases remains general. Another difference is that Refined Two-Phases starts adjoining the iteration step, actually the last one, only when the original iteration has converged "fully", whereas Delayed Piggyback triggers the adjoint iterations earlier, together with the remaining original ones, when those are converged only "sufficiently". This may be hard to determine automatically. Since Delayed Piggyback adjoint computation starts with slightly approximate values, it may require a few more iterations than Refined Two-Phases. A last difference is that Delayed Piggyback requires adjoining the sequel of the program i.e. the part f after the Fixed-Point iteration, repeatedly inside the adjoint iteration step. This is fine in the chosen setting where the sequel is assumed short, but it has a significant cost in general when the sequel is complex or when Fixed-Point loops are nested.

3.3.4 Comparison between the Blurred Piggyback and Refined Two-Phases approaches

Both Blurred Piggyback and Refined Two-Phases approaches are efficient in terms of memory, i.e. they require the storage of z and the intermediate variables used to compute z during only one iteration. Both methods consider that the adjoint of a FP loop is a FP loop itself, i.e. it has own initial guess as well as its own stopping criterion. T.Bosse claims that his method Blurred Piggyback requires less computation time than the Two-Phases approach. However, Blurred Piggyback is hard to be implemented inside an AD tool. It requires that the adjoint be implemented as a parallel program. This is can be fine when the original loop is a parallel program itself. However, when the original loop is a sequential program, implementing the blurred piggyback requires in addition a parallelization of the original program.

3.3.5 Our choices

We detailed above various methods that propose efficient adjoint for FP loops. These methods manage to avoid naive inversion of the original sequence of iterations, therefore saving the cost of data-flow reversal. The main difference between these approaches is mainly when starting the adjoint computations. Some of them start adjoining since the first iterations of the original loop, some others wait until that the original values become sufficiently converged and some others compute the adjoint only when the original values have fully converged. Among these adjoints, we select the one we find the best suited to be implemented in our AD tool. We choose the approach that:

- covers more cases, i.e. we prefer not have assumptions in the iteration shape, and that preserves the general structure of adjoint codes. This is unfortunately not the case of Piggyback class of methods.
- considers that the adjoint of FP loop is a FP loop itself. This guarantees the convergence of the derivatives and gives the user the opportunity to set a good initial guess for the adjoint. Unfortunately, this is not the case of Black Box class of methods.
- computes the value of $\overline{x}$ outside the adjoint loop which may reduce, consequently, the computation time of the adjoint. This is unfortunately not the case of the Two-Phases approach.

For these reasons, we currently select the Refined Two-Phases approach to be implemented in our AD tool Tapenade.

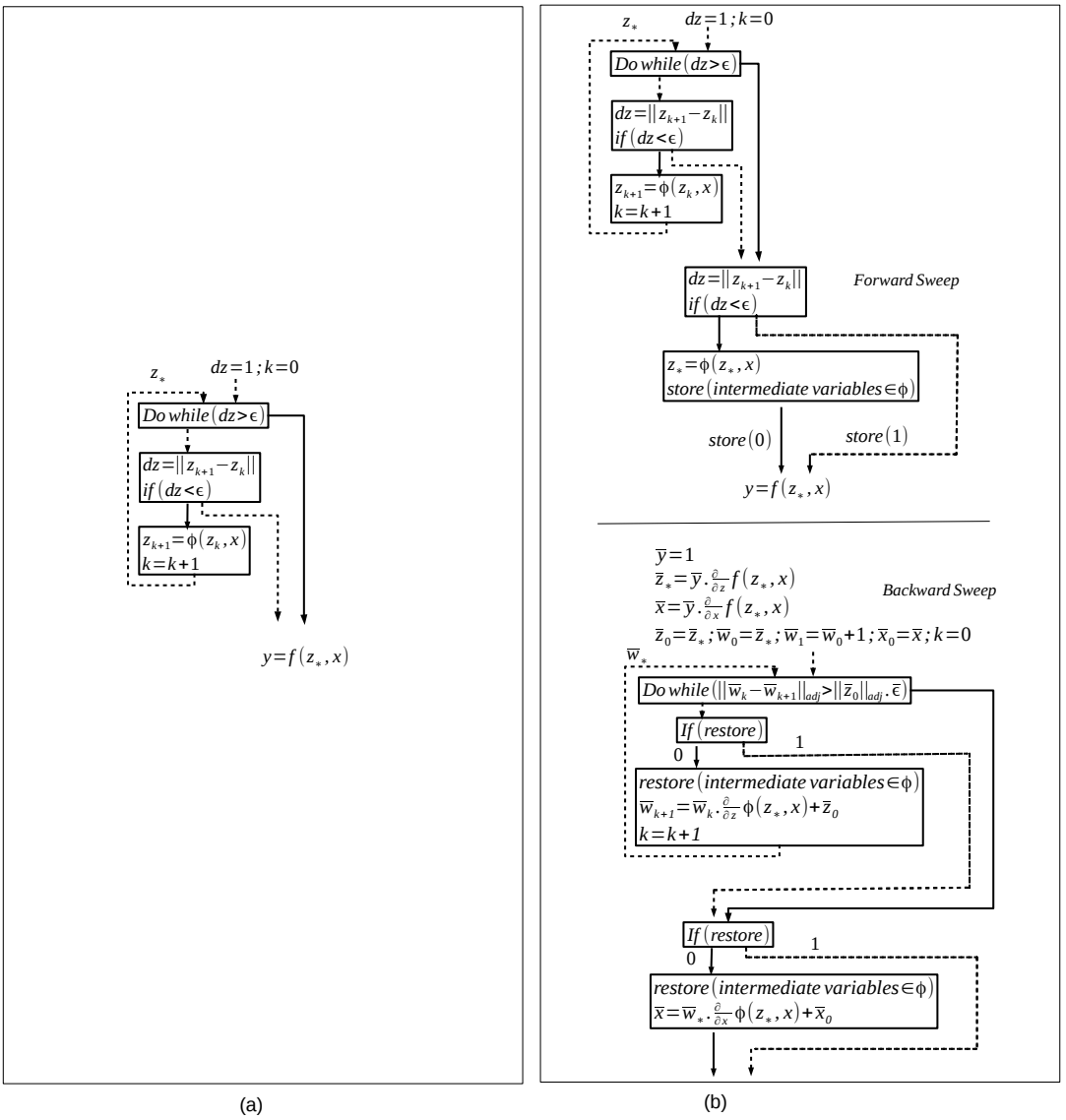

FIGURE 3.8: (a) FP Loop with two exits. (b) Applying the Two-phases method to the loop. Dashed lines show the trajectory followed during run-time

For simplicity, in the sequel we will call this method "Two-Phases" rather than refined Two-Phases.

3.4 Questions related to the code structure

Theoretical works about the FP loops often present these loops schematically as a while loop around a single call to a function ϕ that implements the FP iteration (see figure 3.6 (a)). FP loops in real codes almost never follow this structure. Even when obeying a classical while loop structure, the candidate FP loop may exhibit multiple loop exits and its body may contain more than only ϕ e.g. I/O. In many cases, these structures prevent application of the theoretical adjoint FP method.

Consider a first example (see figure 3.8 (a)), where the original loop is a while loop that contains an alternate exit at the middle of the loop body. More precisely, the alternate exit is located just before the computation of ϕ. Figure 3.8 (b) shows the application of the refined two-phases to this example. The Forward sweep of the adjoint contains a copy of the original loop followed by an extra iteration that saves the values of intermediate variables used to compute the state. This extra iteration is basically the last iteration of the original loop. The backward sweep of the adjoint contains a new FP loop that computes the adjoint values by using the values already stored during the extra iteration of the Forward sweep. Because of the exit located in the middle of the body, the last iteration does not sweep through ϕ. Actually, this iteration contains only a test of the convergence of the state. As the adjoint loop adjoins repeatedly the last iteration and this latter contains only non-active variables, the adjoint computes nothing and it returns the value of $\overline{x}$ as it was before the adjoint loop, i.e. $\overline{x} = \overline{x}_0$.

Consider now a second example (see figure 3.9 (a)) in which ϕ is the composition of two functions ϕ_1 and ϕ_2, so that $\phi(z, x) = \phi_2(\phi_1(z, x), x)$. The FP loop contains two exits: one exit at the top of the loop and a second exit at the middle of the loop body so that it splits ϕ into two parts. The first part contains the computation of $\phi_1(z, x)$ and the second part contains the computation of $\phi_2(h, x)$, where h is an intermediate variable that holds the value of $\phi_1(z, x)$. One may observe, that the last iteration of the loop sweeps only through ϕ_1. Consequently, the adjoint loop computes only the derivative of ϕ_1 with respect to z. More precisely it computes $\overline{w}_{k+1} = \overline{h}.\frac{\partial}{\partial z}\phi_1(z_*, x) + \overline{z}_0$. As the variable h is not used at the sequel of the FP loop, its corresponding adjoint $\overline{h}$ is null at the entry of the adjoint loop and thus $\overline{w}_{k+1} = \overline{z}_0$. Similarly, $\overline{x}$ results from the differentiation of ϕ_1 with respect to x. As the value of $\overline{h}$ is null, the adjoint returns the value of $\overline{x}$ as it was before the adjoint loop, i.e. $\overline{x} = \overline{x}_0$.

One might remove the second exit of the example of figure 3.9 (a) by introducing Boolean variables and apply again the refined two-phases method. Unfortunately, the last iteration of the transformed loop (see figure 3.10) still sweeps only through ϕ_1. Therefore, the adjoint loop computes only the derivative of ϕ_1 with respect to z. Here also, the value of $\overline{h}$ is null at the entry of the adjoint loop. This makes the value of $\overline{x}$ equals to $\overline{x}_0$ at the exit of the adjoint which is clearly incorrect result. To enforce the last iteration to sweep through the whole ϕ, one may transform the loop of figure 3.9 (a) by the peeling method. Applying the refined two-phases approach to this new transformed loop, yields to an adjoint (see figure 3.11) that repeatedly calls:

$$\overline{w}_{k+1} = \overline{w}_k.\frac{\partial}{\partial z}\phi_1(z_*, x).\frac{\partial}{\partial h}\phi_2(h, x)$$

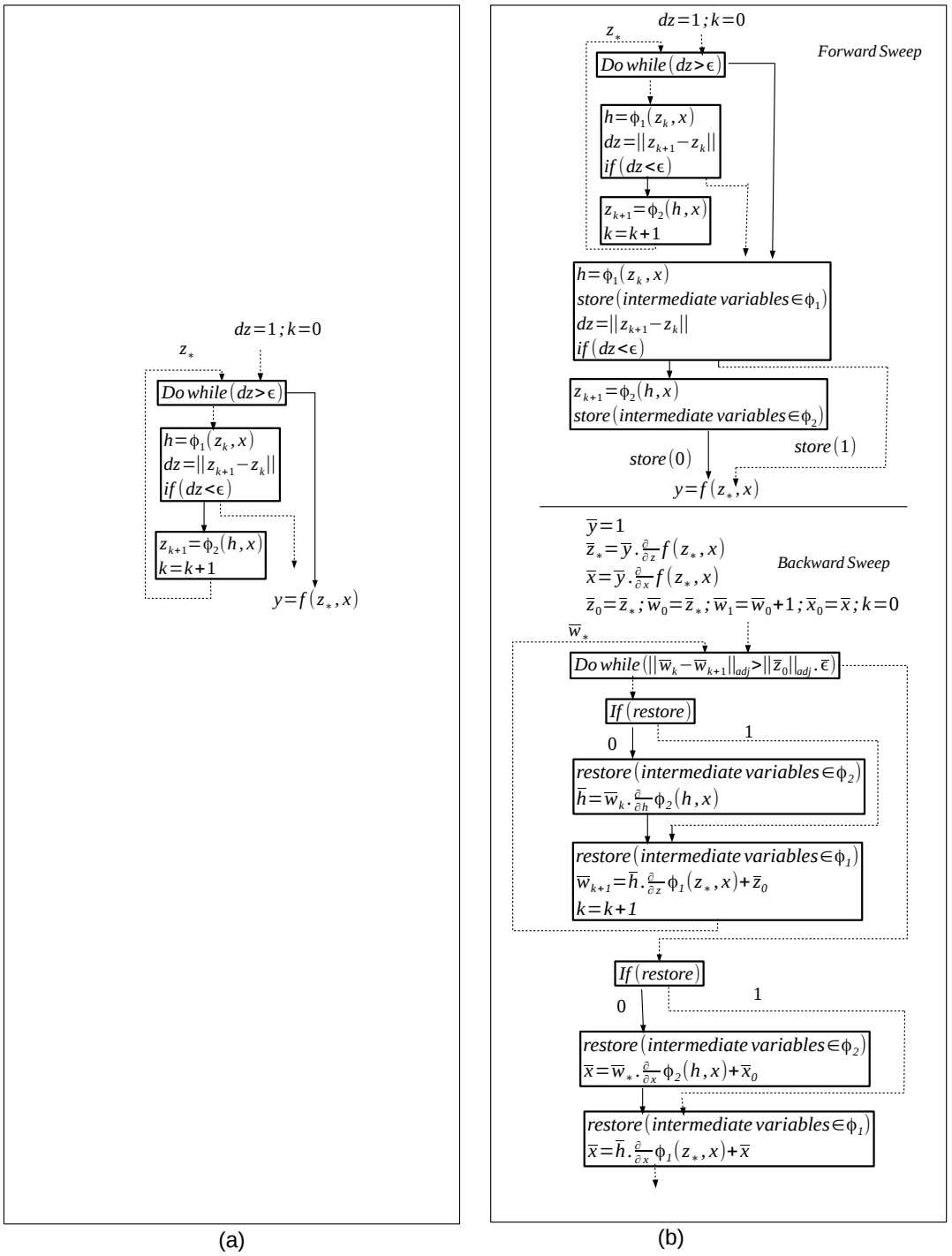

FIGURE 3.9: (a) FP Loop with two exits. (b) Applying the Two-phases method to the loop. Dashed lines show the trajectory followed during run-time

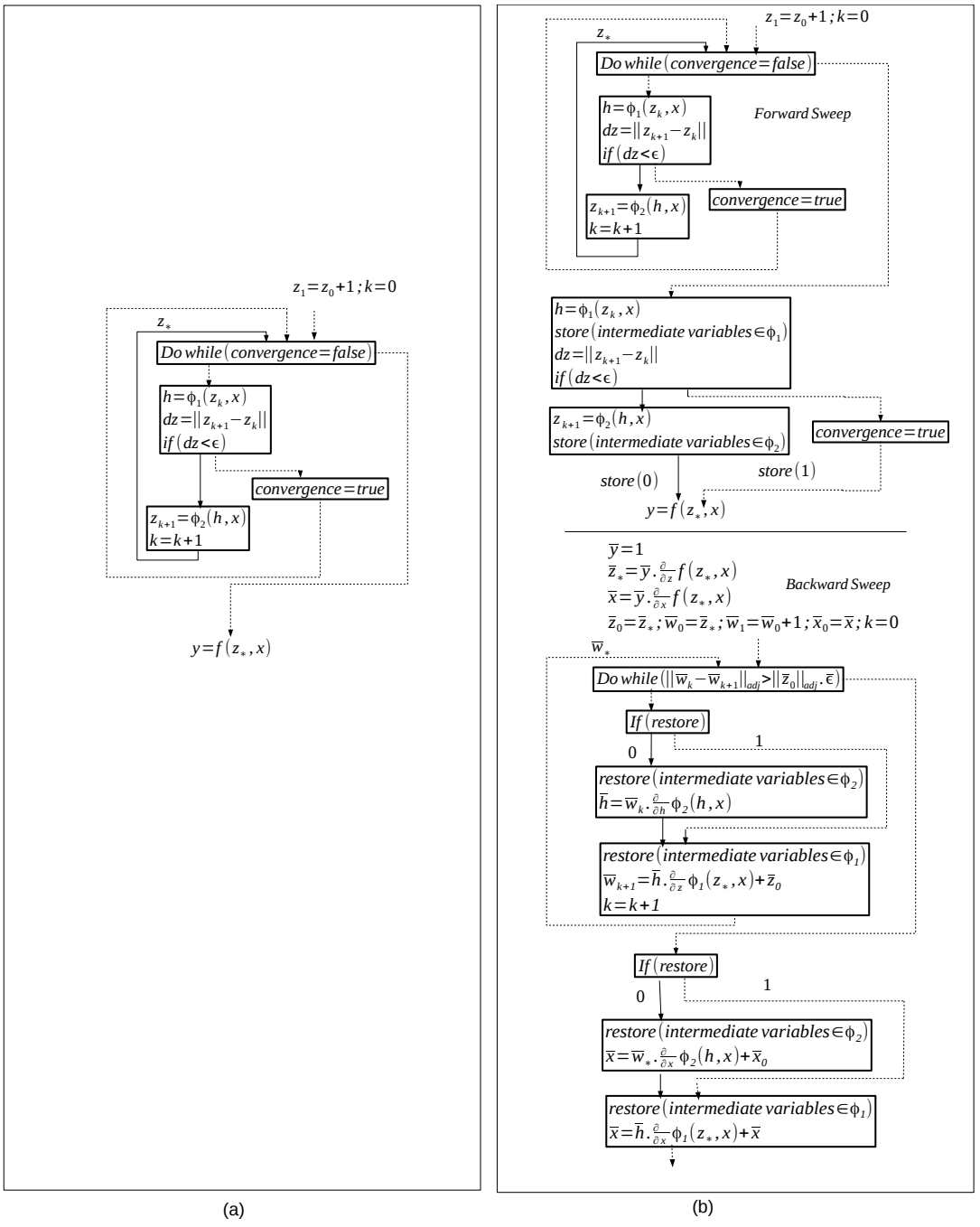

FIGURE 3.10: (a) FP Loop with one exit on the top. (b) Applying the Two-Phases method to the loop Dashed lines show the trajectory followed during run-time.

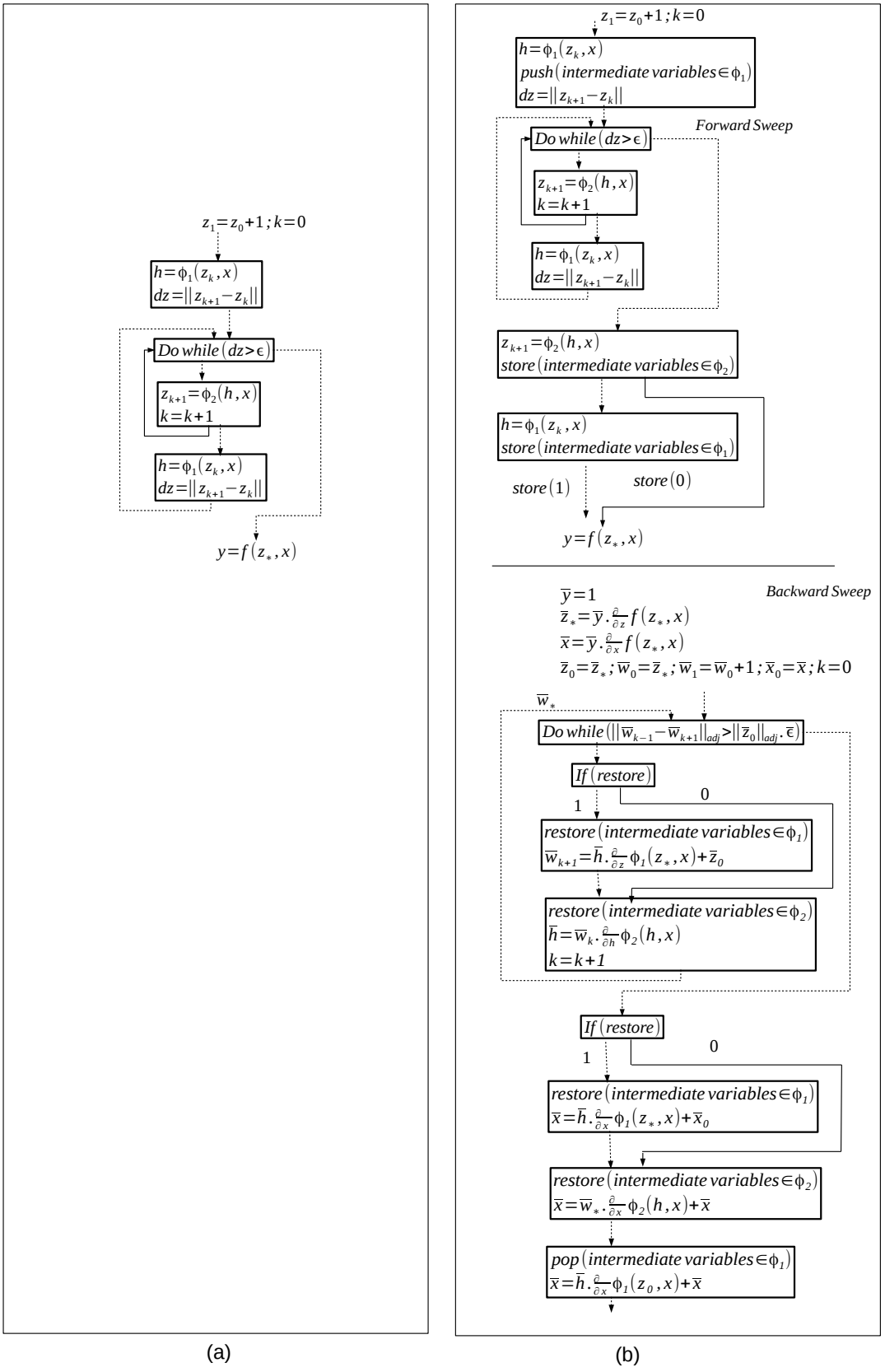

FIGURE 3.11: (a) FP Loop with one exit. (b) Applying the Two-phases method to the loop. Dashed lines show the trajectory followed during run-time.

until convergence of $\overline{w}$, i.e. $\overline{w}$ reaches $\overline{w}_*$. At the end of the adjoint, $\overline{x}$ may be written as :

$$\overline{x} = \overline{x}_0 + \overline{w}_*.\frac{\partial}{\partial x}\phi_2(h,x) + \overline{w}_*.\frac{\partial}{\partial h}\phi_2(h,x).\frac{\partial}{\partial x}\phi_1(z_*,x)$$

We saw in subsection 3.2.5, that mathematically $\overline{x}$ is written as :

$$\overline{x} = \overline{x}_0 + \overline{w}_*.\frac{\partial}{\partial x}\phi(z_*,x),$$

where $\overline{w}_*$ is the solution of the FP equation: $\overline{w}_* = \overline{z}_0 + \overline{w}_*.\frac{\partial}{\partial z}\phi(z_*,x)$.
As $\phi(z_*,x) = \phi_2(\phi_1(z_*,x),x)$, $\overline{x}$ may be written as :

$$\overline{x} = \overline{x}_0 + \overline{w}_*.\frac{\partial}{\partial x}\phi_2(h,x) + \overline{w}_*.\frac{\partial}{\partial h}\phi_2(h,x).\frac{\partial}{\partial x}\phi_1(z_*,x),$$

where $\overline{w}_*$ is the solution of the FP equation $\overline{w}_* = \overline{z}_0 + \overline{w}_*.\frac{\partial}{\partial h}\phi_2(h,x).\frac{\partial}{\partial z}\phi_1(z_*,x)$.
We may observe thus that the $\overline{x}$ obtained by applying the refined two-phases method to the transformed loop matches $\overline{x}$ obtained mathematically.
In order to apply the refined two-phases approach, we need thus to define a set of sufficient conditions on the candidate FP loop. Obviously, the first condition is that the state variables reach a fixed point i.e. their values are stationary during the last iteration, up to a certain tolerance ϵ.
Moreover, the last iteration must contain the complete computation of ϕ. This forbids loops with alternate exits, since the last iteration does not sweep through the complete body. Classically, one might transform the loop body to remove alternate exits, by introducing Boolean variables and tests that would affect only the last iteration. We must forbid these transformed loops as well. To this end, we add the condition that even the control flow of the loop body must become stationary at convergence of the FP loop. This is a strong assumption that cannot be checked statically, but could be checked dynamically.
Conversely, the candidate FP loop could contain more than just ϕ. We must forbid that it computes other differentiable variables that do not become stationary. To enforce this, we require that every variable overwritten by the FP loop body is stationary. One tolerable exception is about the computation of the FP residual, which is not strictly speaking a part of ϕ. Similarly, we may tolerate loop bodies that contain I/O or other non-differentiable operations.
It may happen that the (unique) loop exit is not located at the loop header itself but somewhere else in the body. These loops can be transformed by peeling, so that the exit is placed at the loop head, and the conditions above are satisfied. This peeling is outside the scope of this work and we will simply require that the loop exit is at loop header.
These are sufficient applicability conditions to apply not only the refined two-phases

approach, but also the two-phases approach, the refined black box approach and all the approaches based on adjoining only the last iteration.

3.5 Implementation

In this section, we describe the way we implemented the Two-Phases approach in our AD tool Tapenade. We believe that an efficient implementation of this special FP adjoint inside the tool has to:

- detect the maximum of features during the compilation phase of the AD tool. This means, for instance, taking advantage from the static analyses performed by the tool.
- use these features to generate an efficient adjoint that has to be as similar as possible to the theoretical one described in subsection 3.2.6.
- apply the special adjoint to nested structure of FP loops.

In subsection 3.5.1, we see how the stack mechanism has been extended in order to allow a repeated access to the last iteration of the FP loop. In subsection 3.5.2, we describe how static analyses are used to detect the various variables needed by the adjoint, i.e. the variables that form the state and those that form the parameters. In subsection 3.5.3, we detail some of the choices we made in order to implement the Two-Phases adjoint. In subsection 3.5.4, we see how we specified our transformation on the Control Flow Graphs. In subsection 3.5.5, we describe how we use the `Activity` analysis, seen in Chapter 2, to differentiate the body of the FP loop once with respect to the state and once with respect to the parameters.

3.5.1 Extension of the stack mechanism

We mentioned in section 3.2.6 that the intermediate values are stored only during the last forward iteration. Then they are repeatedly used in each of the backward iterations. Our standard stack mechanism does not support this behavior. We need to define an extension to specify that some zone in the stack (a "repeated access zone") will be read repeatedly. Our choice is to add three new primitives to our stack, supposed to be called at the middle of a sequence of stack `pop`'s (see figure 3.12).

- `start_repeat_stack()` states that the current stack position is the top of a repeated access zone.

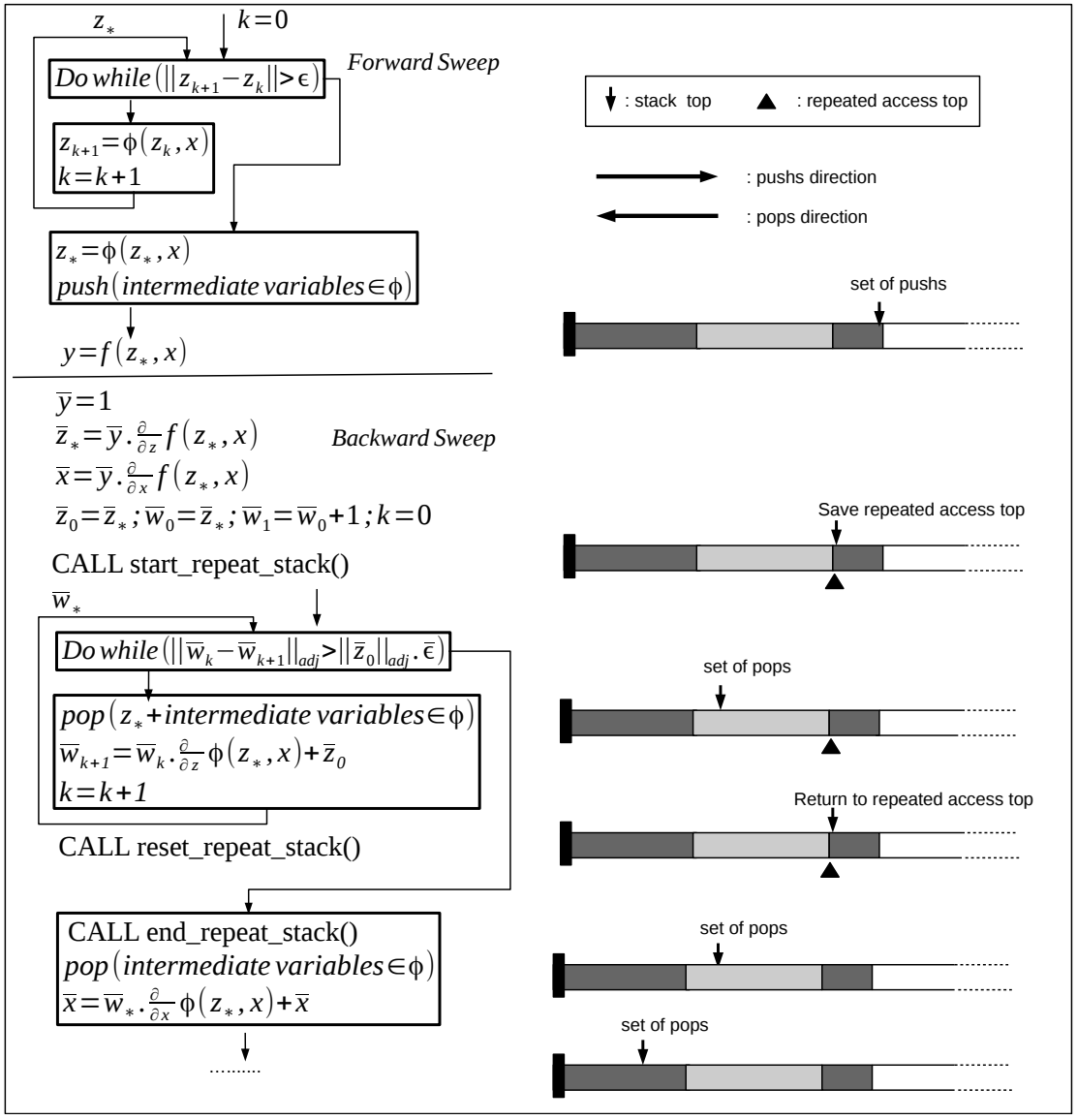

FIGURE 3.12: The new stack primitives allows a repeated access to the values stored during the last iteration of the FP loop

- `reset_repeat_stack()` states that the stack pointer must return to the top of the repeated access zone.
- `end_repeat_stack()` states that there will be no other read of the repeated access zone.

In the adjoint generated code, these procedures must be called :

- `start_repeat_stack()` at the start of the adjoint FP loop.
- `reset_repeat_stack()` at the end of the body of the adjoint FP.
- `end_repeat_stack()` at the end of the adjoint FP loop.

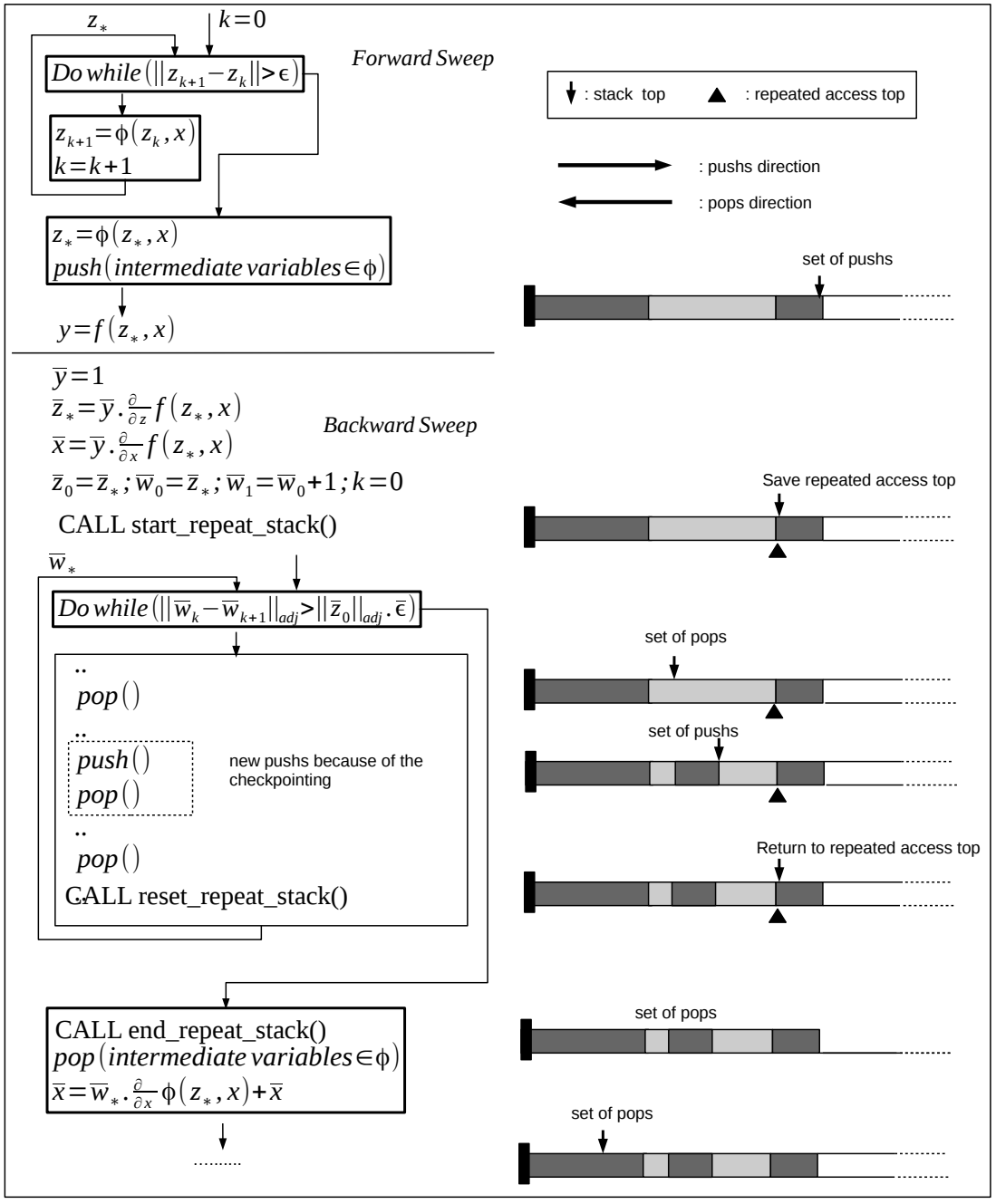

FIGURE 3.13: Checkpointing occurring inside the adjoint iterations overwrites the contents of the repeated access zone

However this set of primitives doesn't handle the case of checkpointing occurring inside the adjoint iterations (see figure 3.13). Checkpointing implies that the stack may grow again (with `push`'s) and the danger is to overwrite the contents of the repeated access zone. Our solution to keep this zone safe is to store the new values at the real top of the stack, i.e. above the repeated access zone. This requires two additional primitives.

- `freeze_repeat_stack()` saves the current stack pointer (we call it "the frozen top") and says that all coming `push`'s must go above the top of the current repeated access zone.
- `unfreeze_repeat_stack()` states that previous `pop`'s have returned the stack pointer to the top of the current repeated access zone, and therefore resets the stack pointer to its saved location so that next `pop`'s will read in the repeated access zone.

This is illustrated by figure 3.14. Notice that `unfreeze_repeat_stack()` is in principle unnecessary, since every `pop` could check if the stack pointer is at the top of a repeated access zone and react accordingly. However this would slow down each call to `pop`, which are frequent. On the other hand, `unfreeze_repeat_stack` may be called only once, at a location that can be statically determined by the AD tool. Therefore, in the adjoint generated code, we will call :

- `freeze_repeat_stack()` before each checkpointed adjoint subroutine call or code fragment during the adjoint backward iteration.
- `unfreeze_repeat_stack()` after the corresponding adjoint subroutine call or code fragment.

Once leaving the adjoint loop, these two primitives should not be called any more since there is no need to protect the repeated access zone.

Similarly to the Two-Phases approach, the two approaches Two-Phases (seen in subsection 3.2.5) and Refined Black Box (seen in subsection 3.2.7) require a repeated access to the stored intermediate values of the last iteration. To apply these two approaches, one may use the new primitives of the stack but in re-arranging them in different way. In both approaches, we must call:

- `start_repeat_stack()` at the start of the adjoint FP loop.
- `reset_repeat_stack()` at the start of the body of the adjoint FP. This call has not to be done during the first iteration, as at the beginning of this iteration we are already at the top of the repeated access zone.

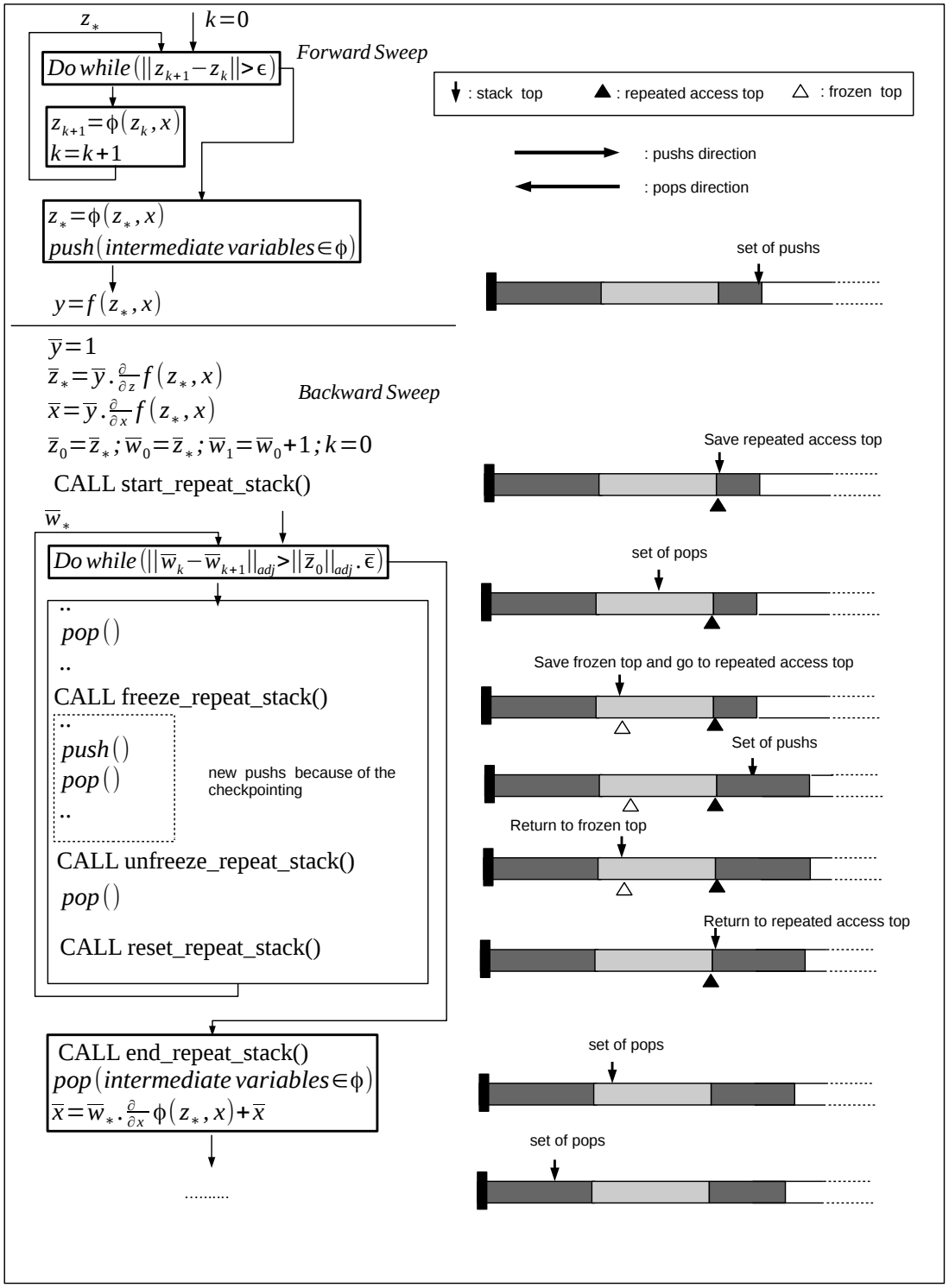

FIGURE 3.14: Because of the additional stack primitives, checkpointing occurring inside the adjoint iterations does not overwrite the contents of the repeated access zone

- `end_repeat_stack()` at the end of the adjoint FP loop.

The two primitives `freeze_repeat_stack()` and `unfreeze_repeat_stack()` must be called as usual, i.e. around each checkpointed adjoint subroutine call or checkpointed code fragment inside the adjoint loop.

One may wonder how these primitives are really implemented inside the stack. In reality, the `push` /`pop` primitives as they are implemented in our AD tool have a special mechanism. These primitives do not deal directly with the stack but have rather access to a set of buffers that in their side deal with the stack. More precisely, the `push`'s primitives save their values in the buffers and the `pop`'s primitives retrieve their values from these buffers. We have one buffer for each type, i.e. one buffer for the reals, one for the integers,..etc. These buffers are actually arrays with fixed length. When a buffer becomes full, we push all its value into the stack. Symmetrically, when a buffer becomes empty, we fulfill it from the stack.
Implementing the new primitives that handle the repeated access to the stack means taking care of this special mechanism. We implemented the primitives so that:

- `start_repeat_stack()`: pushes the values of the non empty buffers into the stack and saves the pointer at the top of the stack (repeated access top). It also saves the number of values pushed from each buffer. We need to make sure that all the values have been saved in the stack. Then we reset every thing as before the call to `start_repeat_stack()`, i.e. we fulfill the buffers as before and we reset the pointer to its old location.

- `reset_repeat_stack()`: flushes the buffers, states the pointer at the saved location, i.e. at the repeated access top and then fulfill the buffers with the same number of values as it was done in the `start_repeat_stack()`, i.e. we already know the number of values pushed in the `start_repeat_stack()`.

- `end_repeat_stack()`: states that we are no more dealing with a repeated access zone.

To implement the two primitives `freeze_repeat_stack()` and `unfreeze_repeat_stack()`, we adopted different strategy from the one used to implement the two primitives
`start_repeat_stack()` and `reset_repeat_stack()`. In fact, pushing the values of non empty buffers into the stack may overwrite the values of the repeated access zone. From the other side, we observed that, if we have a non empty buffer at the moment when the `freeze_repeat_stack()` is executed, this is either because this buffer was

fulfilled by the `start_repeat_stack()` called at the entry of the repeated access zone or by one of the `pop`'s primitives situated between the `start_repeat_stack()` and the `freeze_repeat_stack()`. Based on this observation, our idea is, thus:

- Save the position of the pointer at the top of the stack before each call to a pop primitive situated between the `start_repeat_stack()` and the `freeze_repeat_stack()`. We call this "frozen top". We save one frozen top by type, i.e. one frozen top for the reals, one for integers,..etc.
- At the moment of `freeze_repeated_access()`, we save the number of values inside each non empty buffer.
- At the moment of `unfreeze_repeated_access()`: for each buffer, we set the pointer to the frozen top of the same type and then we fulfill this buffer with the saved number of values.

3.5.2 Fixed-Point directive and automatic detection of Fixed-Point elements

It is very hard or even impossible to detect every instance of FP loop inside a given code. Even when the original loop is a simple loop with one exit at the top, an AD tool cannot determine statically if the control flow of this loop will converge or if every overwritten variable inside this loop will reach a fixed point. Therefore, we rely on the end-user to provide this information, for instance through a directive. As we required in subsection 3.4 that the candidate FP loop has the syntactic structure of a loop, one directive, placed on the loop header, is enough to designate it. Thanks to AD-specific data-flow analyses, described further insubsection 3.5.5, the AD tool can distinguish between the code that contains the computation of the state and the code that contains other non-differentiable operations. An example of non-differentiable operations is the residual computation since it computes the number of iterations of the loop which is essentially discrete and therefore non-differentiable.

To apply Two-Phases method, we need to distinguish between the state z and parameters x for three main reasons:

- We need to differentiate $\phi(z_*, x)$ with respect to z inside the adjoint loop and with respect to x outside the adjoint loop.
- At the end of each adjoint iteration, we add to $\overline{w}$ the value of $\overline{z}$ computed before the adjoint FP loop.

- The stopping criterion checks at each iteration the convergence of the adjoint of the state only.

To detect the state and parameters we rely on the results of IN-OUT analysis run by the AD tool on the original program. The state is the set of variables that are modified inside the FP loop and the parameters are the variables that are only read inside the loop. Given the **use** set of the variables read by the FP loop and the **out** set of the variables written by the FP loop, we can define:

$$\begin{aligned} \text{state} &= \mathbf{out}(\text{FP loop}) \\ \text{parameters} &= \mathbf{use}(\text{FP loop}) \backslash \mathbf{out}(\text{FP loop}) \end{aligned}$$

One may observe that the variables that are modified inside the FP loop and not used by the sequel f, we call them z_{nu}, have an adjoint null at the entry of the adjoint FP loop. Therefore, each computation that adds to $\overline{w}$ the value of $\overline{z}_{nu}$ may be eliminated from the adjoint FP loop. To do so, one may refine the set of state variables by specifying that the state variables are only the variables that are modified inside the FP loop and used at the sequel f. Formally we write:

$$\text{state} = \mathbf{out}(\text{FP loop}) \cap \mathbf{live}$$

where **live** is the set of the variables that are used in the sequel of the FP loop. As we are only looking for differentiable influences of the parameters on the state, we may further restrict the above sets to the variables of differentiable type i.e. REAL or COMPLEX.

3.5.3 Specification of the Implementation

In this subsection, we detail some of the choices we made in order to implement the Two-Phases adjoint in our AD tool.

3.5.3.1 The stopping criterion of the adjoint loop

In Two-Phases method, see subsection 3.2.6, the stopping criterion of the original FP loop checks at each iteration if $||z_{k+1} - z_k|| \leq \epsilon$ with z_{k+1} is the value of the state at the current iteration and z_k is the value of the state at the previous iteration. On the other hand, the stopping criterion of the adjoint loop checks at each iteration if $||\overline{w}_k - \overline{w}_{k+1}||_{adj} \leq ||\overline{z}_0||_{adj}.\overline{\epsilon}$ with $\overline{w}_{k+1}$ is the value of $\overline{w}$ at the current iteration, $\overline{w}_k$ is the value of $\overline{w}$ at the previous iteration, $||..||_{adj}$ is the norm of the adjoint vectors and $\overline{\epsilon}$ is computed by using estimations of some constants.

In practice, even if the AD tool manages to detect the location of the residual computation, it is quite impossible that it understands the mathematical equation behind it and then generates the appropriate $||..||_{adj}$ and $\bar{\epsilon}$. It is also quit difficult to detect the variable that holds the value of the state at the previous iteration, i.e. detect the intermediate variable that represents z_k. Therefore, for every FP loop we will:

- Create an intermediate variable that holds the value of $\overline{w}$ at the previous iteration. We will call it for instance $\overline{w}_{inter}$.
- Set the stopping criterion of the adjoint, so that it tests at each iteration if $||\overline{w} - \overline{w}_{inter}|| < \bar{\epsilon}$ with $||..||$ is the euclidean norm and $\bar{\epsilon}$ is a constant that holds the value 10^{-6}. The user can always change the value of $\bar{\epsilon}$ by adding the required value as an additional parameter to the FP directive.

3.5.3.2 Renaming the intermediate variables

The Two-Phases adjoint, sketched in figure 3.6 (b), makes use of an intermediate adjoint set of variables $\overline{w}$ which are temporary utility variables that do not correspond exactly to the adjoint of original variables. However, this $\overline{w}$ has the same size and shape as the state z.

For implementation reasons, actual differentiation of the loop body is performed by a recursive call to the standard differentiation mechanism, which systematically names the adjoint variables after their original variables, so that $\overline{w}$ will actually be named $\overline{z}$. The adjoint loop body must therefore have the form: $\overline{z}_{k+1} = \overline{z}_k . \frac{\partial}{\partial z}\phi(z_*, x)$. To accommodate this form, we transformed the BWD sweep of the FP adjoint, introducing in $\overline{z}_{orig}$ a copy of the $\overline{z}$, yielding the equivalent formulation shown in figure 3.15.

3.5.4 Specifying the transformation on Control Flow Graphs

As far as a theoretical description is concerned, it is perfectly acceptable to represent a FP loop with a simple body consisting of a call to ϕ. However, for real codes this assumption is too strong. We need to specify the adjoint transformation, so it can be applied to any structure of FP loops, possibly nested, that respect the conditions of subsection 3.4. Since these structures of interest inside a Control Flow Graph are obviously nested, the natural structure to capture them is a tree. Therefore, our strategy is to superimpose a tree of nested **Flow Graph Levels** (FGLs) on the Control Flow Graph of any subroutine.

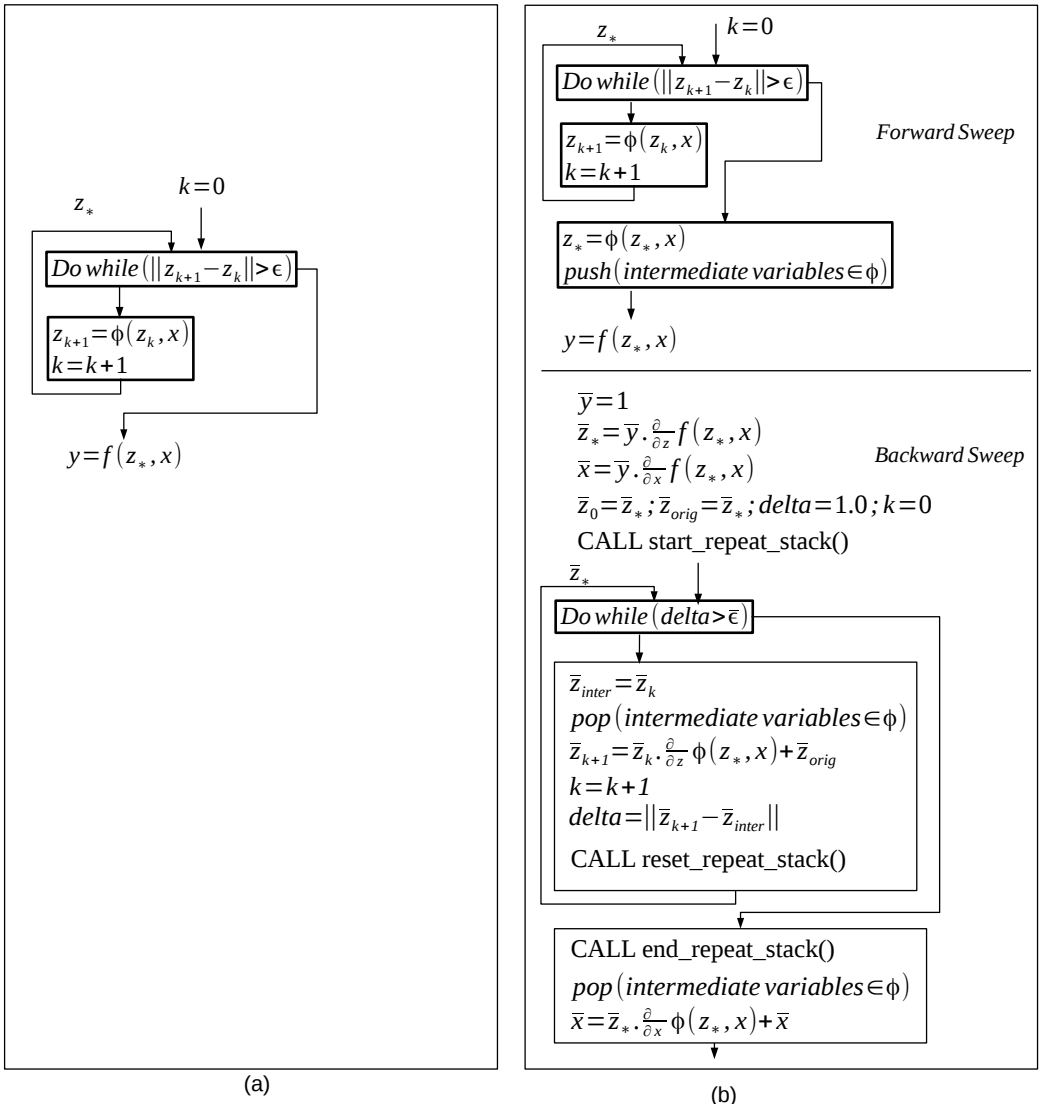

FIGURE 3.15: (a) Example of code that contains a FP loop. (b) Two-Phases applied to this code after renaming the intermediate variables.

A FGL is either a single Basic Block or a graph of deeper FGLs. This way, the adjoint of a FGL is defined as a new FGL that connects the adjoints of the child FGLs and a few Basic Blocks required by the transformation. Adjoining a Flow Graph is thus a recursive transformation on the FGLs. Every enclosing FGL needs to know about its children FGLs, their entry point, which is a single flow arrow, and their exit points, which may be many, i.e. many arrows. We introduce a level in the tree of nested FGLs, containing a particular piece of code, to express that this piece has a specific, probably more efficient adjoint. For instance, we introduce such a level for parallel loops, time-stepping loops, plain loops, and now for FP loops.

Specifically for a FP loop, the original FGL (see figure 3.16 (left)) is composed of a loop header Basic Block and a single child FGL for the loop body. We arbitrarily place two cycling arrows after the loop body to represent the general case where one FGL may have

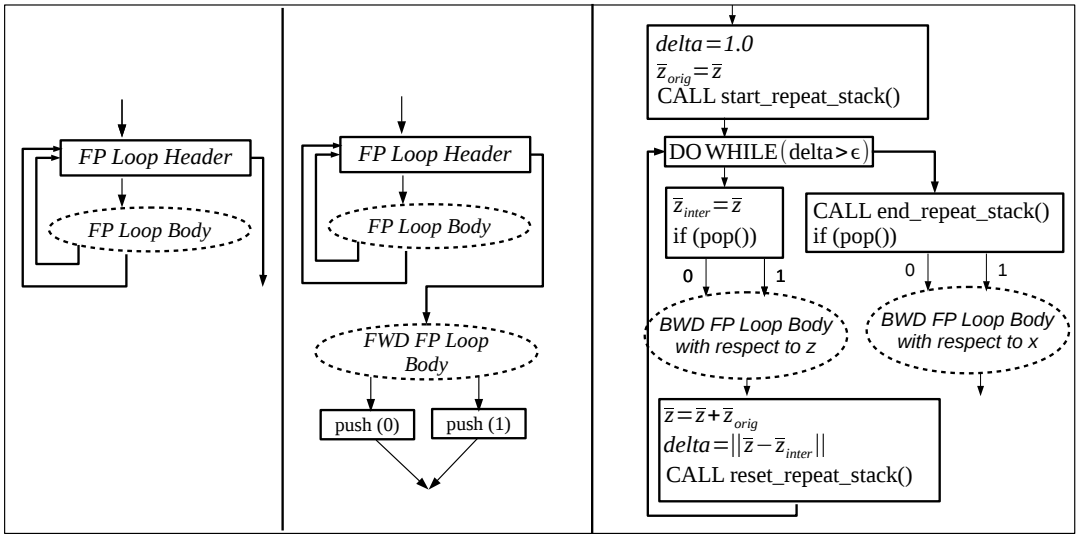

FIGURE 3.16: left: flow graph level of a Fixed-Point loop, middle: flow graph level of the FWD sweep of this Fixed-Point loop, right: flow graph level of the BWD sweep of this Fixed-Point loop

several exit points. The FWD sweep of the FP loop adjoint (see figure 3.16 (middle)) basically copies the original loop structure, but inserts after this loop the FWD sweep of the adjoint of the loop body, thus storing intermediate values only for the last iteration. The BWD sweep (see figure 3.16 (right)) introduces several new Basic Blocks to hold:

- the calls that enable a repeated access to the stack.
- the computation of the variation of $\overline{z}$ into a variable **delta** which is used in the exit condition of the while loop.
- the initial storage of $\overline{z}$ into $\overline{z}_{orig}$ and its use at the end of each iteration.

The FWD and BWD sweeps of the FP loop body, resulting recursively from the adjoint differentiation of the loop body FGL are new FGL's represented in figure 3.16 by oval dashed boxes. They are connected to the new Basic Blocks as shown. The characteristic of the adjoint of a FP loop, visible in figure 3.15, is that the FP body must be differentiated twice, once with respect to z and once with respect to x. This accounts for the two FGL (oval dashed boxes) in figure 3.16, that stand for the two different adjoint BWD sweeps of the loop body.

3.5.5 Differentiation of the loop body in two different contexts

We see that the code produced following the Two-Phases method has a fixed, ad-hoc skeleton that contains in two places some pieces of code that can be produced by standard

adjoint AD. These two pieces appear respectively in the algorithm sketched in figure 3.15 as:

$$\overline{\mathtt{z}} = \overline{\mathtt{z}}.\frac{\partial}{\partial \mathtt{z}}\phi(\mathtt{z},\mathtt{x})$$

and

$$\overline{\mathtt{x}} = \overline{\mathtt{x}} + \overline{\mathtt{z}}.\frac{\partial}{\partial \mathtt{x}}\phi(\mathtt{z},\mathtt{x})$$

Therefore, these two pieces can and must be generated automatically with an AD tool such as Tapenade. One might just decide to recuperate those from the naive adjoint code produced by the AD tool. Actually the naive adjoint code effectively contains a part, the adjoint of $\mathtt{z} = \phi(\mathtt{z},\mathtt{x})$, that computes $\overline{x} = \overline{x} + \overline{z}.\frac{\partial}{\partial x}\phi(z,x)$; $\overline{z} = \overline{z}.\frac{\partial}{\partial z}\phi(z,x)$. But these computations are blended and almost impossible to separate. Consequently the final specialized Two-Phases adjoint, although already well optimized in terms of memory, would duplicate code and repeatedly run useless parts of the derivative computation. The solution to this problem is to perform differentiation of $\phi(\mathtt{z},\mathtt{x})$ *twice*. One differentiation will be specialized to produce code for:

$$\overline{x} = \overline{x} + \overline{z}\frac{\partial}{\partial x}\phi(z,x)$$

only, and the other differentiation, separate from the first, will produce code for:

$$\overline{z} = \overline{z}\frac{\partial}{\partial z}\phi(z,x).$$

This can be arranged for, but with special care on the usual questions: what is the function to differentiate, for which of its inputs and for which of its outputs?

At this point we need to start a discussion on the notion of "independent" and "dependent" parameters. Source-transformation AD classically features an analysis phase, followed by a code generation phase that actually builds the differentiated code. In particular the so-called "`Activity` analysis" , see subsection 2.3.2, has a strong influence on the future differentiated code. `Activity` analysis detects, for each occurrence of a variable in the code, whether this variable is active or not. An active variable is such that its derivative is at the same time (a) not trivially (i.e. structurally) zero and (b) needed for later computations. Conversely when a variable is *not* active, the differentiated code can be simplified, sometimes vastly. It is therefore essential for the efficiency of the adjoint code that the detected active occurrences of variables form a set as small as possible. Overapproximation on the set of active variables is unavoidable in general, but we must strive to keep it minimal.

In turn, let us consider the ingredients to `Activity` analysis. `Activity` analysis on a piece of code F takes as input:

- The independents, which is the subset of the inputs of F with respect to which a derivative will be required. In other words the derivative code of F will be used in a context that will use some derivatives with respect to the independents. The independents are the ingredient of the first half of `Activity` analysis, known as the "`varied`" analysis. The `varied` analysis propagates, forwards through the source of F, the variables whose current value may depend in a differentiable way on the independents.

- The dependents, which is the subset of the outputs of F of which a derivative will be required. In other words the derivative code of F will be used in a context that will use some derivatives of the dependents. The dependents are the ingredient of the second half of `Activity` analysis, known as the "`useful`" analysis. The `useful` analysis propagates, backwards through the source of F, the variable whose current value may have a differentiable influence on the dependents.

In the sequel, we will use the following equivalences between notions, that directly result from the definitions. The following three statements are equivalent:

- A given occurrence of variable `v` is `varied`.
- At the corresponding location in the tangent code, $\dot{\mathtt{v}}$ is not trivially zero.
- At the corresponding location in the adjoint code, $\overline{\mathtt{v}}$ is needed by the following adjoint computations.

Similarly, the following three are equivalent:

- A given occurrence of variable `v` is `useful`.
- At the corresponding location in the tangent code, $\dot{\mathtt{v}}$ is needed by the following tangent computations.
- At the corresponding location in the adjoint code, $\overline{\mathtt{v}}$ is not trivially zero.

A variable occurrence will be considered active if it is at the same time `varied` and `useful`. It is therefore essential to provide AD with the right sets of independents and dependents to let it produce correct and efficient derivative code.

Going back to the production of efficient code for $\frac{\partial}{\partial x}\phi(z,x)$ on the one hand, and for $\frac{\partial}{\partial z}\phi(z,x)$ on the other hand, the question is to find the right sets of (in)dependents for these two separate invocations of AD on $\phi(\mathtt{z},\mathtt{x})$. Let us consider first that z is the set of

variables that are overwritten inside the FP loop and x is the set of variables that are only read inside this loop. Obviously running AD on $\phi(\mathtt{z},\mathtt{x})$ with the standard `Activity` information obtained by analysis of the full code, will result on the code that we already have and that computes $\frac{\partial}{\partial x}\phi(z,x)$ and $\frac{\partial}{\partial z}\phi(z,x)$ jointly. So we would gain nothing. Still, choosing the (in)dependent sets for either $\frac{\partial}{\partial x}\phi(z,x)$ or $\frac{\partial}{\partial z}\phi(z,x)$ is delicate and error-prone, so let us deduce these sets safely from an analysis of the new adjoint Fixed-Point algorithm, sketched again in figure 3.17.

We have identified intermediate points in the algorithm, named `P1` to `P8`. Some points

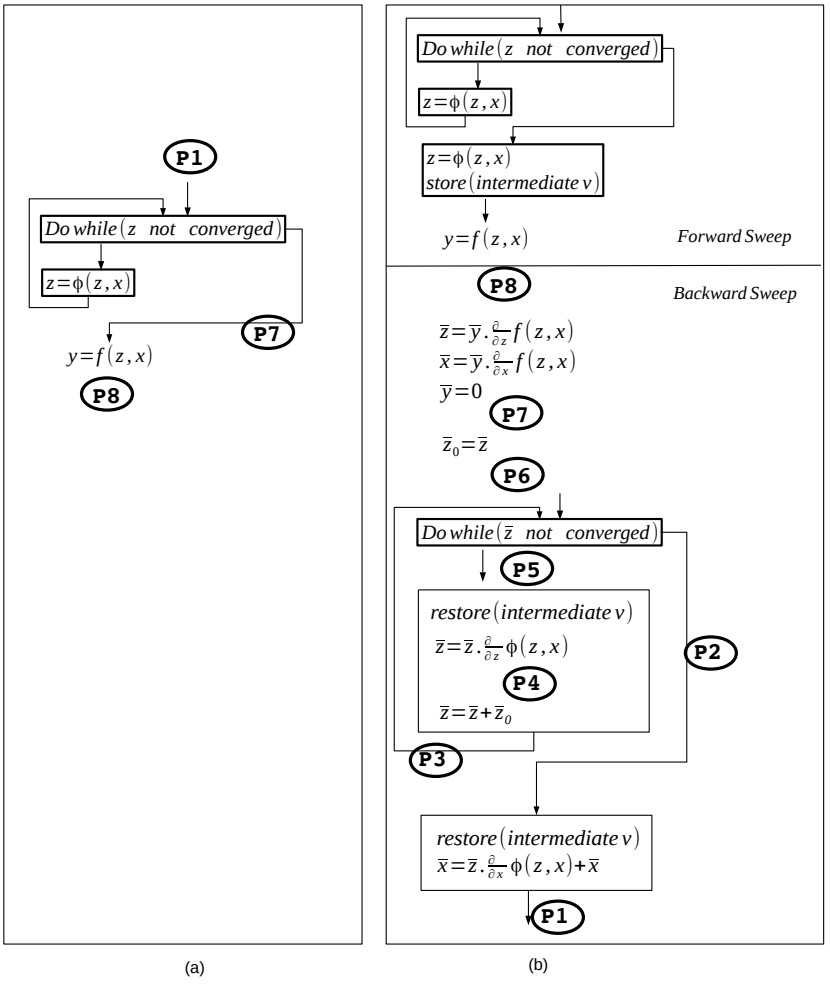

FIGURE 3.17: (a) A Fixed-Point loop. (b) Two-Phases method applied to this loop. In Two-Phases we need to specify which are the dependents and independents for $\frac{\partial}{\partial z}\phi(z,x)$ and $\frac{\partial}{\partial x}\phi(z,x)$.

(`P1, P7, P8`) have a corresponding location in the original, non-differentiated Fixed-Point loop. The other points have no correspondent because they are specific to the skeleton of the special adjoint algorithm.

Consider the "`useful`" analysis first. It is a backward analysis so what we have to begin with is $\mathrm{U}_{\mathtt{P7}}$ the "`useful`" variables at point `P7`, which are known from `Activity`

analysis of standard differentiation ("`useful`" phase). From now on, we view $\mathrm{U}_{\mathtt{P7}}$ as the variables whose adjoint is not trivially zero. Let us now proceed through figure 3.17. As we are proceeding through the backward sweep of the adjoint code, we are actually going forwards through figure 3.17:

- $\mathrm{U}_{\mathtt{P6}}$ is the same as $\mathrm{U}_{\mathtt{P7}}$, as $\overline{\mathtt{z}}_0$ is an artificial copy of $\overline{\mathtt{z}}$ not involved in `Activity` analysis.
- $\mathrm{U}_{\mathtt{P5}}$ is the union of $\mathrm{U}_{\mathtt{P6}}$ and $\mathrm{U}_{\mathtt{P3}}$ (which we don't know yet).
- $\mathrm{U}_{\mathtt{P4}}$ will be found by running the "`useful`" analysis through $\mathtt{z} = \phi(\mathtt{z}, \mathtt{x})$: view $\mathrm{U}_{\mathtt{P5}}$ equivalently as the `useful` output variables of $\mathtt{z} = \phi(\mathtt{z}, \mathtt{x})$, and the analysis propagation through this instruction returns us with the `useful` input variables of $\mathtt{z} = \phi(\mathtt{z}, \mathtt{x})$, i.e. $\mathrm{U}_{\mathtt{P4}}$.
- $\mathrm{U}_{\mathtt{P3}}$ is the union of $\mathrm{U}_{\mathtt{P4}}$ and $\mathrm{U}_{\mathtt{P6}}$, as the resulting $\overline{\mathtt{z}}$ becomes non-trivial-zero either because it was already so or because the corresponding z_0 is so.
- $\mathrm{U}_{\mathtt{P2}}$ is the same as $\mathrm{U}_{\mathtt{P3}}$ because the loop iterates at least once.
- $\mathrm{U}_{\mathtt{P1}}$ will be found by running the "`useful`" analysis through $\mathtt{z} = \phi(\mathtt{z}, \mathtt{x})$ with $\mathrm{U}_{\mathtt{P2}}$ as the `useful` outputs of this instruction.

Consider now the "`varied`" analysis. It is a forward analysis so what we have to begin with is $\mathrm{V}_{\mathtt{P1}}$ the "`varied`" variables at point `P1`, which are known from `Activity` analysis of standard differentiation ("`varied`" phase). From now on, we view $\mathrm{V}_{\mathtt{P1}}$ as the variables whose adjoint is needed by following adjoint computations. Let us now proceed through figure 3.17. As we are proceeding through the backward sweep of the adjoint code, we are actually going backwards through figure 3.17:

- $\mathrm{V}_{\mathtt{P2}}$ will be found by running the "`varied`" analysis through $\mathtt{z} = \phi(\mathtt{z}, \mathtt{x})$: view $\mathrm{V}_{\mathtt{P1}}$ equivalently as the `varied` input variables of $\mathtt{z} = \phi(\mathtt{z}, \mathtt{x})$, and the analysis propagation through this instruction returns us with the `varied` output variables of $\mathtt{z} = \phi(\mathtt{z}, \mathtt{x})$, i.e. $\mathrm{V}_{\mathtt{P2}}$.
- $\mathrm{V}_{\mathtt{P3}}$ is the union of $\mathrm{V}_{\mathtt{P2}}$ and $\mathrm{V}_{\mathtt{P5}}$ (which we don't know yet).
- $\mathrm{V}_{\mathtt{P4}}$ is the same as $\mathrm{V}_{\mathtt{P3}}$, as the instruction in between is just an increment that doesn't change the subset of $\overline{\mathtt{z}}$ needed in the sequel of the adjoint code.
- $\mathrm{V}_{\mathtt{P5}}$ will be obtained by running the "`varied`" analysis through $\mathtt{z} = \phi(\mathtt{z}, \mathtt{x})$ with $\mathrm{V}_{\mathtt{P4}}$ as the `varied` inputs of this instruction. $\mathrm{V}_{\mathtt{P5}}$ will be found as the resulting `varied` outputs of $\mathtt{z} = \phi(\mathtt{z}, \mathtt{x})$.

- $V_{\texttt{P6}}$ is the same as $V_{\texttt{P5}}$ because the loop iterates at least once.
- $V_{\texttt{P7}}$ is the same as $V_{\texttt{P6}}$.

From the above, we find easily the independent and dependent sets for the separate differentiation of $\phi(\texttt{z},\texttt{x})$ that will produce $\frac{\partial}{\partial \texttt{x}}\phi(\texttt{z},\texttt{x})$. Things are slightly more intricate for $\frac{\partial}{\partial \texttt{z}}\phi(\texttt{z},\texttt{x})$, as there is a loop involved: there are possibly several propagations through $\phi(\texttt{z},\texttt{x})$, for both "`varied`" and "`useful`" analysis, with inputs that may differ from an iteration to the other because of $U_{\texttt{P3}}$ and $V_{\texttt{P5}}$ respectively. In other words, we need a fixed point in these data-flow analyses. Fortunately, this is exactly what happens already for data-flow analysis of loop. It is easy to see that the propagations detailed above are exactly those that occur in the "`varied`" and "`useful`" analysis not of $\phi(\texttt{z},\texttt{x})$ alone, but rather of the original Fixed-Point loop itself `do while`$(...)$ $\texttt{z} = \phi(\texttt{z},\texttt{x})$ `enddo`.

To summarize, we propose the following sequence of data-flow analysis to obtain an optimized `Activity` information, resulting in an optimized adjoint code for $\frac{\partial}{\partial x}\phi(z,x)$ and $\frac{\partial}{\partial z}\phi(z,x)$:

1. Build Cx, a temporary copy of code $\texttt{z} = \phi(\texttt{z},\texttt{x})$ to hold `Activity` information for $\frac{\partial}{\partial x}\phi(z,x)$.
2. Build Cz, another temporary copy of code `do while`$(...)$ $\texttt{z} = \phi(\texttt{z},\texttt{x})$ `enddo` to hold `Activity` information for $\frac{\partial}{\partial z}\phi(z,x)$.
3. Retrieve $V_{\texttt{P1}}$ from the standard "`varied`" analysis at point `P1`.
4. Retrieve $U_{\texttt{P7}}$ from the standard "`useful`" analysis at point `P7`.
5. Run "`varied`" analysis on Cx, feeding in $V_{\texttt{P1}}$ as its `varied` inputs, obtaining $V_{\texttt{P2}}$ as its `varied` outputs.
6. Run "`varied`" analysis on Cz, feeding in $V_{\texttt{P2}}$ as its `varied` inputs.
7. Run "`useful`" analysis on Cz, feeding in $U_{\texttt{P7}}$ as its `useful` outputs, obtaining $U_{\texttt{P2}}$ as its `useful` inputs.
8. Run "`useful`" analysis on Cx, feeding in $U_{\texttt{P2}}$ as its `useful` outputs.

Differentiation of Cx with its resulting `Activity` information will produce optimized code for $\overline{x} = \overline{x} + \overline{z}\frac{\partial}{\partial x}\phi(z,x)$. Differentiation of the loop body of Cz with its resulting `Activity` information will produce optimized code for $\overline{z} = \overline{z}\frac{\partial}{\partial z}\phi(z,x)$.

Along the lines of the `Activity` analysis, one may run two specific `TBR` analyses, see subsection 2.3.4, one for $\frac{\partial}{\partial z}\phi(z,x)$ and another for $\frac{\partial}{\partial x}\phi(z,x)$ in order to reduce the

number of values stored in memory. These two analyses can take advantage from the results of the `Activity` analysis described above. However, in Two-Phases method, the last iteration saves in memory all the intermediate values that are needed at the same time in $\frac{\partial}{\partial z}\phi(z,x)$ and in $\frac{\partial}{\partial x}\phi(z,x)$. Consequently, the variables `to be recorded` during the last iteration have to be the union of the results of the two specific `TBR` analyses. Therefore, we think that there is no big benefit we may get from running these special `TBR` analyses.

Similarly, one may run two specific `Diff-liveness` analyses, see subsection 2.3.3, one for $\frac{\partial}{\partial z}\phi(z,x)$ and another for $\frac{\partial}{\partial x}\phi(z,x)$. These two specific analyses aim to reduce the number of primal instructions that appear in the last iteration of the FP loop. These two specific analyses can also take advantage from the results of the `Activity` analysis described above. However, in Two-Phases approach the primal instructions of the last iteration compute the intermediate values that are needed at the same time in $\frac{\partial}{\partial z}\phi(z,x)$ and in $\frac{\partial}{\partial x}\phi(z,x)$. Consequently, what will be used during the generation of the last iteration is actually the union of the results of the two specific `Diff-liveness` analyses. Therefore, we think here also that there is no big benefit we may get from running these special `Diff-liveness` analyses.

In our implementation of the Two-Phases method, the forward sweep of the adjoint copies the original loop and inserts after it the forward sweep of the adjoint of the loop body, see figure 3.16. Since the forward sweep of the loop body is actually a copy of the loop body in which the intermediate values are saved, this part of code does not contain only the computation of ϕ, but also the computations of the stopping criterion of the FP loop. As there is no need to keep these computations outside the loop, we may run a specific `Diff-liveness` analysis on the FP loop. In this analysis, we specify that each variable used inside the header of the loop is not `diffLive`. Consequently, during the following iterations of the iterative process, all the instructions that compute these variables become $\texttt{non} - \texttt{diffLive}$ as well and therefore do not appear in the forward sweep of the FP loop body.

3.6 Checkpointing inside the Two-Phases adjoint

In this subsection, we show how the checkpointing mechanism applied on a piece of code C inside the FP loop may reduce the efficiency of the Two-Phases adjoint. Instead of saving the intermediate values of this piece of code only once during the FW sweep of the adjoint and retrieving these values many times during the BWD sweep, the checkpointed code will save and retrieve these values as many times as needed to converge the adjoint loop. At the end, the checkpointed code reduces the peak memory consumption of

saving the intermediate values of C only once which is negligible in comparison with the execution time cost.

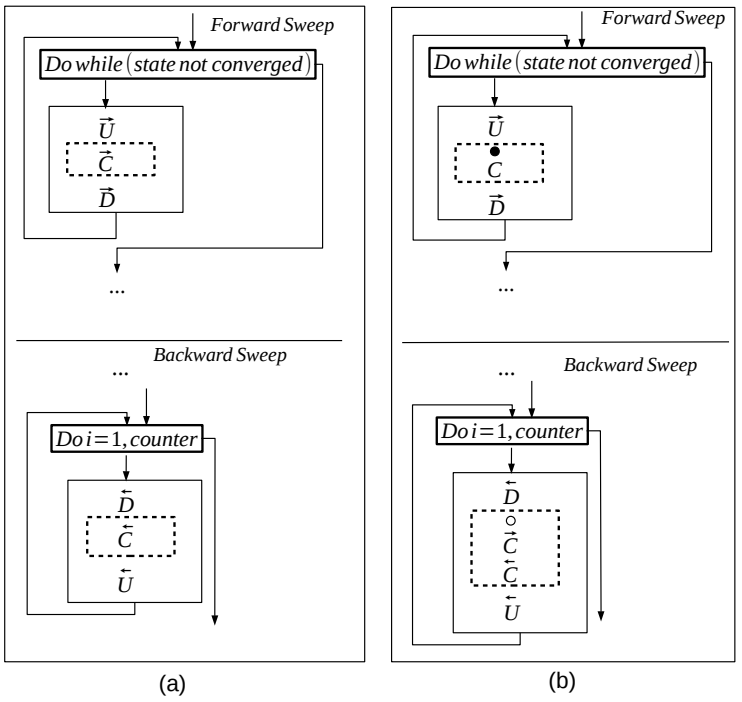

FIGURE 3.18: (a) The Black Box approach applied on a FP loop. (b) The black Box approach applied on a FP loop in which we checkpoint a piece of code "C". We call "U" the piece of code before "C" and "D" the piece of code after "C". The black dot reflects the storage of the snapshot and the white dot reflects its retrieval.

We saw in chapter 2, that in general checkpointing reduces the peak memory consumption of saving the intermediate values of a piece of code, at the cost of re-executing this piece of code another time during the BWD sweep of the adjoint. This cost includes also the storage of the snapshot. When this piece of code is actually contained in a loop as shown in figure 3.18 (b), the total cost is multiplied by the number of iterations of this loop. However, the number of times we save and retrieve the intermediate values of this piece of code remains unchanged. This means that, if the intermediate values of the piece of code `C` are saved and retrieved `n` times in the non-checkpointed code, see figure 3.18 (a), these values will be saved and retrieved `n` times as well in the checkpointed code, see figure 3.18 (b). We recall here, that the intermediate values are saved during the FWD sweep of `C`, $\overrightarrow{\texttt{C}}$, and retrieved during the BWD sweep of `C`, $\overleftarrow{\texttt{C}}$. The cost in terms of time of checkpointing a piece of code `C` in a loop, `ckpTimeCost`, may be formalized as:

$$\texttt{ckpTimeCost} = \texttt{n} * \texttt{timeCost}(\texttt{C}) + \texttt{n} * \texttt{timeCost}(\bullet) + \texttt{n} * \texttt{timeCost}(\circ)$$

where, `n` is the number of iterations of the original loop and $\bullet$ and $\circ$ reflect saving and retrieving the snapshot.

At this time cost, checkpointing reduces the peak memory consumption of:

$$\texttt{n} * (\texttt{memoryCost(C)} - \texttt{memoryCost}(\bullet))$$

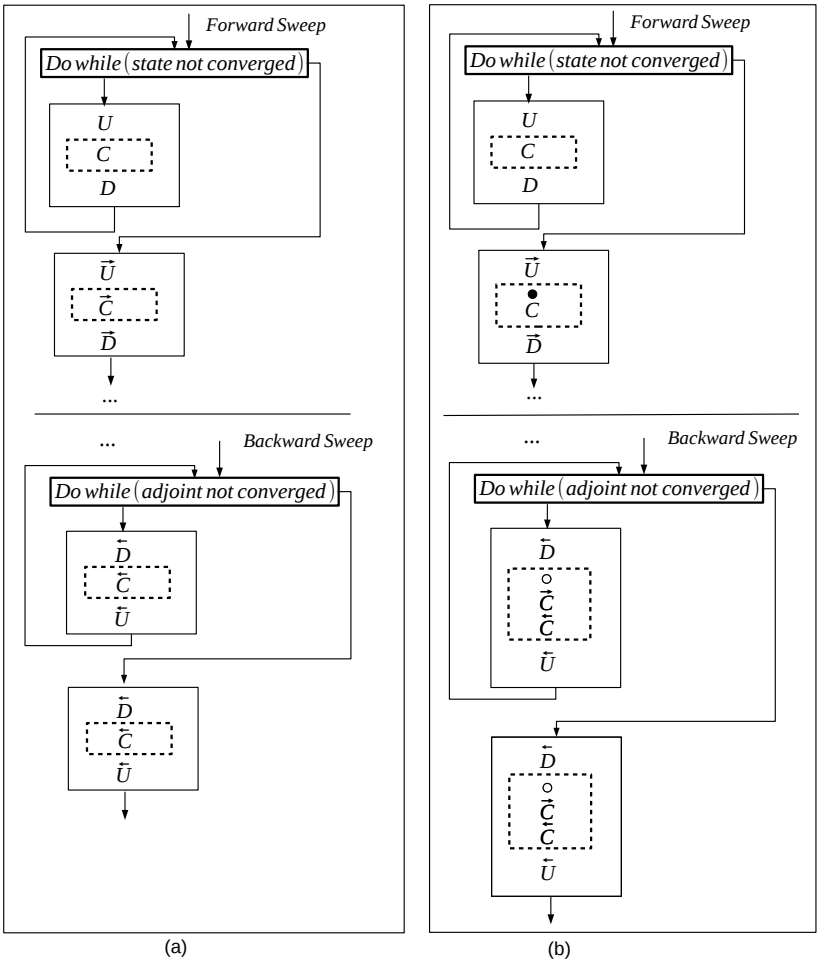

FIGURE 3.19: (a) The Two-Phases adjoint applied on a FP loop. (b) The Two-Phases adjoint applied on a FP loop in which we checkpoint a piece of code "C". We call "U" the piece of code before "C" and "D" the piece of code after "C". The black dot reflects the storage of the snapshot and the white dot reflects its retrieval.

Now, let us assume that our piece of code C is included inside a FP loop that respects the applicability conditions of the refined two-phases method (see subsection 3.4).
Figure 3.19 (a) shows the application of the Two-Phases adjoint to this loop.
Figure 3.19 (b) shows the application of the Two-Phases adjoint to this loop together with checkpointing the piece of code C. We observe that because of checkpointing, the piece of code C is not only re-executed many times during the BWD sweep of the adjoint, but also its intermediate values are saved and retrieved as many times as needed to converge the adjoint loop. The main difference between checkpointing in the case of the Black Box adjoint and checkpointing in the case of the Two-Phases one is that, in the Black Box adjoint, we already save the intermediate values of C n times and that

checkpointing will not add additional storage of these intermediate values. In the case of the Two-Phases adjoint, however, the non-checkpoined code saves the intermediate values of `C` only once during the FWD sweep of the adjoint, see figure 3.19 (a), and, thus, because of checkpointing we will add `m` storage of these intermediate values, with `m` is the number of iterations of the adjoint loop.
In the case of the Two-Phases adjoint, the time cost of checkpointing the piece of code `C`, `ckpTimeCost`, may be formalized as:

$$\texttt{ckpTimeCost} = \texttt{timeCost}(\texttt{C}) + \texttt{m} * \texttt{timeCost}(\overrightarrow{C}) + \texttt{timeCost}(\bullet) + (\texttt{m} + 1) * \texttt{timeCost}(\circ)$$

At this time cost, checkpointing in the case of the Two-Phases adjoint reduces the peak memory consumption of:

$$(\texttt{memoryCost}(\texttt{C}) - \texttt{memoryCost}(\bullet)),$$

which is actually quite small benefit in comparison with the total cost in terms of execution time.

The Two-Phases adjoint does such a good job at reducing the memory cost (but less importantly the execution time) that application of classical checkpointing inside the FP loop body actually looses a part of this benefit, to a point where it may become counter productive. Therefore, we advise to use caution and carefully evaluate the cost benefit when applying checkpointing inside a FP loop.

3.7 Experiments and performances

To validate our implementation, we selected two different codes: The first is a medium size code that contains a FP loop. This code has been developed at Queen Mary University Of London (QMUL). The second is a home-made code that contains a nested structure of FP loops. The main objective behind the two experiments is to quantify the benefits of the Two-Phases adjoint in terms of memory consumption and accuracy of the final derivatives. We apply the Black Box adjoint, seen in section 3.2.1, as well as the Two-Phases adjoint on both codes. On the second code, we try different initial guess for the inner loop. Some of these initial guess are constant over the outer iterations and some others depend on the results of previous iterations. In subsection 3.7.2.1, we see the implications of these initial guess on the results of the Black Box adjoint as well as the Two-Phases adjoint. In subsection 3.7.2.2, we define the initial guess of the backward inner loop of the Two-Phases adjoint as the result of this loop at the previous outer

iteration. We will see how this new initial guess will reduce significantly the number iterations of the backward adjoint loop.

3.7.1 Experiment on real-medium size code

The real medium-size code named GPDE is a Fortran90 program. It is a an unstructured pressure-based steady-state Navier-Stokes solver with finite volume spatial discretization. It is based on the SIMPLE (Semi-Implicit Method for Pressure Linked Equations)[47] algorithm for incompressible viscous flow computation. The FP loop of the program computes the pressure and velocity (the state variables) of an incompressible flow by using the SIMPLE algorithm. In every iteration, the algorithm computes the velocity by solving the momentum equation. Then it uses the obtained value to compute the pressure via solving the continuity equation.

The FP loop of the program does not originally respect the structure of subsection 3.4. It is a while loop that contains an alternate exit at the middle of the body, so that the last iteration does not sweep through the whole function $\phi(z, x)$. Therefore, to apply the Two-Phases adjoint to the code, we transformed the loop by removing the alternate exit. The transformation was performed by using the peeling method as we did in subsection 3.4.
For comparison, we differentiated the transformed loop with the Black Box adjoint differentiation as well as with the Two-Phases adjoint. To trigger the Two-Phases adjoint, we placed the FP directive, described in subsection 3.5.2, just before the transformed FP loop. Since we don't have a deep knowledge about the mathematical equations behind the code, we relied on the AD tool to detect the variables that form the state and those that form the parameters. The value of $\bar{\epsilon}$ is set to 10^{-6}. We observed a minor benefit on run-time and its link to accuracy. By construction, the Black Box adjoint runs for 66 iterations, which is the iteration count of the original FP loop. On the other hand, the Two-Phases adjoint runs exactly as many times as needed to converge $\bar{z}$. Figure 3.20 shows the error of the adjoint compared with a reference value (obtained by forcing the FP loop to run 151 times) as a function of the number of adjoint iterations for the Two-Phases adjoint.

For the Black Box FP adjoint, which runs exactly 66 iterations by construction, we have only one point on figure 3.20. For the Two-Phases adjoint, we have a curve as the error decreases as iterations go. It takes only 46 iterations to reach the same accuracy ($7.8 * 10^{-5}$) as the Black Box FP adjoint. Moreover, as we left the Two-Phases adjoint converge further, we actually reach a slightly better accuracy. We explain this by the fact that the adjoint is computed using only the fully converged values.

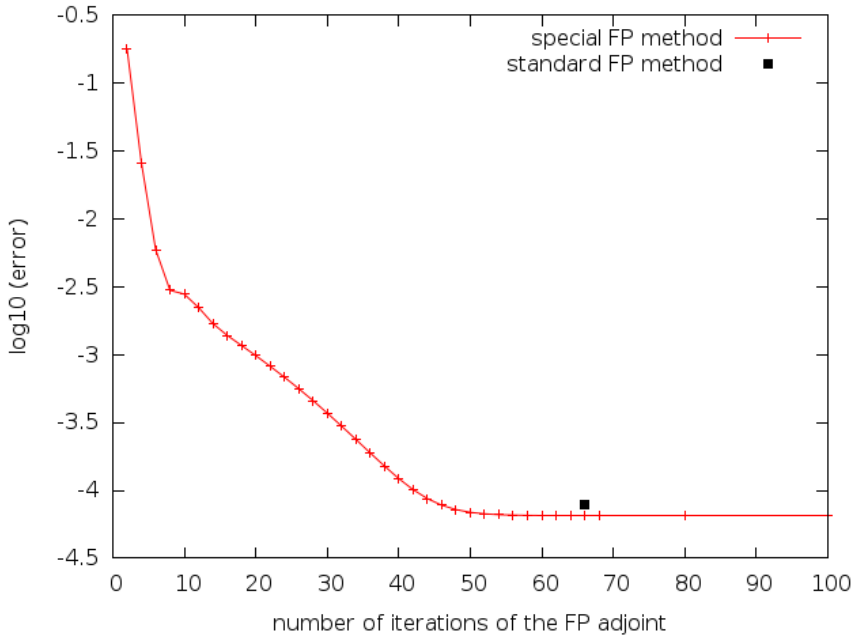

FIGURE 3.20: Error measurements of both Black Box and Two-Phases adjoint methods

Notice however that the principal benefit of the Two-Phases method is not about accuracy nor run time but about reduction of the memory consumption, since the intermediate values are stored only during the last forward iteration. The peak stack space used by the Two-Phases adjoint is 60 times smaller than the space used by the Black Box adjoint (10.1 Mbytes vs. 605.5 Mbytes).

3.7.2 Experiment on nested FP loops

We chose an algorithm that solves for u in an equation similar to a heat equation, with the form :

$$-\Delta u + u^3 = F \tag{3.21}$$

where F is given. The solving algorithm uses two nested Fixed-Point resolutions.

- On the outside is a (pseudo)-time integration, considering that u evolves with time towards the stationary solution $u(\infty)$, following the equation:

$$\frac{u(t+1) - u(t)}{\Delta t} - \Delta u(t+1) + u^3(t) = F \tag{3.22}$$

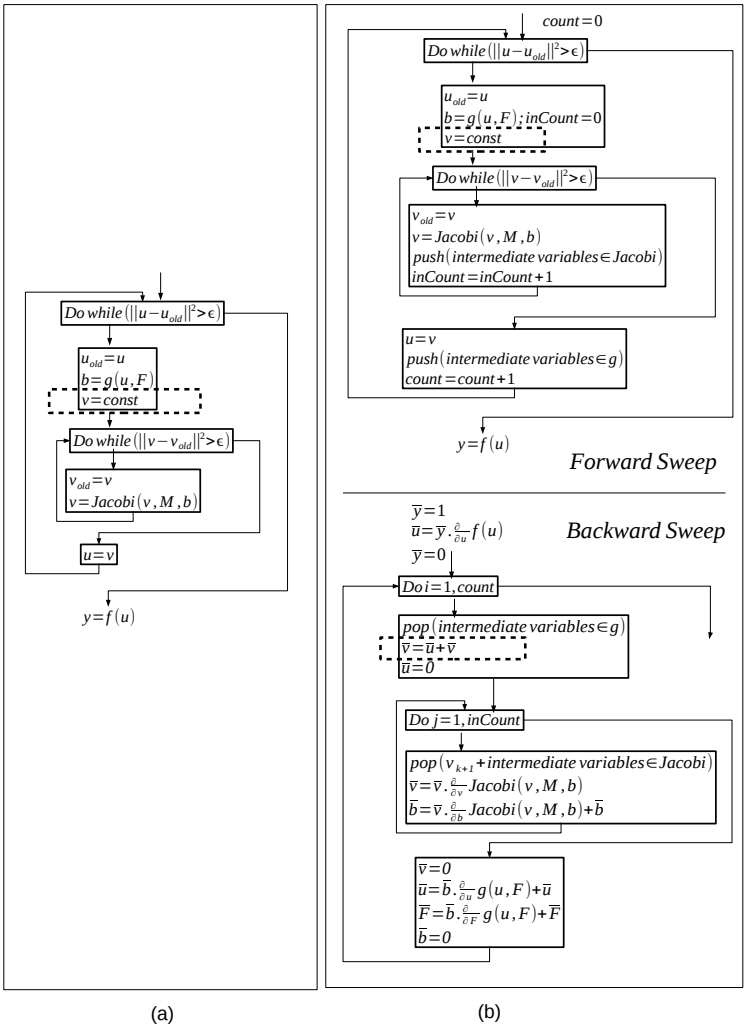

FIGURE 3.21: (a) An algorithm that contains a nested structure of FP loops. The initial guess of the inner loop is constant during the iterations of the outer loop. (b) The Black Box approach applied to this algorithm

- On the inside is the resolution of the implicit equation for $v(t+1)$ (where v is an intermediate variable) as a function of $v(t)$ and F. This resolution uses a Jacobi iteration method, which results in another Fixed-Point algorithm.

This algorithm is sketched in figure 3.21 (a). In figure 3.21 (a), the function g computes for each u and F the value $\frac{u}{\Delta t} - u^3 + F$, the matrix M is defined as $M = \frac{1}{\Delta t} - \Delta$ and the function $Jacobi$ solves for v in the equation $M * v = b$. The initial guess of the inner loop (represented by dashed rectangle) is set constant over the outer iterations, i.e. we placed the instruction $v = const$ at the entry of the inner loop.

We differentiated the algorithm with the Black Box adjoint as well as with the Two-Phases adjoint. To trigger, the Two-Phases adjoint, we placed the FP directive before both loops. Figure 3.21 and 3.22 show the application of respectively the Black Box adjoint and the Two-Phases adjoint on the nested FP loops. For clarity sake, we apply

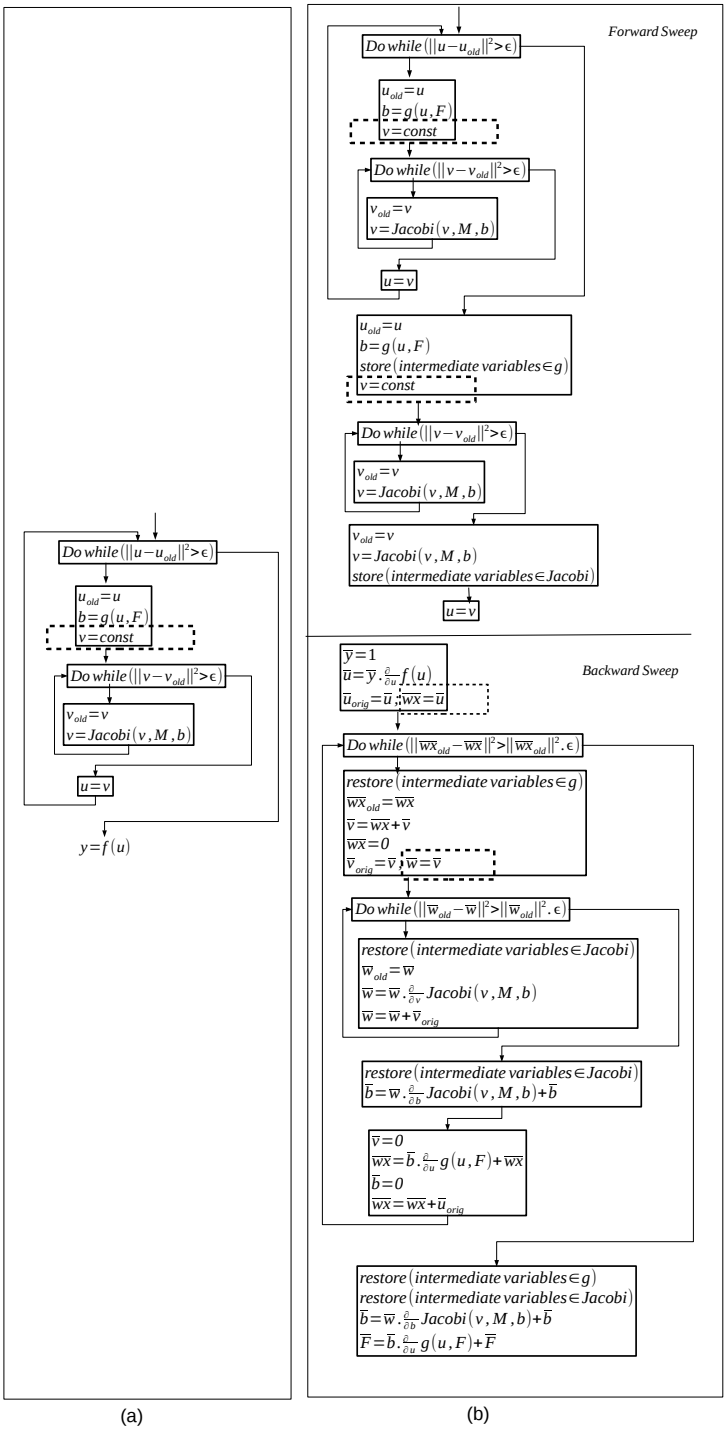

FIGURE 3.22: (a) An algorithm that contains a nested structure of FP loops. The initial guess of the inner loop is constant during the iterations of the outer loop. (b) The Two-Phases approach applied to this algorithm

in figure 3.22 the Two-Phases adjoint as it is described in the theory, i.e. as it is described in subsection 3.2.6, and not as it is implemented in our AD tool. We recall that in our implementation of the Two-Phases adjoint we do not use the intermediate adjoint set of variables $\overline{w}$. This has been explained in subsection 3.5.3.2. To apply the Two-Phases adjoint, we introduced, thus, in figure 3.21 (b), two intermediate variables: $\overline{w}$ which represents the state of the inner adjoint loop and $\overline{wx}$ which represents the state of the outer adjoint loop. For our needs, we chose the stopping criterion of the inner adjoint loop so that it tests at each iteration if $||\overline{w}_{old} - \overline{w}||^2 \leq ||\overline{w}_{old}||^2.\epsilon$ with $||..||$ is the euclidean norm and and $\overline{w}_{old}$ is $\overline{w}$ computed at previous iteration. Similarly, the stopping criterion of the outer adjoint loop tests at each iteration if $||\overline{wx}_{old} - \overline{wx}||^2 \leq ||\overline{wx}_{old}||^2.\epsilon$ with $\overline{wx}_{old}$ is $\overline{wx}$ computed at previous iteration.

We compared performance of the code that uses the Two-Phases adjoint with the one that uses the Black Box adjoint. Performance comparison is made difficult by the fact that the two algorithms do not produce the same result: only the Two-Phases adjoint has a stopping criterion that ensures actual stationarity of the adjoint. We observe that the Two-Phases adjoint iterates slightly fewer times than the Black Box one:

- number of iterations in the outside adjoint FP loop is 289 instead of 337.
- number of iterations in the inside adjoint FP loops is uniformally 34 instead of an average of 44.

However the result is less accurate than with the Black Box adjoint, although inside the prescribed accuracy of 10^{-15}. In other words the Black Box adjoint, being forced to iterate more than necessary for the prescribed accuracy, actually produces a more accurate value. Accuracy is estimated by comparison to a result obtained with a much smaller stationarity criterion (10^{-40}). We then took an alternate viewpoint, forcing the Two-Phases adjoint to iterate as much as the Black Box adjoint and examining the accuracy of the result. The result of the Black Box adjoint deviates from the reference result by $2.1 * 10^{-5}\%$. The result of Two-Phases adjoint adjoint deviates by $1.1 * 10^{-5}\%$. These results are similar to those obtained in section 3.7.1. Again, this may be explained by the fact that the Two-Phases adjoint is computed using only the fully converged values. In this experiment, the major improvement is about the reduction of memory consumption. The peak stack space used by the Black Box adjoint is about 86 Kbytes, whereas the Two-Phases adjoint adjoint uses only a peak stack size of 268 bytes.

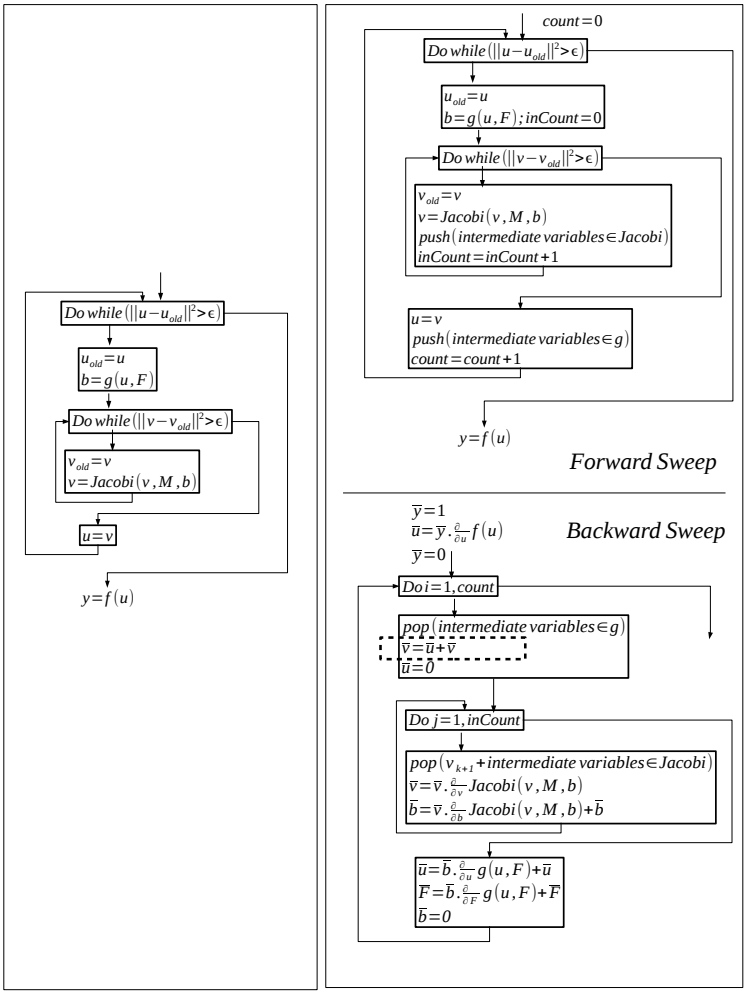

FIGURE 3.23: (a) An algorithm that contains a nested structure of FP loops with a smart initial guess for the inner loop. (b) The Black Box adjoint applied to this algorithm

3.7.2.1 Smart initial guess for the inner loop

We will now look at the choice of the initial guess. We modified the initial guess of the inner loop so that it holds the value of the state computed by the same loop at the previous outer iteration. To do so, we omit the instruction $v = const$ situated at the beginning of the inner loop, see figures 3.21 (a) and 3.23 (a), since in our example the variable v is never modified outside the inner loop.

As result, the original program iterates fewer than in the case where the initial guess is constant over the outer iterations. Actually the total number of inner iterations, which is the sum of the number of inner iterations over the outer iterations, is 8788 instead of 14491. We call this initial guess "smart initial guess" since it reduces the number of inner iterations without reducing the accuracy of the final results.

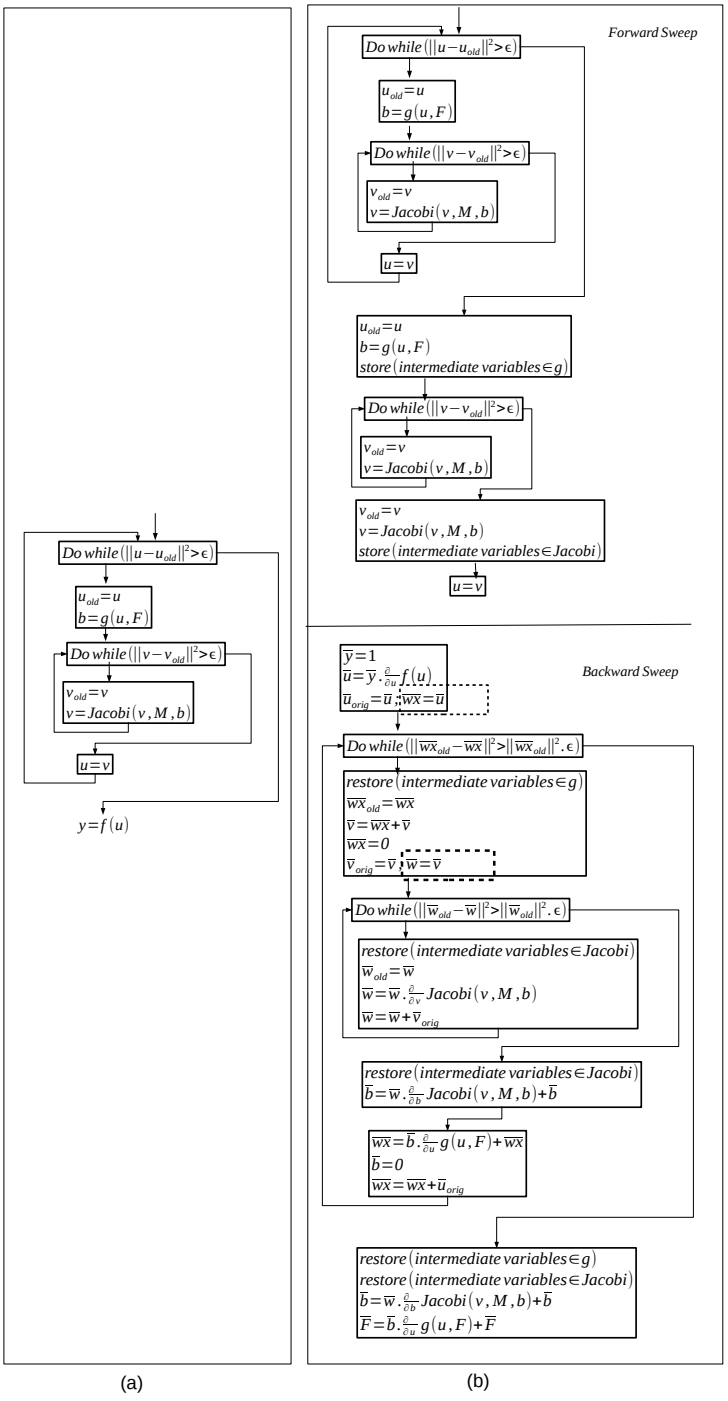

FIGURE 3.24: (a) An algorithm that contains a nested structure of FP loops with a smart initial guess for the inner loop. (b) The Two-Phases adjoint applied to this algorithm

By construction, the Black Box adjoint performs also 8788 inner iterations. Contrary to expectations, see subsection 3.2.1, this reduction of number of iterations did not reduce the accuracy of the final gradient. Actually, the Black Box adjoint with the smart initial guess for the original inner loop performs slightly better, i.e. it deviates from the reference result by $2.0 * 10^{-}5$ % whereas the Black Box adjoint with the non-smart initial guess deviates by $2.1 * 10^{-}5$ %. This may be explained by the fact that when the initial guess of the inner loop is smart, i.e. it depends on the value of the state, v, computed by the same loop at the previous outer iteration, the initial guess of the inner adjoint loop becomes smart as well, i.e. it depends on the adjoint of the state, $\overline{v}$, computed by the same loop at the previous outer iteration.
Actually, We see in figures 3.21 and 3.23 that by construction, the initial guess of the inner adjoint loop (represented by dashed rectangle) is the sum of the values of $\overline{u}$ and $\overline{v}$ which are the adjoints of respectively u and v computed at the previous iteration.
In figure 3.21 (a), we place the instruction $v = const$ above the inner loop, to express that the initial guess of this loop is constant over the outer iterations. Since the adjoint of the instruction $v = const$ is by definition the instruction $\overline{v} = 0$, see figure 3.21 (b), the initial guess of the inner adjoint loop which is the sum of $\overline{u}$ and $\overline{v}$ depends in this case only on $\overline{u}$. This means that in the Black Box adjoint, when the initial guess of the original inner loop is independent from the state computed by this loop at the previous iteration, the initial guess of the inner adjoint loop becomes independent as well from the adjoint of the state computed by the adjoint loop at the previous iteration.
We notice that we are talking about the direct dependency here. Actually, in our case $\overline{u}$ depends on $\overline{b}$ which depends on its side on $\overline{v}$ computed inside the inner adjoint loop. This means that the initial guess of the adjoint inner loop depends indirectly on $\overline{v}$. However, this type of dependency is not the subject of our discussion here.
In figure 3.23 (a), there is no instruction that express the stationarity of the initial guess over the outer iterations, i.e. the instruction $v = const$ does not appear. Consequently, see figure 3.23 (b), the value of the adjoint of the state is not null any more at the exit of the adjoint inner loop, i.e. there is no instruction $\overline{v} = 0$ after the adjoint inner loop. Thus, in this case, the initial guess of the inner adjoint loop which is the sum of the values of $\overline{u}$ and $\overline{v}$ depends on the value of $\overline{v}$ computed inside the adjoint inner loop at the iterations before. This means that in the Black Box adjoint, when the initial guess of the original inner loop depends on the state computed by this loop at the previous iteration, the initial guess of the inner adjoint loop becomes dependent as well on the adjoint of the state computed by the adjoint loop at the previous iteration. This explains why in this experiment the reduction of the number of iterations did not reduce the accuracy of the final gradient. We say here that in the Black Box adjoint, the initial guess of the adjoint loop inherits the smartness of the initial guess of the original one.

Unlike the Black Box adjoint, the Two-Phases adjoint performs exactly the same number of iterations as in the case where the inner loop of the original program has no smart initial guess, i.e. the total number of inner iterations is 10132 . The accuracy of the final gradient remains unchanged as well, i.e. it deviates from the reference by $1.2 * 10^{-4}$ %. This result, however, may be explained by the fact that, in this case, the initial guess of the inner adjoint loop did not inherit the smartness of the initial guess of the original inner loop as it is the case of the Black Box adjoint. The main reason behind it, is that inside the adjoint loop, the Two-Phases approach does not compute the adjoint of the state, i.e. it does not compute the real $\overline{v}$, but rather an intermediate variable that is similar to it, i.e. it computes $\overline{w}$.
Actually, as recommended in subsection 3.2.6, we set the initial guess of the inner adjoint loop of the Two-Phases adjoint so that it holds the value of $\overline{v}$ resulting from the upstream computations, see figures 3.22 (b) and 3.24 (b). In our case, the value of $\overline{v}$ is the sum of the values of $\overline{v}$ and $\overline{wx}$ computed at the previous outer iteration. We see in figure 3.24 (b), that $\overline{v}$ is never modified inside the adjoint inner loop, i.e. inside the adjoint inner loop we compute rather the set of intermediate variables $\overline{w}$. This means that at each outer iteration, the initial guess of the inner adjoint loop (represented by dashed rectangle) depends mainly on $\overline{wx}$ computed at the previous iteration. Consequently, the initial guess of the adjoint inner loop does not depend on $\overline{w}$ computed inside the adjoint inner loop. This behavior is clearly different from the one of the Black Box approach where the initial guess of the inner adjoint loop depends on the value of $\overline{v}$ computed by this loop at the previous outer iteration. This may explain why in the case of the Two-Phases adjoint, setting a smart initial guess for the original inner loop did not improve the accuracy of the final gradient as it is the case of the Black Box adjoint.
We may notice here, that when the initial guess of the original inner loop is constant over the outer iterations, see figure 3.22, the initial guess of the inner adjoint loop of the Two-Phases adjoint does not depend as well on the value of $\overline{w}$ computed inside the adjoint inner loop.

At the end, we may see that when the inner loop of our original program has a a smart initial guess that takes advantage from the computations just before, applying the Black Box adjoint is more efficient in terms of number of iterations as well as in terms of accuracy than applying the Two-Phases adjoint, i.e. number of inner iterations 8788 vs 10132 and deviation from the reference $2 * 10^{-5}$ % vs $1.2 * 10^{-4}$ %.

3.7.2.2 Smart initial guess for the Two-Phases adjoint

We saw in subsection 3.7.2.1, that when the initial guess of the original inner loop uses the value of the state from the previous outer iteration, the initial guess of the

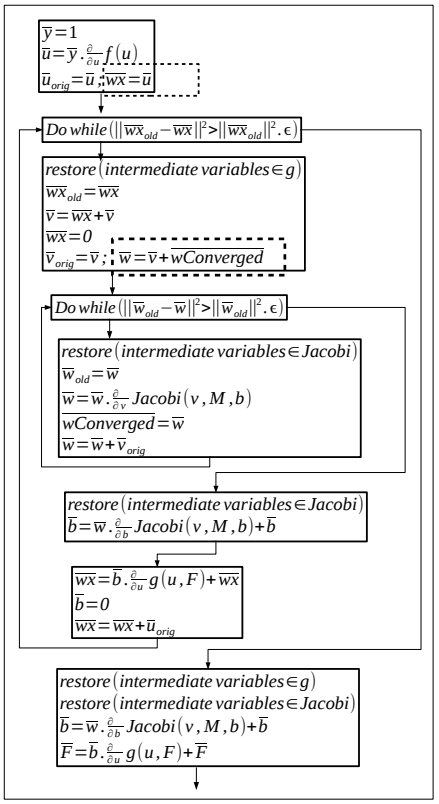

FIGURE 3.25: The backward sweep of the Two-Phases adjoint with a smart initial guess for the inner adjoint loop. The Two-Phases adjoint is applied on a nested structure of FP loops in which the inner loop has a smart initial guess.

adjoint inner loop of the Black Box approach uses, consequently, the value of the adjoint of the state from the previous outer iteration. The initial guess is considered here as smart because it takes advantages from the previous computations. Along these lines, we define a smart initial guess for the inner adjoint loop of the Two-Phases adjoint. This new initial guess does not use only the value of $\overline{v}$ resulting from the upstream computations, as it is the case in the figures 3.22 (b) and 3.24 (b), but also the value of the intermediate variable $\overline{w}$ computed inside the inner adjoint loop. More precisely, the new initial guess is the sum of the values of $\overline{v}$ and $\overline{wConverged}$, which is a new variable that holds the converged value of $\overline{w}$, see figure 3.25.

As results, the new initial guess has decreased the total number of inner iterations by almost half, i.e. 5219 instead of 10132 and slightly improved the accuracy of the final gradient, i.e. the deviation from the reference is $1.0 * 10^{-4}$ % instead of $1.2 * 10^{-4}$ %. These results are obtained by applying the Two-Phases adjoint on the original program whatever the initial guess of its inner loop is, i.e. we have the same results whether we apply the Two-Phases adjoint on the original program with a smart initial guess for the inner loop or we apply this Two-Phases adjoint on the original program with a constant initial guess for the inner loop.

	Black Box adjoint	Two-Phases adjoint	Two-Phases adjoint with smart initial guess for the inner adjoint loop
Original program	Iterations: 14491 Deviation: $2.1 * 10^{-5}$ %	Iterations: 10132 Deviation: $1.2 * 10^{-4}$ % If iterations = 14491 then deviation = $1.1 * 10^{-5}$ %	Iterations: 5219 Deviation: 10^{-4} % If iterations = 14491 then deviation = 10^{-5} %
Original program with smart initial guess for the inner loop	Iterations: 8788 Deviation: $2 * 10^{-5}$ %	Iterations: 10132 Deviation: $1.2 * 10^{-4}$ % If iterations = 8788 then deviation = $3.4 * 10^{-3}$ %	Iterations: 5219 Deviation: 10^{-4} % If iterations = 8788 then deviation = $9.8 * 10^{-6}$ %

TABLE 3.1: Results of applying Black Box and Two-Phases approach on a nested structure of FP loops

Table 3.1 summarizes the results of the Black Box adjoint, the Two-Phases adjoint without smart initial guess for the inner adjoint loop and the Two-Phases adjoint with the smart initial guess in the different cases. In this table, we see that the Two-Phases adjoint with the smart initial guess is the most efficient in terms of accuracy. For instance, when the original program has a smart initial guess for the inner loop and for the same number of iterations 8788, the Two-Phases adjoint deviates from the reference of $3.4 * 10^{-3}$ %, the Black Box adjoint deviates of $2 * 10^{-5}$ % and the Two-Phases adjoint with the smart initial guess for the inner adjoint loop deviates of $9.8 * 10^{-6}$ %.

3.8 Conclusion and further work

We are seeking to improve performance of adjoint codes produced by the adjoint mode of Automatic Differentiation in the frequent case of Fixed-Point loops, for which several authors have proposed adapted adjoint strategies. We explained why we consider the strategy initially proposed by Christianson as the best suited for our needs. In this chapter we described the way we implemented this strategy our the AD tool Tapenade. We experimented this strategy on a some a real medium size code and quantified its benefits, which are marginal in terms of run-time, and significant in terms of memory consumption. We studied the related question of the initial guess in the case of nested iterations.

There are a number of questions that might be studied further to achieve better results and wider applicability:

Theoretical numerical analysis papers discuss the question of the best stopping criterion for the adjoint fixed point loop. However these criteria seem far too theoretical for an automated implementation. In this implementation, the stopping criterion of the adjoint loop is reasonable, but so far arbitrary. It might be interesting if we could in the future derive it mechanically from the original loop's stopping criterion, perhaps using software analysis rather than numerical analysis.

In many applications, the FP loop is enclosed in another loop and the code takes advantage of this to use the result of the previous FP loop as a smart initial guess for the next FP loop. We believe that the adjoint FP loop can use a similar mechanism, even if the variable $\overline{w}$ is not clearly related to some variable of the original code. We made such experiments by reusing the previous $\overline{w}$, see subsection 3.7.2.2. The number of inner adjoint iterations has been decreased by almost half. It might be interesting to study the choice of the adjoint loop initial guess in the general case of nested structures of FP loops.

We have stated a number of restrictions on the structure of candidate FP loops. These are sufficient conditions, but we believe that some restrictions on the shape of FP loops can be lifted at the cost of some loop transformation. The request that the flow of control becomes stationary at the end of the FP loop is essential, and we have no means of checking it statically in general on the source. However, it might be interesting to check it at run-time.

In section 3.5.5 we studied the two repeated separate data-flow analyses that optimize the code generation, for each of the two phases of the Fixed-Point adjoint. The existing manual implementations of this same algorithm that we know, e.g. the compressible discrete adjoint solver of Queen Mary University Of London Mgopt [9], do not go to this level of refinement. In general, they just apply AD twice on the loop body, with a simpler specification of the dependent and independent. We believe that these implementations could be improved by reusing the analysis that we provided.

This adjoint FP loop strategy is for us a first illustration of the interest of differentiating a given piece of code (i.e. ϕ) twice, with respect to different sets of independent variables. This is a change from our tool's original choice, which is to maintain only one differentiated version of each piece of code and therefore to generalize activity contexts to the union of all possible run time activity contexts. Following in this direction, a recent development in our tool allows the user to request many specialized differentiated versions of any given subroutine. This development is another benefit from the AboutFlow European project which funded this thesis, as it was implemented mostly by AboutFlow student (Jan Hückelheim) from Queen Mary University Of London. An article describing the results is in preparation.

Chapter 4

Checkpointing Adjoint MPI-Parallel Programs

4.1 Introduction

Many large-scale computational science applications are parallel programs based on Message-Passing, implemented for instance by using the MPI message passing library [52]. These programs (called "MPI programs") consist of one or more processes that communicate through message exchanges.

In most attempts to apply checkpointing to adjoint MPI codes (the "popular" approach), a number of restrictions apply on the form of communications that occur in the checkpointed piece of code. In many works, these restrictions are not explicit, and an application that does not respect these restrictions may produce erroneous results.

In this chapter, we focus on MPI parallel programs with point-to point communications. We propose techniques to apply checkpointing to these programs, that either do not impose these restrictions, or explicit them so that the end users can verify their applicability. These techniques rely on both adapting the snapshot mechanism of checkpointing and on modifying the behavior of communication calls. We prove that these techniques preserve the semantics of adjoint code. We experiment these techniques on representative codes and we discuss their efficiency in terms of time and memory consumption.

4.1.1 Adjoint MPI parallel programs

There have been several works on AD of MPI parallel programs in general [33], [8], [32] and on the adjoint mode in particular [41], [54]. One point to point communication

`send`(a)`/recv`(b), in which the variable a holds the sent value and the variable b holds the received value, may be considered equivalent to the assignment statement $b = a$. We saw in chapter 2 that the adjoint statements corresponding to $b = a$ are $\overline{a} = \overline{a} + \overline{b}$; $\overline{b} = 0$. Following similar analogy, one may consider the statement $\overline{a} = \overline{a} + \overline{b}$ equivalent to a point to point communication `send`$(\overline{b})$`/recv`$(temp)$, in which $temp$ is an intermediate variable that holds the received value, followed by an increment of $\overline{a}$ by the value of $temp$, i.e. $\overline{a} += temp$. Consequently, we may express:

- the adjoint of the receiving call `recv`(b) as a *send* of the corresponding adjoint value followed by a reset of the value of $\overline{b}$, i.e. $\overline{\texttt{recv}(b)} = \texttt{send}(\overline{b})$; $\overline{b} = 0$.
- the adjoint of the sending call `send`(a) as a *receive* of the corresponding adjoint value followed by an increment of $\overline{a}$ by $temp$, i.e., $\overline{\texttt{send}(a)} = \texttt{recv}(tmp)$; $\overline{a} += temp$

We may consider the blocking call `send`(a) equivalent to the non blocking call `isend`(a, r) followed by its `wait`(r). This means that the adjoint statements corresponding to `isend`(a, r)`; wait`(r) are `recv`(tmp); $\overline{a} += temp$. Since the blocking call `recv`(tmp) may be considered as well equivalent to the non blocking call `irecv`$(temp, r)$ followed by its `wait` (r), the adjoint corresponding to the statements `isend`(a, r)`;` `wait`(r) become `irecv`$(temp, r)$`;` `wait`(r); $\overline{a} += temp$.
Similarly, one may consider the blocking call `recv`(b) equivalent to the non blocking call `irecv`(b, r) followed by its `wait`(r). Following the same steps as in the case of the non blocking *send*, we may find that the adjoint corresponding to the statements `irecv`(b, r)`;` `wait`(r) are `isend`$(\overline{b}, r)$; `wait`(r); $\overline{b} = 0$. Since the adjoint mode is performed in the reverse order of the original program, we may express, thus:

- the adjoint of the non blocking receiving call `irecv`(b, r) as: `wait`(r);$\overline{b} = 0$
- the adjoint of the non blocking sending call `isend`(a, r) as: `wait`(r); $\overline{a} += temp$
- the adjoint of a waiting call `wait`(r) that is paired with an `isend`(a, r) as: `irecv`$(temp, r)$
- the adjoint of a waiting call `wait`(r) that is paired with an `irecv`(b, r) as: `isend`$(\overline{b}, r)$.

A framework that formally proves these rules as well as the rules for adjoining other MPI routines can be found in [41]. In practice, a library called Adjoinable MPI (AMPI) library [54], [48] has been developed in order to make the automatic generation of the adjoint possible in the case of MPI parallel programs. An interface for this library has

already been developed in the operator overloading AD tool dco [49], [53] and under development in our AD tool Tapenade [44]. Further details about this library can be found in subsection 4.5.1.1.

4.1.2 Communications graph of adjoint MPI programs

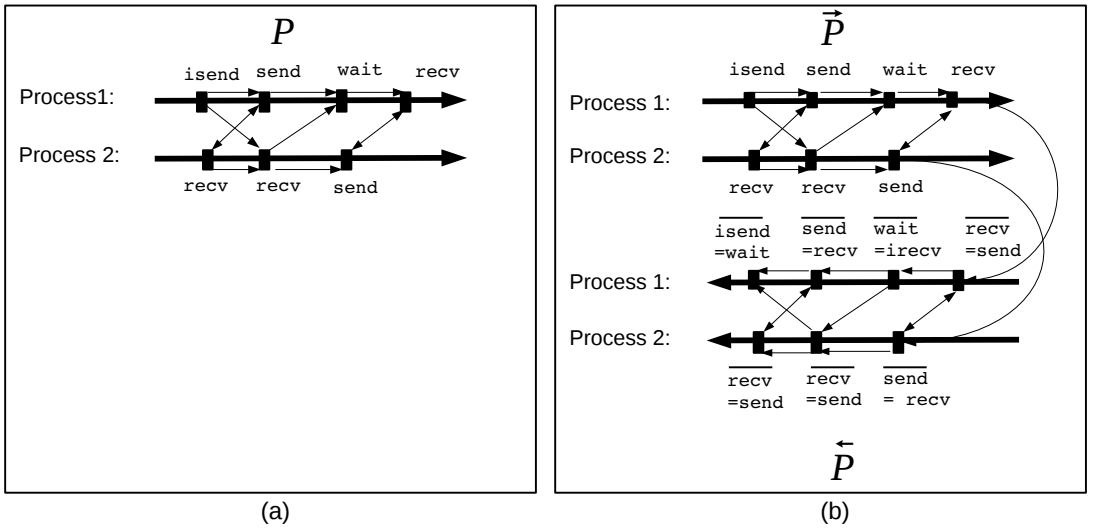

FIGURE 4.1: (a) Communications graph of an MPI parallel program with two processes. Thin arrows represent the edges of the communications graph and thick arrows represent the propagation of the original values by the processes. (b) Communications graph of the corresponding adjoint MPI parallel program. The two thick arrows in the top represent the forward sweep, propagating the values in the same order as the original program, and the two thick arrows in the bottom represent the backward sweep, propagating the gradients in the reverse order of the computation of the original values.

One commonly used model to study message-passing is the communications graph [[52], pp. 399–403], which is a directed graph (see figure 4.1 (a)) in which the nodes are the MPI communication calls and the arrows are the dependencies between these calls. Calls may be dependent because they have to be executed in sequence by a same process, or because they are matching `send` and `recv` calls in different processes.

- The arrow from each `send` to the matching `recv` (or to the `wait` of the matching `isend`) reflects that the `recv` (or the `wait`) cannot complete until the `send` is done. Similarly, the arrow from each `recv` to the matching `send` (or to the `wait` of the matching `irecv`) reflects that the `send` will block until the `recv` is done.
- The arrows between two successive MPI calls within the same process reflect the dependency due to the program execution order, i.e. instructions are executed sequentially. In the sequel, we will not show these arrows.

A central issue for correct MPI programs is to be deadlock free. Deadlocks are cycles in the communications graph.

Given a program P, we denote by $\overrightarrow{P}$ the forward sweep and $\overleftarrow{P}$ the backward sweep of its adjoint $\overline{P}$. Since the adjoint of a sending call is a *receive* of its corresponding adjoint and vice versa, the adjoint code performs a communication of the adjoint value (called "adjoint communication") in the opposite direction of the communication of the primal value, which is what should be done according to the AD model. This creates in the backward sweep $\overleftarrow{P}$ a new graph of communications (see figure 4.1 (b)), that has the same shape as the communications graph of the original program, except for the inversion of the direction of arrows. This implies that if the communications graph of the original program is acyclic, then the communications graph of $\overleftarrow{P}$ is also acyclic. Since the forward sweep $\overrightarrow{P}$ is essentially a copy of the original program P with the same communications structure, the communications graphs of $\overrightarrow{P}$ and $\overleftarrow{P}$ are acyclic if the communications graph of P is acyclic. Since we observe in addition that there is no communication from $\overrightarrow{P}$ to $\overleftarrow{P}$, we conclude that if P is deadlock free, then $\overline{P} = \overrightarrow{P}; \overleftarrow{P}$ is also deadlock free.

4.1.3 Checkpointing

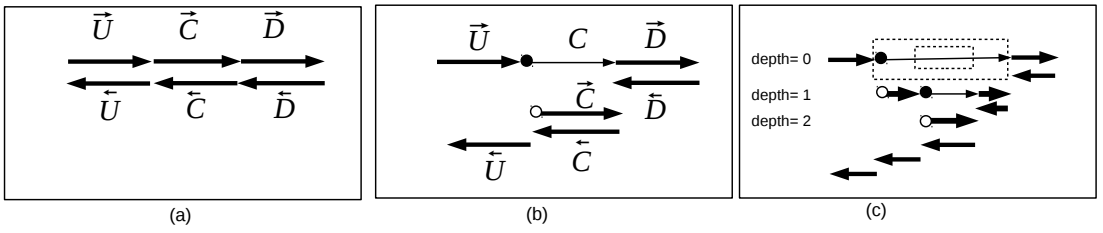

FIGURE 4.2: (a) A sequential adjoint program without checkpointing. (b) The same adjoint program with checkpointing applied to the part of code C. The thin arrow reflects that the first execution of the checkpointed part of code C does not store the intermediate values in the stack. (c) Application of the checkpointing mechanism on two nested checkpointed parts. The checkpointed parts are represented by dashed rectangles.

Storing all intermediate values in the forward sweep of the adjoint consumes a lot of memory space. In the case of serial programs, the most popular solution is the "checkpointing" mechanism. This mechanism was briefly introduced in Chapter 2. In this chapter, we detail further this mechanism with the objective of introducing some of the notations that will be used in the sequel.

Checkpointing is best described as a transformation applied with respect to a piece of the original code (a "checkpointed part"). For instance figure 4.2 (a) and (b) illustrate checkpointing applied to the piece C of a code, consequently written as $U; C; D$.

On the adjoint code of $U;C;D$ (see figure 4.2 (a)), checkpointing C means in the forward sweep **not** storing the intermediate values during the execution of C. As a consequence, the backward sweep can execute $\overleftarrow{D}$ but lacks the intermediate values necessary to execute $\overleftarrow{C}$. To cope with that, the code after checkpointing (see figure 4.2 (b)) runs the checkpointed piece again, this time storing the intermediate values. The backward sweep can then resume, with $\overleftarrow{C}$ then $\overleftarrow{U}$. In order to execute C twice (actually C and later $\overrightarrow{C}$), one must store (a sufficient part of) the memory state before C and restore it before $\overleftarrow{C}$. This storage is called a *snapshot*, which we represent on figures as a $\bullet$ for taking a snapshot and as a $\circ$ for restoring it. Taking a snapshot "$\bullet$" and restoring it "$\circ$" have the effect of resetting a part of the machine state after "$\circ$" to what it was immediately before "$\bullet$". We will formalize and use this property in the demonstrations that follow. To summarize, for original code $U;C;D$, whose adjoint is $\overrightarrow{U};\overrightarrow{C};\overrightarrow{D};\overleftarrow{D};\overleftarrow{C};\overleftarrow{U}$, checkpointing C transforms the adjoint into $\overrightarrow{U};\bullet;C;\overrightarrow{D};\overleftarrow{D};\circ;\overrightarrow{C};\overleftarrow{C};\overleftarrow{U}$.
The benefit of checkpointing is to reduce the peak size of the stack in which intermediate values are stored: without checkpointing, this peak size is attained at the end of the forward sweep, where the stack contains $k_U \oplus k_C \oplus k_D$, where k_X is the values stored by code X and $\oplus$ is a non commutative operator that reflects adding values to the stack. In contrast, the checkpointed adjoint reaches two maxima $k_U \oplus \bullet \oplus k_D$ after $\overrightarrow{D}$ and $k_U \oplus k_C$ after $\overrightarrow{C}$. The cost of checkpointing is twofold: the snapshot must be stored, generally on the same stack. Obviously, one will apply checkpointing only when the size of the snapshot is much smaller than k_C. The other part of the cost is that C is executed twice, thus increasing run time.

4.1.4 Checkpointing on MPI adjoints

Checkpointing MPI parallel programs is restricted due to MPI communications. In previous works, the "popular" checkpointing approach has been applied in such a way that a checkpointed piece of code always contains both ends of each communication it performs. In other words, no MPI call inside the checkpointed part may communicate with an MPI call which is outside. Furthermore, non-blocking communication calls and their corresponding waits must be both inside or both outside of the checkpointed part. This restriction is often not explicitly mentioned. However, if only one end of a point to point communication is in the checkpointed part, then the above method will produce erroneous code. Consider the example of figure 4.3 (a), in which only the `send` is contained in the checkpointed part. The checkpointing mechanism duplicates the checkpointed part and thus duplicates the `send`. As the matching `recv` is not duplicated, the second `send` is blocked. The same problem arises if only the `recv` is contained in the checkpointed part (see figure 4.3 (b)). The duplicated `recv` is blocked. Figure 4.3

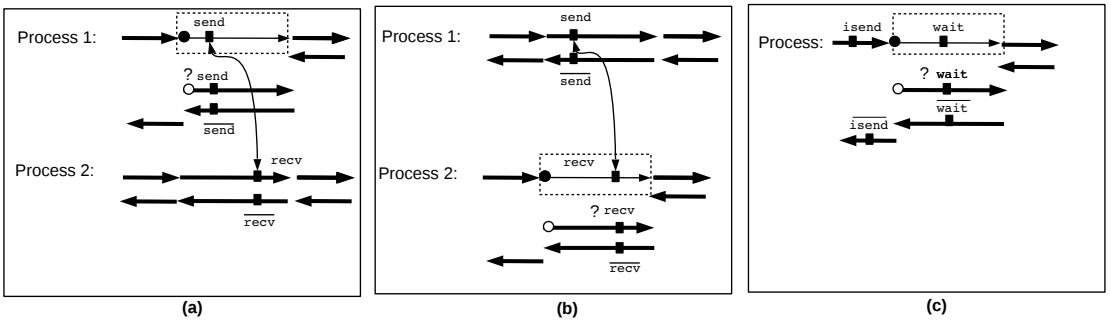

FIGURE 4.3: Three examples of careless application of checkpointing to MPI programs, leading to wrong code. For clarity, we separated processes: process 1 on top and process 2 at the bottom. In (a), an adjoint program after checkpointing a piece of code containing only the *send* part of point-to-point communication. In (b), an adjoint program after checkpointing a piece of code containing only the *recv* part of point-to-point communication. In (c), an adjoint program after checkpointing a piece of code containing a `wait` without its corresponding non blocking routine `isend`.

(c) shows the case of a non-blocking communication followed by its `wait`, and only the `wait` is contained in the checkpointed part. This code fails because the repeated `wait` does not correspond to any pending communication.

We propose techniques that adapt checkpointing to MPI programs, focusing on point-to-point communications. These techniques either do not impose restrictions on the form of communications that occur in the checkpointed part of code, or explicit them so that the end user can verify their applicability. One technique is based on logging the values received, so that the duplicated communications need not take place. Although this technique completely lifts restrictions on checkpointing MPI codes, message logging makes it costly. However, we can refine this technique to replace message logging with communications duplication whenever it is possible, so that the refined technique now encompasses the popular approach. In section 4.2, we give a proof framework for correction of checkpointed MPI adjoint, that will give some sufficient conditions on the MPI adapted checkpointing technique so that the checkpointed adjoint is correct. In section 4.3, we introduce our MPI adapted checkpointing technique based on message logging. We prove that this technique respects the assumptions of section 4.2 and thus that it preserves the semantics of the adjoint code. In section 4.4, we show how this technique may be refined by re-sending messages, in order to reduce the number of values stored in memory. We prove that the refinement we propose respects the assumptions of section 4.2 and thus that it preserves the semantics of the adjoint code as well. In section 4.5, we propose an implementation of our refined technique inside the AMPI library. In section 4.6 and 4.7, we discuss practical questions about the choice of the combination of techniques to be applied within a checkpointed part and the choice of the checkpointed part itself. In section 4.8, we experiment our refined technique on representative codes in which we perform various choices of checkpointed parts. We quantify

the expenses in terms of memory and number of communications for each resulting checkpointed adjoint.

4.2 Elements Of Proof

We propose adaptations of the checkpointing method to MPI adjoint codes, so that it provably preserves the semantics [51] of the resulting adjoint code for any choice of the checkpointed part. To this end, we will first give a proof framework of correction of checkpointed MPI adjoints, that relies on some sufficient conditions on the MPI adapted checkpointing method so that the checkpointed adjoint is correct.

On large codes, checkpointed codes are nested (see figure 4.2 (c)), with a nesting level often as deep as the depth of the call tree. Still, nested checkpointed parts are obtained by repeated application of the simple pattern described in figure 4.2 (b). Specifically, checkpointing applies to any sequence of forward, then backward code (e.g. $\overrightarrow{C}$; $\overleftarrow{C}$ on figure 4.2 (b)) independently of the surrounding code. Therefore, it suffices to prove correctness of one elementary application of checkpointing to obtain correctness for every pattern of nested checkpointed parts.

To compare the semantics of the adjoint codes without and with checkpointing, we define the effect $\mathcal{E}$ of a program P as a function that, given an initial machine state σ, produces a new machine state $\sigma_{new} = \mathcal{E}(P, \sigma)$. The function $\mathcal{E}$ describes the semantics of P. It describes the dependency of the program execution upon all of its inputs and specifies all the program execution results. The function $\mathcal{E}$ is naturally defined on the composition of programs by :
$\mathcal{E}((P_1; P_2), \sigma) = \mathcal{E}(P_2, \mathcal{E}(P_1, \sigma))$.

When P is in fact a parallel program, it consists of several processes p_i run in parallel. Each p_i may execute point-to-point communication calls. We will define the effect $\mathcal{E}$ of one process p. To this end, we need to specify more precisely the contents of the execution state σ for a given process, to represent the messages being sent and received by p. We will call "R" the (partly ordered) collection of messages that will be received (i.e. are expected) during the execution of p. Therefore R is a part of the state σ which is input to the execution of p, and it will be consumed by p. It may well be the case that R is in fact not available at the beginning of p. In real execution, messages will accumulate as they are being sent by other processes. However, we consider R as a part of the input state σ as it represents the communications that are expected by p. Symmetrically, we will call "S" the collection of messages that will be sent during the execution of p. Therefore, S is a part of the state σ_{new} which is output by execution of

p and it is produced by p.
We must adapt the definition of $\mathcal{E}$ for the composition of programs accordingly. We explicit the components of σ as follows. The state σ contains:

- W, the values of variables
- R, the collection of messages expected, or "to be received" by p
- S, the collection of messages emitted by p

With this shape of σ, the form of the semantic function $\mathcal{E}$ and the rule of the composition of programs become more complex. Definition of $\mathcal{E}$ on one process p imposes the prefix R_p of R (the messages to be received) that is required by p and that will be consumed by p. Therefore, the function $\mathcal{E}$ applies pattern matching on its R argument to isolate this "expected" part. Whatever remains in R is propagated to the output R. Similarly, S_P denotes the suffix set of messages emitted by p, to be added to S. Formally, we will write this as:
$\mathcal{E}(p, \langle W, R_P \oplus R, S\rangle) = \langle W', R, S \oplus S_P\rangle$
To explicit the rule of code sequence, suppose that p runs pieces of code C and D in sequence, with C expecting incoming received messages R_C and D expecting incoming received messages R_D. Assuming that the effect of C on the state is:
$\mathcal{E}(C, \langle W, R_C \oplus R, S\rangle) = \langle W', R, S \oplus S_C\rangle$
and the effect of D on the state is:
$\mathcal{E}(D, \langle W', R_D \oplus R, S\rangle) = \langle W'', R, S \oplus S_D\rangle$,
then $C; D$ expects received messages $R_C \oplus R_D$ (for the appropriate concatenation operator $\oplus$) and its effect on the state is:
$\mathcal{E}(C; D, \langle W, R_C \oplus R_D \oplus R, S\rangle) = \langle W'', R, S \oplus S_C \oplus S_D\rangle$.

Adjoint programs operate on two kinds of variables. On one hand, the variables of the original primal code are copied in the adjoint code. In the state σ, we will note their values "V". On the other hand, the adjoint code introduces new adjoint variables to hold the derivatives. In the state σ, we will denote their values "$\overline{V}$".
Moreover, adjoint computations with the store-all approach use a stack to hold the intermediate values that are computed and pushed during the forward sweep $\overrightarrow{P}$ and that are popped and used during the backward sweep $\overleftarrow{P}$. We will denote the stack as "k". In the sequel, we will use a fundamental property of the stack mechanism of AD adjoints, which is that when a piece of code has the shape $\overrightarrow{P}; \overleftarrow{P}$, then the stack is the same before and after this piece of code. To be complete, the state should also describe the sent and received messages corresponding to adjoint values (see section 4.1.2). As these parts of the state play a very minor role in the proofs, we will omit them. Therefore,

we will finally split states σ of a given process as: $\sigma = \langle V, \overline{V}, k, R, S \rangle$.
For our needs, we formalize some classical semantic properties of adjoint programs. These properties can be proved in general, but this is beyond the scope of this paper. We will consider these properties as axioms.

- Any "copied" piece of code X (for instance C) that occurs in the adjoint code operates only on the primal values V and on the R and S communication sets, but not on $\overline{V}$ nor on the stack. Formally, we will write:
  $\mathcal{E}(X, \langle V, \overline{V}, k, R_X \oplus R, S \rangle) = \langle V_{new}, \overline{V}, k, R, S \oplus S_X \rangle$, with the output V_{new} and S_X depending only on V and on R_X.

- Any "forward sweep" piece of code $\overrightarrow{X}$ (for instance $\overrightarrow{U}$, $\overrightarrow{C}$ or $\overrightarrow{D}$) works in the same manner as the original or copied piece X, except that it also pushes on the stack new values noted δk_X, which only depend on V and R_X. Formally, we will write:
  $\mathcal{E}(\overrightarrow{X}, \langle V, \overline{V}, k, R_X \oplus R, S \rangle) = \langle V_{new}, \overline{V}, k \oplus \delta k_X, R, S \oplus S_X \rangle$

- Any "backward sweep" piece of code $\overleftarrow{X}$ (for instance $\overleftarrow{U}$, $\overleftarrow{C}$ or $\overleftarrow{D}$), on one hand operates on the adjoint variables $\overline{V}$ and, on the other hand, uses exactly the top part of the stack δk_X that was pushed by $\overrightarrow{X}$. In the simplest AD model, δk_X is used to restore the values V that were held by the primal variables immediately before the corresponding forward sweep $\overrightarrow{X}$. There exists a popular improvement in the AD model in which this restoration is only partial, restoring only a subset of V to their values before $\overrightarrow{X}$. This improvement called TBR, see subsection 2.3.4, guarantees that the non-restored variables have no influence on the following adjoint computations and therefore need not be stored. The advantage of TBR is to reduce the size of the stack. Without loss of generality, we will assume in the sequel that the full restoration is used, i.e. no TBR is used. With the TBR mechanism, the semantics of the checkpointed adjoint are preserved at least for the output $\overline{V}$ so that this proof is still valid. Formally, we will write:
  $\mathcal{E}(\overleftarrow{X}, \langle V, \overline{V}, k \oplus \delta k_X, R, S \rangle) = \langle V_{new}, \overline{V}_{new}, k, R, S \rangle$, where V_{new} is equal to the value V before running $\overrightarrow{X}$ (which is achieved by using δk_X and V) and $\overline{V}_{new}$ depends only on V, $\overline{V}$ and δk_X.

- A "take snapshot" operation "$\bullet$" for a checkpointed piece C does not modify V nor $\overline{V}$, expects no received messages, and produces no sent messages. It adds into the stack enough values Snp_C to permit a later re-execution of the checkpointed part. Formally, we will write :
  $\mathcal{E}(\bullet, \langle V, \overline{V}, k, R, S \rangle) = \langle V, \overline{V}, k \oplus Snp_C, R, S \rangle$, where Snp_C is a subset of the values in V, thus depending on only V.

- A "restore snapshot" operation "$\circ$" of a checkpointed piece C does not modify $\overline{V}$, expects no received messages and produces no sent messages. It pops from the stack the same set of values Snp_C that the "take snapshot" operation pushed "onto" the stack. This modifies V so that it holds the same values as before the "take snapshot" operation.

  We introduce here the additional assumption that restoring the snapshot may (at least conceptually) add some messages to the output value of R. In particular:

  Assumption 1. The duplicated *recvs* in the checkpointed part will produce the same values as their original calls.

  Formally, we will write:
  $\mathcal{E}(\circ, \langle V, \overline{V}, k \oplus Snp_C, R, S\rangle) = \langle V_{new}, \overline{V}, k, R_C \oplus R, S\rangle$ where V_{new} is the same as V from the state input to the take snapshot.

Our goal is to demonstrate that the checkpointing mechanism preserves the semantics i.e.:

Theorem 4.1. *For any individual process p, for any checkpointed part C of p, (so that p = {U;C:D}), for any state σ and for any checkpointing method that respects the* **Assumption 1***:*

$$\mathcal{E}(\{\overrightarrow{U}; \overrightarrow{C}; \overrightarrow{D}; \overleftarrow{D}; \overleftarrow{C}; \overleftarrow{U}\}, \sigma) = \mathcal{E}(\{\overrightarrow{U}, \bullet, C, \overrightarrow{D}, \overleftarrow{D}, \circ, \overrightarrow{C}, \overleftarrow{C}, \overleftarrow{U}\}, \sigma)$$

Proof. We observe that the non-checkpointed adjoint and the checkpointed adjoint share a common prefix $\overrightarrow{U}$ and also share a common suffix $\overleftarrow{C}$;$\overleftarrow{U}$. Therefore, as far as semantics equivalence is concerned, it suffices to compare $\overrightarrow{C}; \overrightarrow{D}; \overleftarrow{D}$ with $\bullet, C, \overrightarrow{D}, \overleftarrow{D}, \circ, \overrightarrow{C}$.
Therefore, we want to show that for any initial state σ_0 :

$$\mathcal{E}(\{\overrightarrow{C}; \overrightarrow{D}; \overleftarrow{D}\}, \sigma_0) = \mathcal{E}(\{\bullet, C, \overrightarrow{D}, \overleftarrow{D}, \circ, \overrightarrow{C}\}, \sigma_0)$$

Since the semantic function $\mathcal{E}$ performs pattern matching on the R_0 part of its σ_0 argument, and the non-checkpointed adjoint has the shape $\{\overrightarrow{C}; \overrightarrow{D}; \overleftarrow{D}\}$, R_0 matches the pattern $R_C \oplus R_D \oplus R$. Therefore, what we need to show writes as:

$$\mathcal{E}(\{\overrightarrow{C}; \overrightarrow{D}; \overleftarrow{D}\}, \langle V_0, \overline{V}_0, k_0, R_C \oplus R_D \oplus R, S_0\rangle) =$$
$$\mathcal{E}(\{\bullet, C, \overrightarrow{D}, \overleftarrow{D}, \circ, \overrightarrow{C}\}, \langle V_0, \overline{V}_0, k_0, R_C \oplus R_D \oplus R, S_0\rangle)$$

We will call σ_2, σ_3 and σ_6 the intermediate states produced by the non-checkpointed adjoint (see figure 4.4 (a)). Similarly, we call σ'_1, σ'_2, σ'_3, σ'_4, σ'_5, σ'_6 the intermediate states of the checkpointed adjoint (see figure 4.4 (b)). In other words: $\sigma_2 = \mathcal{E}(\overrightarrow{C}, \sigma_0)$;

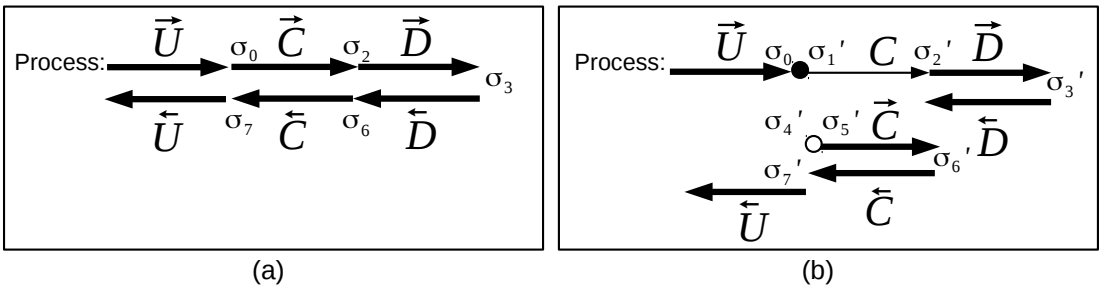

FIGURE 4.4: (a) An adjoint program run on one process. (b) The same adjoint after applying checkpointing to C. The figures show the locations (times) in the execution for the successive states σ_i and σ_i'.

$\sigma_3 = \mathcal{E}(\overrightarrow{D}, \sigma_2)$; $\sigma_6 = \mathcal{E}(\overleftarrow{D}, \sigma_3)$ and similarly $\sigma_1' = \mathcal{E}(\bullet, \sigma_0)$; $\sigma_2' = \mathcal{E}(C, \sigma_1')$; $\sigma_3' = \mathcal{E}(\overrightarrow{D}, \sigma_2')$;
$\sigma_4' = \mathcal{E}(\overleftarrow{D}, \sigma_3')$; $\sigma_5' = \mathcal{E}(\circ, \sigma_4')$;
$\sigma_6' = \mathcal{E}(\overrightarrow{C}, \sigma_5')$.
Our goal is to show that $\sigma_6' = \sigma_6$. Considering first the non-checkpointed adjoint, we propagate the state σ by using the axioms already introduced:

$$\begin{aligned} \sigma_2 \doteqdot \mathcal{E}(\overrightarrow{C}, \sigma_0) &= \mathcal{E}(\overrightarrow{C}, \langle V_0, \overline{V_0}, k_0, R_C \oplus R_D \oplus R, S_0 \rangle) \\ &= \langle V_2, \overline{V_0}, k_0 \oplus \delta k_C, R_D \oplus R, S_0 \oplus S_C \rangle \end{aligned}$$

with V_2, S_C and δk_C depending only on V_0 and R_C. The operator $\doteqdot$ signifies renaming, i.e. the left hand side of this operator is by definition equal to the right hand side.

$$\begin{aligned} \sigma_3 \doteqdot \mathcal{E}(\overrightarrow{D}, \sigma_2) &= \mathcal{E}(\overrightarrow{D}, \langle V_2, \overline{V_0}, k_0 \oplus \delta k_C, R_D \oplus R, S_0 \oplus S_C \rangle) \\ &= \langle V_3, \overline{V_0}, k_0 \oplus \delta k_C \oplus \delta k_D, R, S_0 \oplus S_C \oplus S_D \rangle \end{aligned}$$

with V_3, S_D and δk_D depending only on V_2 and R_D

$$\begin{aligned} \sigma_6 \doteqdot \mathcal{E}(\overleftarrow{D}, \sigma_3) &= \mathcal{E}(\overleftarrow{D}, \langle V_3, \overline{V_0}, k_0 \oplus \delta k_C \oplus \delta k_D, R, S_0 \oplus S_C \oplus S_D \rangle) \\ &= \langle V_2, \overline{V_6}, k_0 \oplus \delta k_C, R, S_0 \oplus S_C \oplus S_D \rangle \end{aligned}$$

with V_2 and $\overline{V}_6$ depending only on V_3, $\overline{V_0}$ and δk_D

Considering now the checkpointed adjoint, we propagate the state σ', starting from $\sigma_0' = \sigma_0$ by using the axioms already introduced:

$$\sigma_1' \doteqdot \mathcal{E}(\bullet, \sigma_0) = \mathcal{E}(\bullet, \langle V_0, \overline{V_0}, k_0, R_C \oplus R_D \oplus R, S_0 \rangle)$$

The snapshot-taking operation • stores a subset of the original values V_0 in the stack "Snp_C".

$$\sigma_1' = \langle V_0, \overline{V_0}, k_0 \oplus Snp_C, R_C \oplus R_D \oplus R, S_0 \rangle$$

$$\sigma_2' \div \mathcal{E}(C, \sigma_1') = \mathcal{E}(C, \langle V_0, \overline{V_0}, k_0 \oplus Snp_C, R_C \oplus R_D \oplus R, S_0 \rangle)$$

The forward sweep of the checkpointed part $\overrightarrow{C}$ is essentially a copy of the checkpointed part C. As the only difference between the two states σ_1' and σ_0 is the stack k and both C and $\overrightarrow{C}$ don't need the stack during run time ($\overrightarrow{C}$ stores values in the stack, but doesn't use it), the effect of C on the state σ_1' produces exactly the same output values V_2 and the same collection of sent values S_C as the effect of $\overrightarrow{C}$ on the state σ_0 .

$$\sigma_2' = \langle V_2, \overline{V_0}, k_0 \oplus Snp_C, R_D \oplus R, S_0 \oplus S_C \rangle$$

The next step is to run $\overrightarrow{D}$:

$$\sigma_3' \div \mathcal{E}(\overrightarrow{D}, \sigma_2') = \mathcal{E}(\overrightarrow{D}, \langle V_2, \overline{V_0}, k_0 \oplus Snp_C, R_D \oplus R, S_0 \oplus S_C \rangle$$

The output state of $\overrightarrow{D}$ uses only the input state's original values V and received values R. As V and R are the same in both σ_2' and σ_2, the effect of $\overrightarrow{D}$ on the state σ_2' produces the same variables values V_3, the same collection of messages sent through MPI communications S_D and the same set of values stored in the stack δk_D as the effect of of $\overrightarrow{D}$ on the state σ_2.

$$\sigma_3' = \langle V_3, \overline{V_0}, k_0 \oplus Snp_C \oplus \delta k_D, R, S_0 \oplus S_C \oplus S_D \rangle$$

Then, the backward sweep starts with the backward sweep of D.

$$\sigma_4' \div \mathcal{E}(\overleftarrow{D}, \sigma_3') = \mathcal{E}(\overleftarrow{D}, \langle V_3, \overline{V_0}, k_0 \oplus Snp_C \oplus \delta k_D, R, S_0 \oplus S_C \oplus S_D \rangle$$

The output state of $\overleftarrow{D}$ uses only its input state's original values V, the values of the adjoint variables $\overline{V}$ and the values stored in the top of the stack δk_D. As V, $\overline{V}$ and δk_D are the same in both σ_3' and σ_3, the effect of $\overleftarrow{D}$ on the state σ_3' produces exactly the same variables values V_2 and the same values of adjoint variables $\overline{V}_6$ as the effect of $\overleftarrow{D}$

on the state σ_3.

$$\sigma'_4 \quad = \quad \langle V_2, \overline{V_6}, k_0 \oplus Snp_C, R, S_0 \oplus S_C \oplus S_D \rangle$$

$$\sigma'_5 \div \mathcal{E}(\circ, \sigma'_4) \quad = \quad \mathcal{E}(\circ, \langle V_2, \overline{V_6}, k_0 \oplus Snp_C, R, S_0 \oplus S_C \oplus S_D \rangle$$

The snapshot-reading operation $\circ$ overwrites V_2 by restoring the original values V_0. According to **Assumption 1**, the snapshot-reading $\circ$ conceptually also restores the collection of values that have been received during the first execution of the checkpointed part R_C.

$$\sigma'_5 \quad = \quad \langle V_0, \overline{V_6}, k_0, R_C \oplus R, S_0 \oplus S_C \oplus S_D \rangle$$

$$\sigma'_6 \div \mathcal{E}(\overrightarrow{C}, \sigma'_5) \quad = \quad \mathcal{E}(\overrightarrow{C}, \langle V_0, \overline{V_6}, k_0, R_C \oplus R, S_0 \oplus S_C \oplus S_D \rangle$$

The output state after $\overrightarrow{C}$ uses only on the input state's values V and the received values R. As V and R are the same in both σ'_5 and σ_0, the effect of $\overrightarrow{C}$ on the state σ'_5 produces the same original values V_2 and the same set of values stored in the stack δk_C as the effect of $\overrightarrow{C}$ on the state σ_0.

$$\sigma'_6 \quad = \quad \langle V_2, \overline{V_6}, k_0 \oplus \delta k_C, R, S_0 \oplus S_C \oplus S_D \rangle$$

Finally we have $\sigma'_6 = \sigma_6$. □

We have shown the preservation of the semantics at the level of one particular process p_i. The semantics preservation at the level of the complete parallel program P requires to show in addition that the collection of messages sent by all individual processes p_i matches the collection of messages expected by all the p_i. At the level of the complete parallel code, the messages expected by one process will originate from other processes and therefore will be in the messages emitted by other processes.
This matching of emitted and received messages depends on the particular parallel communication library used (e.g. MPI) and is driven by specifying communications, tags, etc. Observing the non-checkpointed adjoint first, we have identified the expected *receives* and produced *sends* $S_U \oplus S_C \oplus S_D$ of each process. Since the non-checkpointed adjoint is assumed correct, the collection of $S_U \oplus S_C \oplus S_D$ for all processes p_i matches the collection of $R_U \oplus R_C \oplus R_D$ for all process p_i.
The study of the checkpointed adjoint for process p_i has shown that it can run with the

same expected *receives* $R_U \oplus R_C \oplus R_D$ and produces at the end the same sent values $S_U \oplus S_C \oplus S_D$. This shows that the collected *sends* of the checkpointed version of P matches its collected expected *receives*.

However, matching *sends* with expected *receives* is a necessary but not sufficient con-

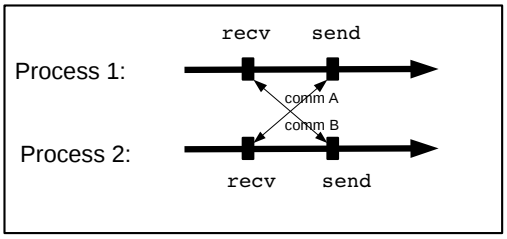

FIGURE 4.5: Example illustrating the risk of deadlock if *send* and *receive* sets are only tested for equality.

dition for correctness. Consider the example of figure 4.5, in which we have two communications between two processes ("comm A" and "comm B"):

- The set of messages that process 1 expects to receive R= {comm B}. The set of messages that it will send is S= {comm A}.
- The set of messages that process 2 expects to receive R= {comm A}. The set of messages that it will send is S= {comm B}.

The above required property that the collection of *sends* {comm A, comm B} matches the collection of *receives* {comm A, comm B} is verified. However, this code will fall into a deadlock.

Semantic equivalence between two parallel programs requires not only that collected *sends* match collected *receives* but also that there is no deadlock. Assuming that we can prove it:

Assumption 2. the resulting adjoint code after checkpointing is deadlock free,

then, the semantics of the checkpointed adjoint is the same as that of its non-checkpointed version.

To sum up, a checkpointing adjoint method adapted to MPI programs is correct if it respects these two assumptions:

Assumption 1. The duplicated *recvs* in the checkpointed part will receive the same values as their original calls.

Assumption 2. The resulting adjoint code after checkpointing is deadlock free.

For instance, the "popular" checkpointing approach that we find in most previous works is correct because the checkpointed part which is duplicated is self-contained regarding communications. Therefore, it is clear that the *receive* operations in that duplicated part receive the same value as their original instances. In addition, the duplicated part, being a complete copy of a part of the original code that does not communicate with the rest, is clearly deadlock free.
We believe, however, that this constraint of a self-contained checkpointed part can be alleviated. We will propose a checkpointing approach that respects our two assumptions for any checkpointed piece of code. We will then study a frequent special case where the cost of our proposed checkpointing approach can be reduced.

4.3 A General MPI-Adjoint Checkpointing Method

We introduce here a general technique that adapts checkpointing to the case of MPI parallel programs and that can be applied to any checkpointed piece of code. This adapted technique, sketched in figure 4.6, is called "receive-logging" technique. It relies on logging every message at the time when it is received.

- During the first execution of the checkpointed part, every communication call is executed normally. However, every *receive* call (in fact its *wait* in the case of non-blocking communication) stores the value it receives into some location local to the process. Calls to *send* are not modified.
- During the duplicated execution of the checkpointed part, every *send* operation does nothing (it is "deactivated"). Every *receive* operation, instead of calling any communication primitive, reads the previously received value from where it has been stored during the first execution.
- The type of storage used to store the received values is First-In-First-Out. This is different from the stack used by the adjoint to store the trajectory.

In the case of nested checkpointed parts, this strategy can either reuse the storage prepared for enclosing checkpointed parts, or free it at the level of the enclosing checkpointed part and re-allocate it at the time of the enclosed checkpoint. This can be managed using the knowledge of the nesting depth of the current checkpointed part.

Notice that this management of storage and retrieval of received values, triggered at the time of the `recv`'s or the `wait`'s, together with nesting depth management, can be implemented by a specialized wrapper around MPI calls, for instance inside the AMPI library. We discuss this further in subsection 4.5.1.

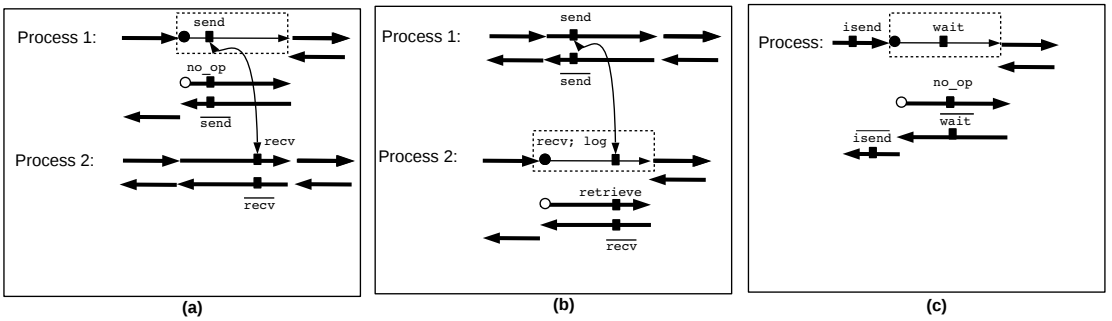

FIGURE 4.6: Three examples in which we apply checkpointing coupled with receive-logging. For clarity, we separated processes: process 1 on top and process 2 at the bottom. In (a), an adjoint program after checkpointing a piece of code containing only the *send* part of point-to-point communication. In (b), an adjoint program after checkpointing a piece of code containing only the *recv* part of point-to-point communication. In (c), an adjoint program after checkpointing a piece of code containing a `wait` without its corresponding non blocking routine `isend`.

To show that this strategy is correct, we will check that it verifies the two assumptions of section 4.2.

4.3.1 Correctness

By construction, this strategy respects **Assumption 1** because the duplicated *receives* read what the initial *receives* have received and stored.

To verify **Assumption 2** about the absence of deadlocks, it suffices to consider one elementary application of checkpointing, shown in the top part of figure 4.7. Communications in the checkpointed adjoint occur only in $\overrightarrow{U}$, C, $\overrightarrow{D}$ (about primal values) on one hand, and in $\overleftarrow{D}$, $\overleftarrow{C}$, $\overleftarrow{U}$ (about derivatives) on the other hand. The bottom part of the figure 4.7 shows the communications graph of the checkpointed adjoint, identifying the sub-graphs of each piece of code. Dotted arrows express execution order, and solid arrows express communication dependency. Communications may be arbitrary between $G_{\overrightarrow{U}}$, G_C and $G_{\overrightarrow{D}}$ but the union of these 3 graphs is the same as for the forward sweep of the non-checkpointed adjoint, so it is acyclic by hypothesis.

Similarly, communications may be arbitrary between $G_{\overleftarrow{D}}$, $G_{\overleftarrow{C}}$ and $G_{\overleftarrow{U}}$ but (as $G_{\overrightarrow{C}}$ is by definition empty) these graphs are the same as for the non-checkpointed backward sweep. Since we assume that the non-checkpointed adjoint is deadlock free, it follows that the checkpointed adjoint is also deadlock free.

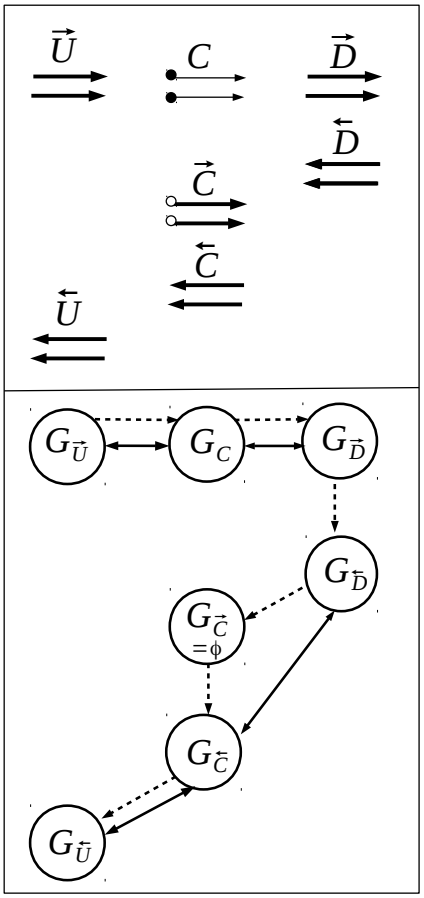

FIGURE 4.7: Communications graph of a checkpointed adjoint with pure receive-logging method

4.3.2 Analogy with "Message logging" in the context of resilience

Checkpointing in the context of AD-Adjoint (Adjoint-checkpointing) has common points with checkpointing in the context of resilience [6] (Resilience-checkpointing). For instance, in both mechanisms processes take snapshots of the values they are computing to be able to restart from these snapshots when it is needed. However, checkpointing in the case of resilience is performed to recover the system after failure, whereas in the case of AD-adjoint, checkpointing is mostly to reduce the peak memory consumption.
There are two types of checkpointing in the context of resilience: the non-coordinated checkpointing, in which every process takes its own checkpoint independently from the other processes and the coordinated checkpointing in which every process has to coordinate with other process before taking its own checkpoint. In the non-coordinated checkpointing coupled with "Message logging"[5], every process saves in a remote storage checkpoints, i.e. complete images of the process memory. It saves also the messages that have been received and every `send` or `recv` event that have been performed. In case of failure, only the failed process restarts from its last checkpoint. It runs exactly in the same way as before the failure, except that it does not perform any `send` call

already done. The restarted process does not perform either any `recv` call already done, but retrieves instead the value that has been received and stored by the `recv` before the failure. Saving the received values during the first execution and retrieving these values during the re-execution of the process remind us the principle of receive-logging described in section 4.3.

4.3.3 Discussion

The receive-logging strategy applies for any choice of the checkpointed piece(s). However, it may have a large overhead in memory. At the end of the general forward sweep of the complete program, for every checkpointed part (of level zero) encountered, we have stored all received values, and none of these values has been used and released yet. This is clearly impractical for large codes.
On the other hand, for checkpointed parts deeply nested, the receive-logging has an acceptable cost as stored values are used quickly and their storage space may be released and used by checkpointed parts to come. We need to come up with a strategy that combines the generality of receive-logging with the memory efficiency of an approach based on re-sending.

4.4 Refinement of the general method: Message Re-sending

We may refine the receive-logging by re-executing communications when possible. The principle is to identify `send-recv` pairs whose ends belong to the same checkpointed part, and to re-execute these communication pairs identically during the duplicated part, thus performing the actual communication twice. Meanwhile, communications with one end not belonging to the checkpointed part are still treated by receive-logging.
Figure 4.8 (b) shows the application of checkpointing coupled with receive-logging technique to some piece of code. In this piece of code, we select a `send-recv` pair and we apply the message-resending to it. As result, see figure 4.8 (c), this pair is re-executed during the duplication of the checkpointed part and the received value is no more logged during the first instance of this checkpointed part.

However, to apply the message-resending, the checkpointed part must obey an extra constraint which we will call “right-tight”. A checkpointed part is “right-tight” if no communication dependency goes from downstream the checkpointed part back to the checkpointed part, i.e. there is no communication dependency arrow going from D to C in the communications graph of the checkpointed adjoint. For instance, there must

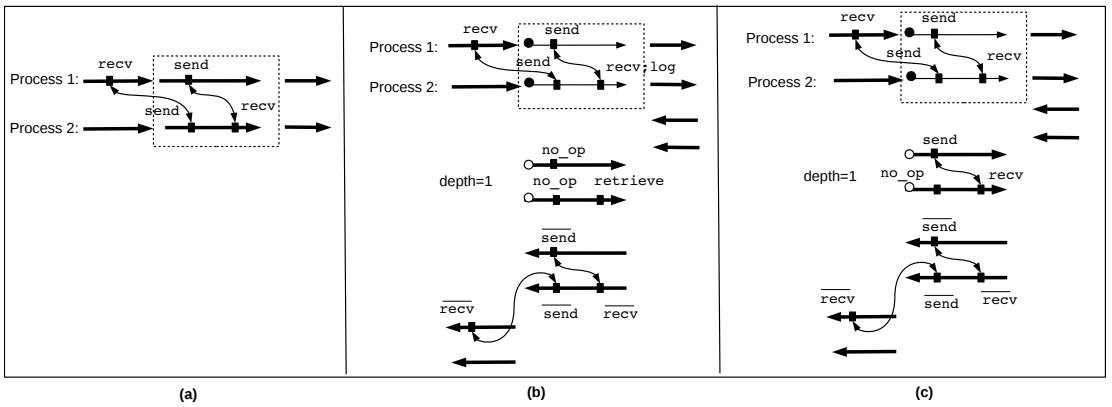

FIGURE 4.8: In (a), an MPI parallel program running in two processes. In (b), the adjoint corresponding to this program after checkpointing a piece of code by applying the receive-logging. In (c), the adjoint corresponding after checkpointing a piece of code by applying the receive-logging coupled the with message-resending.

be no `wait` in the checkpointed part that corresponds with a communication call in an other process which is downstream (i.e. after) the checkpointed part.

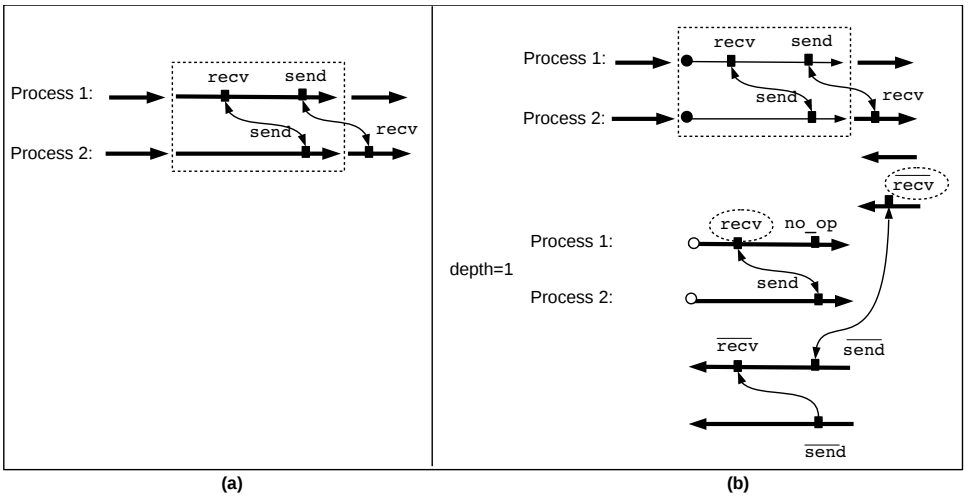

FIGURE 4.9: In (a), an MPI parallel program run on two processes. In (b), the adjoint corresponding after checkpointing a piece of code that is not right-tight by applying the receive-logging coupled with message-resending.

Figure 4.9 shows an example illustrating the danger of applying message re-sending to a checkpointed part which is not right-tight. In Figure 4.9 (a), the checkpointed part is not right-tight as there is a dependency going from the `recv` of process 2 located outside the checkpointed part to the second `send` of process 1 located inside the checkpointed part. If we apply checkpointing to this piece of code by applying message-resending to the `send`-`recv` pair whose ends belong to this checkpointed part and applying receive-logging to the remaining `send`, we obtain figure 4.9 (b) which shows a cycle in the communications graph of the resulting adjoint: between the $\overline{recv}$ of process 2 and the $\overline{send}$ of process 1 takes place the duplicated run of the checkpointed part. In this duplicated run, we find

a duplicated `send-recv` pair that causes a synchronization. Execution thus reaches a deadlock, with process 2 blocked on the $\overline{recv}$, and process 1 blocked on the duplicated `recv`. The $\overline{recv}$ of process 2 and the duplicated `recv` of process 1 are represented by dashed circles in figure 4.9 (b).

One end of communication is called `orphan` with respect to a checkpointed part, if it belongs to this checkpointed part while its partner is not, e.g. `send` that belongs to the checkpointed part while its `recv` is not. In the case where one end of communication is paired with more than one end, e.g. `recv` with wild-card MPI_ANY_SOURCE value for source, this end is considered as `orphan` if one of its partners does not belong to the same checkpointed part as it.

In the general case:

- When the checkpointed part is not right-tight, we can only apply receive-logging to all the ends of communications inside the checkpointed part.
- In the opposite case, i.e. when the checkpointed part is right-tight, we recommend the application of message-resending to all the non-orphan ends of communications that belong to this checkpointed part. For the orphan ones we can only apply receive-logging. The interest of combining the two techniques is that the memory consumption becomes limited to the (possibly few) logged *receives*. The cost of extra communications is tolerable compared to the gain in memory.

4.4.1 Correctness

The subset of the duplicated *receives* that are treated by receive-logging still receive the same value by construction. Concerning the duplicated `send-recv` pair, the duplicated checkpointed part computes the same values as its original execution (see step from σ'_5 to σ'_6 in section 4.2). Therefore the duplicated `send` and the duplicated `recv` transfer the same value.
The proof about the absence of deadlocks is illustrated in figure 4.10. In contrast with the pure receive-logging case, $G_{\overrightarrow{C}}$ is not empty any more because of re-sent communications. $G_{\overrightarrow{C}}$ is a sub-graph of G_C and is therefore acyclic. Since the checkpointed part is right-tight, the dependency from G_C to $G_{\overrightarrow{D}}$ and from $G_{\overleftarrow{D}}$ to $G_{\overleftarrow{C}}$ are unidirectional. There is no communication dependency between $G_{\overrightarrow{C}}$ and $G_{\overrightarrow{D}}$ and $G_{\overleftarrow{C}}$ because $G_{\overrightarrow{C}}$ communicates only primal values and $G_{\overleftarrow{D}}$ an $G_{\overleftarrow{C}}$ communicate only derivative values.
Assuming that the communications graph of the non-checkpointed adjoint is acyclic, it follows that:

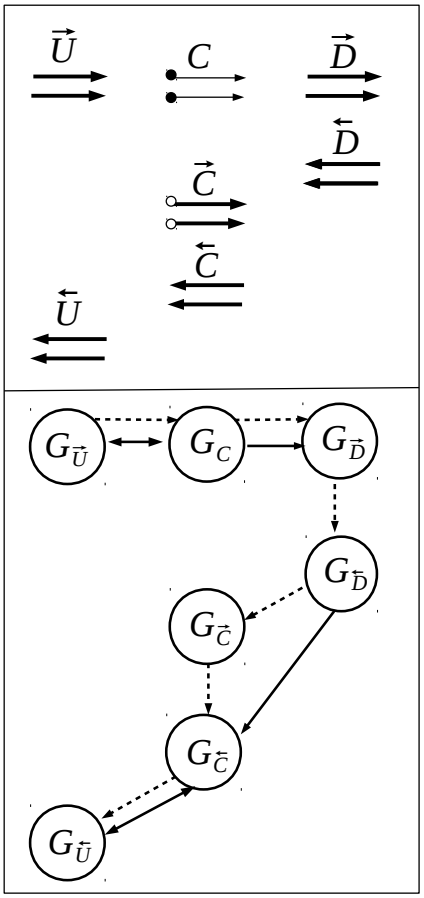

FIGURE 4.10: Communications graph of an adjoint resulting from checkpointing a part of code that is right-tight. Checkpointing is performed by applying message-resending to all the non-orphan ends of communications and receive-logging to all the orphan ones.

- Each of $G_{\overrightarrow{U}}$, $G_{\overrightarrow{C}}$, $G_{\overrightarrow{D}}$, $G_{\overleftarrow{D}}$, $G_{\overleftarrow{C}}$ and $G_{\overleftarrow{U}}$ is acyclic.
- Communications may be arbitrary between $G_{\overrightarrow{U}}$ and G_C but since these pieces of code occur in the same order in the non-checkpointed adjoint, and it is acyclic, there is no cycle involved in $(G_{\overrightarrow{U}}; G_C)$. The same argument applies to $(G_{\overleftarrow{C}}; G_{\overleftarrow{U}})$.

Therefore, the complete graph on the bottom of figure 4.10 is acyclic.

4.5 Combining the receive-logging and message-resending techniques on a nested structure of checkpointed parts

In the general case, we may have a nested structure of checkpointed parts, in which some of the checkpointed parts respect the message-resending conditions of subsection 4.4, i.e. these parts are right-tight, and the others do not respect these conditions. Also, even when all the checkpointed parts respect the message-resending conditions, one end of

communication may be `orphan` with respect to some checkpointed parts and `non-orphan` with respect to the other ones. This means that, for memory reasons, one end of communication may be activated during some depths of the checkpointed adjoint, i.e. we apply the message-resending to this end, and not activated during the other depths, i.e. we apply receive-logging to this end. In the case of *send* operations, combining the receive-logging and message-resending techniques is easy to implement, however, in the case of *receive* operations, this requires a specific behavior. More precisely:

- Every *receive* operation that is activated at depth d calls `recv`. If this operation is de-activated at depth $d+1$, it has to log the received value.
- Every *receive* operation that is de-activated at depth d reads the previously received value from where it has been stored. If this *receive* is activated at depth $d+1$, it has to free the logged value.

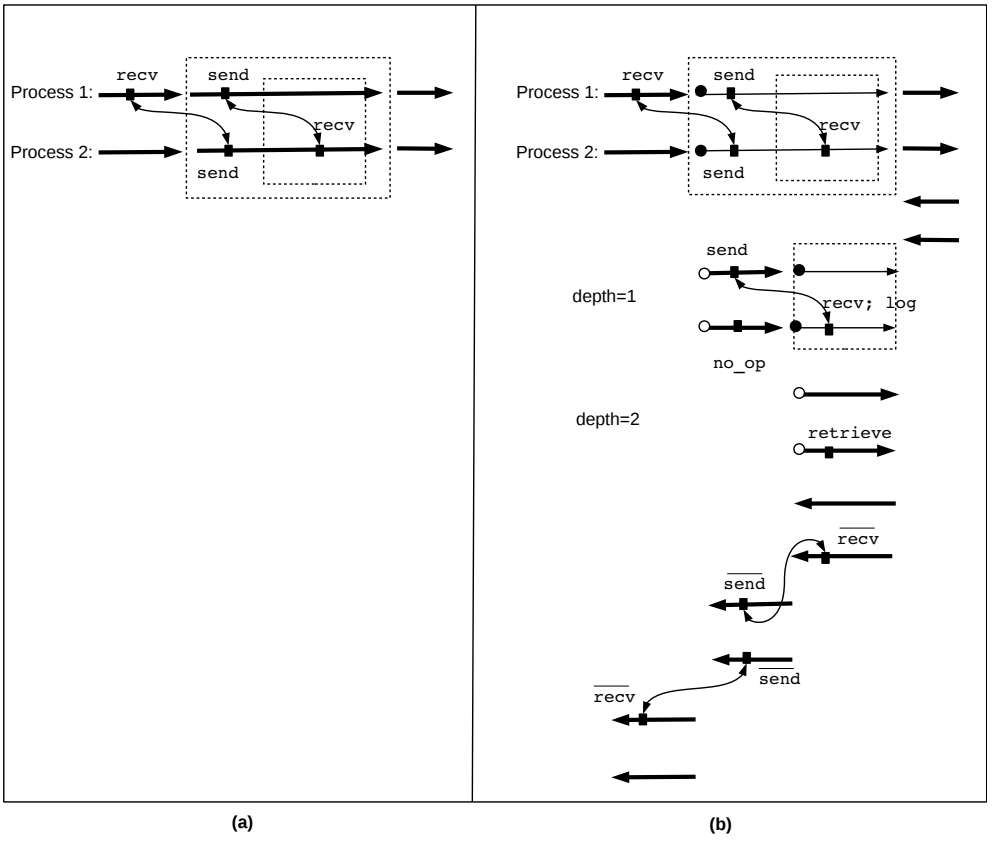

FIGURE 4.11: In (a), an MPI parallel program run on two processes. In (b), the adjoint corresponding after checkpointing two nested checkpointed parts, both of them right-tight. The receive-logging is applied to the orphan ends of communications and the message-resending is applied to the non-orphan ones

Figure 4.11 (a) shows an example, in which we selected two nested checkpointed parts. In figure 4.11 (a), we see that the `recv` of process 2 is `non-orphan` with respect to the

outer checkpointed part and `orphan` with respect to the inner one, i.e. its corresponding `send` belongs only to the outer checkpointed part. Since the outer checkpointed part is right-tight, we chose to apply message re-sending to the `recv` of process 2 together with its `send`. As result of checkpointing, see figure 4.11 (b), the `recv` of process 2 is activated when the depth of checkpointing is equal to 1. Since this `recv` will be de-activated during the depth just after, i.e. during depth=2, its received value has been logged during the current depth and retrieved during the depth just after.

4.5.1 Implementation Proposal

We propose an implementation of the combination method inside the AMPI library. This proposal allows for each end of communication to be activated during some depths of the checkpointed adjoint, i.e. we apply the message-resending to it, and de-activated during some others, i.e. we apply the receive-logging to it.

4.5.1.1 General view

The AMPI library is a library that wraps the calls to MPI subroutines in order to make the automatic generation of the adjoint possible in the case of MPI parallel programs. This library provides two types of wrappers:

- The "forward wrappers", called during the forward sweep of the adjoint code. Besides calling the MPI subroutines of the original MPI program, these wrappers store in memory the needed information to determine for every MPI subroutine, its corresponding adjoint, we call this "adjoint needed information". For instance, the forward wrapper that corresponds to a `wait`, `FWD_AMPI_wait` calls `wait` and stores in memory the type of non blocking routine with whom the `wait` is paired.
- The "backward wrappers" called during the backward sweep of the adjoint code. These wrappers retrieve the information stored in the forward wrappers and use it to determine the adjoint. For instance, the backward wrapper that corresponds to a `wait`, `BWD_AMPI_wait` calls `irecv` when the original `wait` is paired with an `isend`, i.e. we saw in subsection 4.1.1 that the adjoint for a `wait` depends on the non blocking routine with whom this `wait` is paired.

A possible implementation of the refined receive-logging techniques inside the AMPI library will either add new wrappers to this library, or change the existing forward wrappers so that they handle the combination method described at the beginning of section 4.5. We assume that the future implementation will rather change the existing

forward wrappers. In this case, these wrappers will be called more than once during the checkpointed adjoint, i.e. these wrappers will be called every time the checkpointed part is duplicated. An important question to be asked thus, when the adjoint needed information has to be saved? Is it better to save this information during the first execution of the checkpointed part or is it better to save this information each time the message-resending is applied, or is it better to save this information the last time the message-resending is applied?
Since this information is used only to determine the adjoint, we think that the third option is the best in terms of memory consumption. We notice, however, that if no message-resending is applied to the forward wrapper, then, we have to save this information during the first execution of the checkpointed part. Also, if the stack is the mechanism we use to save and retrieve the adjoint needed information, then, this information has to be retrieved and re-saved each time we do not apply the message-resending.

4.5.1.2 Interface proposal

It is quite difficult to detect statically if a checkpointed part is right-tight or if an MPI routine is orphan or not with respect to a given checkpointed part. This could be checked dynamically but it would require performing additional communications, i.e. each `send` has to tell its corresponding `recv` in which checkpointed part it belongs and vice versa. We believe that a possible implementation of receive-logging coupled with message-resending will require the help of the user to specify when applying the message-resending, for instance through an additional parameter to the `AMPI_send` and `AMPI_recv` subroutines. We call this parameter "`resending`". To deal with the case of nested structure of checkpointed parts, the `resending` parameter may for instance, specify for each depth of the nested structure, whether or not message-resending will be applied e.g. an array of booleans, in which the value 1 at index i reflects that message-resending will be applied at depth=i and the value 0 at index j reflects that message-resending will not be applied at depth=j, i.e. we will apply rather receive-logging.

From the other side, we may detect dynamically the depth of each end of communication belonging to a nested structure of checkpointed parts. The main idea is to:

- define a new global variable, that we call "`Depth`", and initiate it to zero at the beginning of the adjoint program.
- increment the variable `Depth`, before each forward sweep of a checkpointed part.
- decrement the variable `Depth`, after each backward sweep of a checkpointed part.

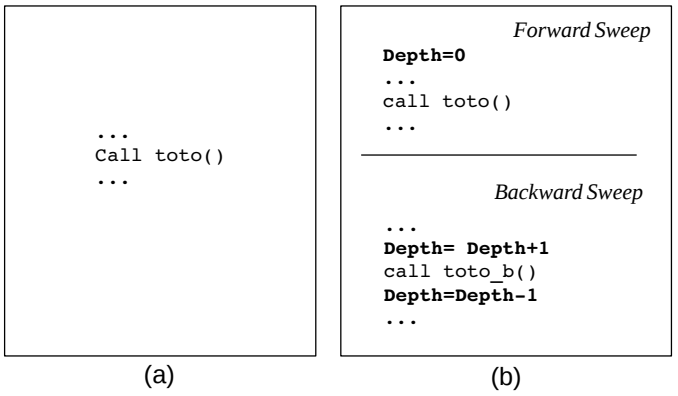

FIGURE 4.12: (a) a program that contains a call to a subroutine "`toto`". (b) the adjoint program after checkpointing the call to "`toto`". In the checkpointed adjoint, instructions have been placed to detect the depth of "toto" at run-time

At run time, the depth of an end of communication is the value of `Depth`. The instructions that allow initiating, incrementing and decrementing `Depth` may be easily placed by an AD tool inside the adjoint program. For instance, our AD tool Tapenade checkpoints every call to a subroutine. This means that if we have a call to a subroutine "`toto`" in the original code, we will have a call to "`toto`" in the forward sweep of the adjoint code and a call to "`toto_b`" in the the backward sweep of this code, in which "`toto_b`" contains the forward sweep and the backward sweep of the subroutine "`toto`", see figure 4.12. To detect the depth of each end of communication that belongs to "`toto`" at run time, it suffices to increment `Depth` before the call to "`toto_b`" and decrement `Depth` after the call to "`toto_b`", see figure 4.12.

Let us assume that `Depth` will be set as an AMPI global variable. i.e. `AMPI_Depth`. Figure 4.13 shows the various modifications we suggest for the wrappers `AMPI_FWD_send` and `AMPI_FWD_recv`. We see in figure 4.13 that we added `resending` as an additional parameter to our AMPI wrappers. For each end of communication, we check if the message-resending is applied at the current depth through a call to a function called "`isApplied`". This function takes `AMPI_depth` and `resending` as inputs and returns true if the message-resending is applied at `AMPI_Depth` and false in the opposite case. We check also if the message-resending will ever be applied in the following depths through a call to a function called "`willEverBeApplied`". This function takes `AMPI_Depth` and `resending` as inputs and returns true if the message-resending will ever be applied after `AMPI_Depth` and false in the opposite case. The algorithm sketched in figure 4.13 may be explained as:

- When message-resending is applied at a depth d, `sends` and their corresponding `recvs` are called. If message-resending is not applied at $d+1$, then we log in addition the received value. If message-resending will never be applied after d,

```
AMPI_FWD_recv(V,resending)
{
If (AMPI_Depth==0)||(isApplied(resending,AMPI_Depth)== true) then
   call MPI_recv(V)
   If (isApplied(resending, AMPI_Depth+1)==false) then
     log(V)
   endif
   If (willEverBeApplied(resending, AMPI_Depth)==false) then
     store the needed information for the adjoint
   Endif
Else
   retrieve(V)
   If (isApplied(resending, AMPI_Depth+1)==true) then
     free(V)
   endif
   restore the needed information for the adjoint
   store the needed information for the adjoint
}
```

```
AMPI_FWD_send(V,resending)
{
If (Depth==0)||(isApplied(resending,AMPI_Depth)== true) then
  call MPI_send(V)
  If (willEverBeApplied(resending, AMPI_Depth)==false) then
     store the needed information for the adjoint
  Endif
Else
   restore the needed information for the adjoint
   store the needed information for the adjoint
}
```

FIGURE 4.13: the modifications we suggest for some AMPI wrappers

then we have to save the adjoint needed information in both send and receive operations.

- When message-resending is not applied at a depth d, we retrieve the logged value in the receive side. If message-resending is applied at $d+1$, than, it is better in terms of memory to free the logged value. As we already mentioned, if the stack is the mechanism we use to save and retrieve the adjoint needed information, then this information has to be retrieved and re-saved in both send and receive operations.

We note that in our implementation proposal, if the user decides to apply the message-resending to one static MPI call, then this decision will be applied to all the run-time MPI calls that match this static call.

4.5.2 Further refinement: logging only the overwritten receives

We propose a further refinement to our receive-logging technique. This refinement consists in not logging every received value that is not used inside the checkpointed part, or, it is used but it is never modified since it has been received until the next use by the

duplicated instance of the checkpointed part, e.g. see figure 4.14. Formally, given `Recv` the set of variables that hold the received values inside the checkpointed part, `Use` the set of variables that are read inside the checkpointed part and `Out` the set of variables that are modified inside the checkpointed part (only the variables that are modified by more than one *receive* operation are included in the `Out` set of variables) and in the sequel of the checkpointed part, we will log in memory the values of variables `OverwrittenRecvs` with:

$$\texttt{OverwrittenRecvs} = \texttt{Recv} \cap \texttt{Use} \cap \texttt{Out}$$

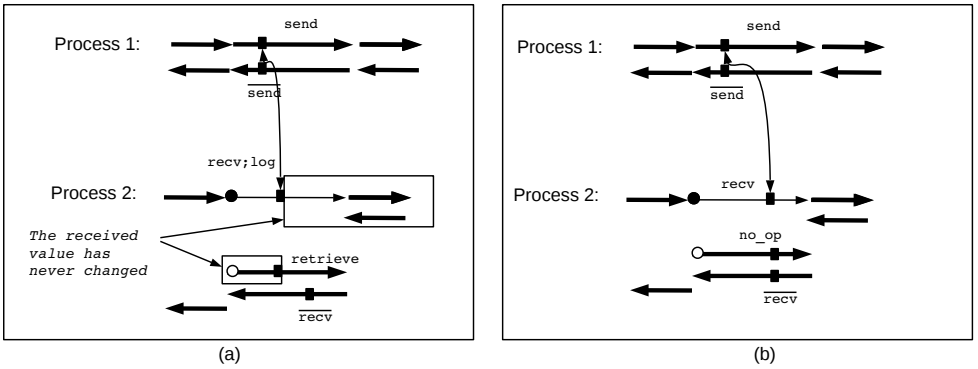

FIGURE 4.14: (a) An adjoint code after checkpointing a piece of code containing only the *receive* part of point-to-point communication. Checkpointing is applied together with the receive-logging technique, i.e. the *receive* call logs its received value during the first execution of the checkpointed part and retrieves it during the re-execution of the checkpointed part. In this example, the received value is never modified since it has been received until the next use by the duplicated instance of the checkpointed part, i.e. in the part of code surrounded by rectangles. (b) The same adjoint after refinement. In this code the received value is not saved anymore.

The values of `OverwrittenRecv` are called "`overwritten recvs`". Clearly, this is a small refinement as in the real codes, the number of `overwritten recvs` is much more important than the number of `non-overwritten` ones.

4.6 Choice of the combination

We saw in the previous subsections various methods to reduce the memory cost of the receive-logging technique. Some of them duplicate the call to MPI communications, which may add extra cost in terms of time execution and some of them propose not logging all the received values, but only those that are used and will be probably overwritten by the rest of the program. One important question to be asked, then, is for a given checkpointed piece, what is the best combination to be applied, i.e. what is the

combination that allows a reduction of the peak memory consumption without consuming too much in terms of time execution?
In the case where the checkpointed part is not right-tight, see subsection 4.4, we can only apply receive-logging to all the ends of communications inside this checkpointed part.
In the opposite case, i.e. the checkpointed part is right-tight:

- for all `orphan` ends of communications, we can only apply receive-logging.
- for the `non-orphan` ends of communications, we have the choice between applying the receive-logging and the message-resending techniques. When the `non-orphan` ends are `overwritten recvs`, then, it is more efficient in terms of memory to apply message-resending to these `overwritten recvs` together with their `sends`. Actually, applying receive-logging to these `recvs` will require extra storage. From the other hand, when the `non-orphan` ends are basically `non-overwritten recvs`, then, applying receive-logging to these `recvs` and theirs `sends` has the same cost in terms of memory as applying message-resending to these pairs `sends-recvs`. Thus, in this case we prefer applying receive-logging to these `recvs` and their `sends` as it requires less number of communications than in the case where message-resending is applied.

4.7 Choice of the checkpointed part

So far, we have discussed the strategies for communication calls, given the placement of checkpointed portions. We note that this placement is also some thing that can be chosen differently by the user, with the objective of improving the efficiency of the adjoint code. This section discusses this issue.

In real codes, the user may want to checkpoint some processes P independently from the others, either because checkpointing the other processes is not worth the effort, i.e. checkpointing the other processes does not reduce significantly the peak memory consumption, or checkpointing them will instead increase the peak memory consumption. In this case, is it more efficient in terms of memory to :

1. checkpoint only P, in which case we will have many `orphan` ends of communications which means applying receive-logging to the majority of MPI calls inside the checkpointed part,

2. or, checkpoint the set of processes P together with the other processes with whom P communicate, in which case we will apply message-resending to all the MPI calls inside the checkpointed part ?

3. or, do not checkpoint neither P nor the other processes with whom P communicate.

As the message-resending technique is in general memory efficient, one may prefer the option 2. However, in real codes, the option 2 may sometimes not be the best choice. Actually, choosing the best option depends on many factors such as: the fact that the checkpointed piece is right-tight or not, the cost of `overwritten recvs`, the cost of snapshot of other processes, etc..

We will study the memory consumption of various possible choices of checkpointing. We limit ourselves to the choice consisting to decide, for each process `i`, if the part of code studied `P` for this process will be checkpointed or not. It is therefore a Boolean function `C` of process number `i`. The memory consumption of a choice `C` results for the non-checkpointed processes in the trajectory storage $\texttt{Traj}_\texttt{i}$ performed during the execution of `P` by process `i` and for the checkpointed processes in the snapshot $\texttt{Snp}_\texttt{i}$ performed at the beginning of the execution of `P` by process `i`. In addition, for each *receive* end of communication that is overwritten, we will have to count the memory cost of a possible receive-logging applied to this end. In the following formulas, we will number each point-to-point communication by `j` from 1 to `m`. The cost of receive-logging will be the size of the received message $\texttt{size}_\texttt{j}$. In the case where the checkpointed part is right-tight, we prefer applying message-resending when it is allowed, i.e when the sending process $\texttt{s}_\texttt{j}$ and the receiving process $\texttt{r}_\texttt{j}$ are both checkpointed. In other cases, the message-resending can never be applied. Therefore, the memory consumption of a choice `C` is given by the following formulas:

When the checkpointed part C is right-tight:

$$\texttt{memo(C)} = \sum_{\texttt{i}=1}^{\texttt{n}} (\texttt{C(i)?Snp}_\texttt{i} : \texttt{Traj}_\texttt{i}) + \sum_{\texttt{j}=1}^{\texttt{m}} (\texttt{C}(\texttt{r}_\texttt{j})\&!\texttt{C}(\texttt{s}_\texttt{j})?\texttt{size}_\texttt{j} : 0)$$

When the checkpointed part C is not right-tight:

$$\texttt{memo(C)} = \sum_{\texttt{i}=1}^{\texttt{n}} (\texttt{C(i)?Snp}_\texttt{i} : \texttt{Traj}_\texttt{i}) + \sum_{\texttt{j}=1}^{\texttt{m}} (\texttt{C}(\texttt{r}_\texttt{j})?\texttt{size}_\texttt{j} : 0)$$

To sum up, the choice of the best checkpointed part in terms of memory boils down here to a comparison between the values of `memo(C)` at each choice of checkpointed part `C`.

4.8 Experiments

To validate our theoretical works, we selected two representative CFD codes in which we performed various choices of checkpointed parts. Both codes resolve the wave equation by using an iterative loop that at each iterations resolves:

$$U(x,t+dt) = 2U(x,t) - U(x,t-dt) + [c*dt/dx]^2 * [U(x-dx,t) - 2U(x,t) + U(x+dx,t)]$$

In which U models the displacement of the wave and c is a fixed constant. To apply checkpointing, we used the checkpointing directives of Tapenade, i.e. we placed \$AD CHECKPOINT-START and \$AD CHECKPOINT-END around each checkpointed part. By default, the checkpointed adjoint applies the message-resending technique, i.e. by default the resulting adjoint duplicates the calls to MPI communications. To apply the receive-logging, we de-activated by hand the duplication of MPI calls. In addition, for each `recv` call, we added the needed primitives that handle the storage of the received value during the first call of this `recv` and the recovery of this value when it is a duplicated instance of the `recv`.

4.8.1 First experiment

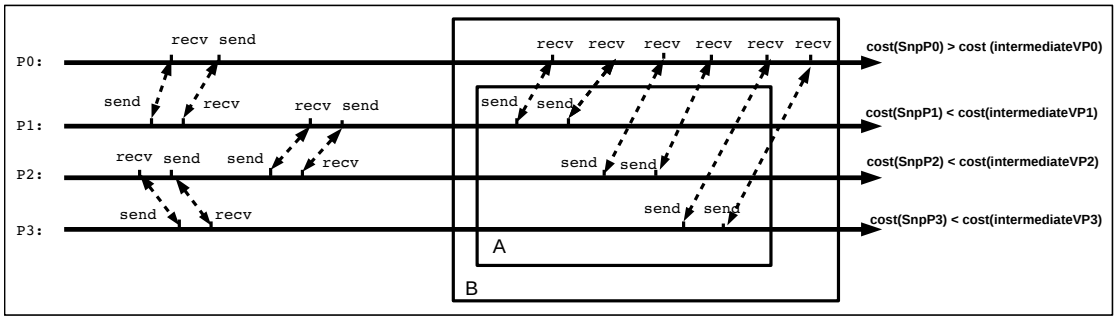

FIGURE 4.15: Representative code in which we selected two checkpointed parts

The first test is run on 4 processes. Figure 4.15 shows the various communications performed by these processes at each iteration of the global loop. We see in this figure, that at the end of each iteration, the process 0 collects the computed values from the other processes. In this code, we selected two alternative checkpointed parts: "A", in which we checkpoint the processes 1,2 and 3 and "B", in which we checkpoint all the processes. We see in figure 4.15, that checkpointing the process 0 increases the peak memory consumption of this process, i.e. the memory cost of snapshot of process 0, cost(SnpP0), is greater than the memory cost of logging its intermediate values, cost(intermediateVP0). We applied the receive logging to all MPI calls of the part of code "A" and the message-resending to all the MPI calls of the part "B".

The results of checkpointing "A" and "B" are shown in table 4.1. We see that the code resulting from checkpointing "A" is more efficient than the code resulting from checkpointing "B" not only in terms of number of communications, i.e. number of communications 48000 vs. 72000, but also in terms of memory consumption, i.e. total memory cost 36.2 Mbytes vs. 37.5 Mbytes. The efficiency in terms of number of communications was expected since receive-logging does not add extra communications to the adjoint code as it is the case of the message-resending. The efficiency in terms of memory can be explained by the fact that the checkpointed part "A" does not contain any `overwritten recvs`, i.e. it contains only `sends`, and thus does not require any extra storage. These results match the analysis of subsection 4.7.

	without CKP	CKP "B"	CKP "A"
Memory cost of P0 (MB)	8	9.3	8
Memory cost of P1,2,3 (MB)	12.6	9.4	9.4
Total Memory cost (MB)	45.8	37.5	36.2
Number of communications	48000	72000	48000

TABLE 4.1: Results of the first experiment.

4.8.2 Second experiment

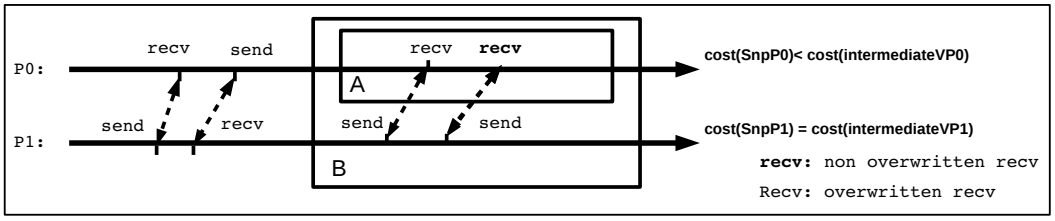

FIGURE 4.16: Representative code in which we selected two checkpointed parts

The second test is run on two processes. The communications performed by these two processes are shown in figure 4.16. In this test we study two alternative checkpointed parts as well. The first part "A" is run on only one process, i.e. process 0 and the second part "B" is run on the two processes.

The results of checkpointing "A" and "B" are shown in the table 4.2. Unlike the first experiment, checkpointing "B" here is more efficient in terms of memory, i.e. total memory cost 24.78 Mbytes vs. 24.82 Mbytes. This can be explained by two facts: the first one is that "A" contains `overwritten recvs` and the second one is that checkpointing process P1 does not decrease the memory consumption. These results also match the analysis of subsection 4.7. We notice here, that checkpointing "A" is always more efficient in

terms of number of communications than checkpointing "B". Clearly, the choice of the best checkpointed part depends here on the needs of the user.

	without CKP	CKP "B"	CKP "A"
Memory cost of P0 (MB)	15.58	12.36	12.39
Memory cost of P1 (MB)	12.45	12.42	12.43
Total Memory cost (MB)	28.03	24.78	24.82
Number of communications	16000	24000	16000

TABLE 4.2: Results of the second experiment

4.9 Discussion And Further Work

We considered the question of checkpointing in the case of MPI-parallel codes. Checkpointing is a memory/run-time trade-off which is essential for adjoint of large codes, in particular parallel codes. However, for MPI codes this question has always been addressed by ad-hoc hand manipulations of the differentiated code, and with no formal assurance of correctness. We investigated the assumptions implicitly made during past experiments, to clarify and generalize them. On one hand we proposed an extension of checkpointing in the case of MPI parallel programs with point-to-point communications, so that the semantics of an adjoint program is preserved for any choice of the checkpointed part. On the other hand, we proposed an alternative extension of checkpointing, more efficient but that requires a number of restrictions on the choice of the checkpointed part. We provided proof of correctness of these strategies, and in particular demonstrate that they cannot introduce deadlocks. We investigated a trade-off between the two extensions. We proposed an implementation of these strategies inside the AMPI library. We discussed practical questions about the choice of strategy to be applied within a checkpointed part and the choice of the checkpointed part itself. At the end, we validated our theoretical results on representative CFD codes.
There are a number of questions that should be studied further.

In this work, we have been driven to extend the notion of checkpoint to parallel codes with multiple processes. In other words, checkpointed parts that are generally thought of as subsequences of some execution trace, must acquire an extra dimension that represents processes. The extension we have come up with has been helpful for our work, but we are still not sure it is the most appropriate representation. Should we think of separate checkpointed parts inside each process, or should we rather build composite checkpoints that cover multiple processes? To answer this question, it is necessary to clarify the link between static and dynamic checkpoints. We like to think of checkpointed parts

as dynamic, as we represent them as duplicate executions of a part of some run-time process of execution. On the other hand, checkpointed parts are defined on the source code, from one location in the source to another. This contradiction should be clarified to facilitate the study of checkpointing in an MPI setting.

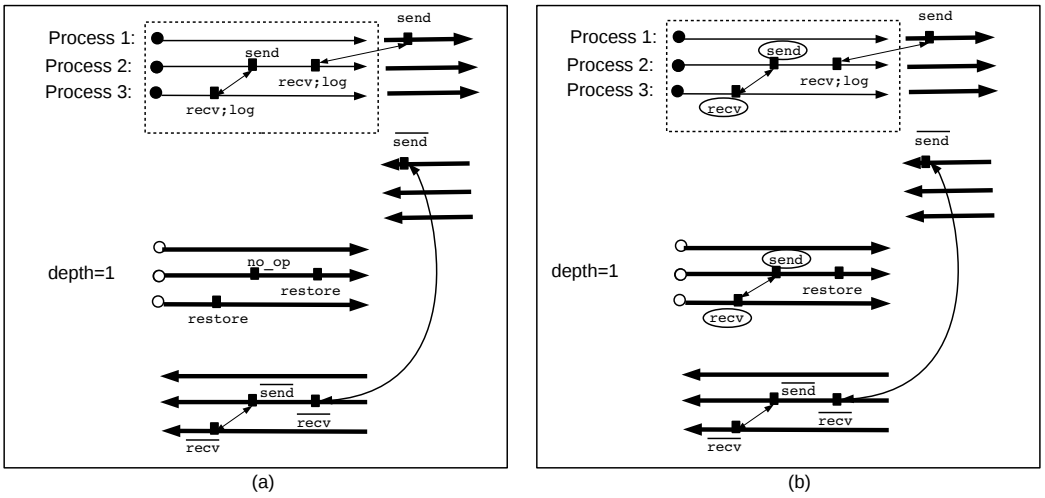

FIGURE 4.17: (a) The receive-logging applied to a parallel adjoint program. (b) Application of the message re-sending to a `send-recv` pair with respect to a non-right-tight checkpointed part of code

We imposed a number of restrictions on the checkpointed part in order to apply the refinement. These are sufficient conditions, but it seems they are not completely necessary. Figure 4.17 shows a checkpointed part of code which is not right-tight. Still, the application of the message re-sending to a `send-recv` pair (whose ends are surrounded by circles) in this checkpointed part, does not introduce deadlocks in the resulting checkpointed adjoint.

The implementation proposal we suggest in section 4.5.1 allows an application of receive-logging coupled with message-resending that may be considered as "semi-automatic". Actually, this proposal requires the help of user to specify for each end of communication, the set of depths in which it will be activated, i.e. in which depths message-resending will be applied to this end. An interesting further research is how to automatically detect this information for instance by detecting if a checkpointed part is right-tight and also if an end of communication is orphan or not with respect to a given checkpointed part.

In the Recompute-All approach, the presence of MPI communications restricts also the choice of parts of code to be recomputed, i.e. these parts have to contain both ends of every point-to-point communication. Receive-logging coupled with message-resending might be a good approach to be applied in this case.

In this work, we studied chekpointing in the case of MPI parallel programs with point-to-point communications. Studying this question in the case of collective communications might be interesting further work.

Finally, we experimented the "receive-logging" and "message-resending" techniques on representative home-made codes. It might be useful to experiment these techniques on real size codes.

Chapter 5

Conclusion (français)

Ce travail sur le mode adjoint de la Différentiation Algorithmique (DA) a mis l'accent sur deux problèmes d'une importance particulière, notamment pour les applications industrielles, dans lesquelles la taille des codes est grande et le temps d'exécution et l'efficacité mémoire sont cruciaux. Le projet européen AboutFlow qui a soutenu cette recherche a fourni la motivation pour ces deux problèmes, ainsi que les codes d'application. Ces deux problèmes sont d'une part sélectionner et implémenter un algorithme adjoint adapté pour les algorithmes point fixe et d'autre part étudier les limitations imposées par l'architecture parallèle MPI sur le mécanisme de checkpointing.

Bien que ces deux questions sont a priori distinctes, elles partagent leur contexte: la Différentiation Algorithmique adjointe et le problème de l'inversion du flux de données, générer un code adjoint qui soit efficace à la fois en termes de temps et en termes de mémoire via checkpointing et la nécessité de détecter et de profiter des structures particulières présentes dans le code différentié.

Une question qui se pose de façon similaire est l'étude des restrictions d'applicabilité. Une spécification précise de ces restrictions reste à trouver. Des parties importantes de ces restrictions pourraient être levées par des travaux ultérieurs. Par exemple, la stratégie Deux-Phases raffinée pour les boucles à point fixe peut probablement être étendue à des boucles avec plusieurs entrées ou sorties (voir la section 3.4). De même, il y a des situations où une partie d'une exécution MPI n'est pas "étanche à droite" et nous pouvons quand même lui appliquer le "message-resending" (voir la section 4.4). Indépendamment de la question particulière abordée, ces restrictions d'applicabilité exigent des outils d'aide. L'un est la possibilité de transformer le code original afin qu'il répond aux restrictions (par exemple peler la boucle Point Fixe). Faut-il que l'utilisateur final soit seul responsable de ces transformations? Nous pensons qu'un outil

de DA par transformation de source est approprié pour effectuer une telle transformation. Néanmoins, pour chaque transformation intrusive telle que le déroulement de boucle, ceci doit être contrôlé par l'utilisateur final par des directives.

La principale question sur les restrictions d'applicabilité est comment vérifier si un code donné les satisfait. Chaque fois que c'est possible, une vérification statique sur le source du code est préférable. Nous avons vu cependant que la détection statique peut être très imprécise, conduisant l'outil à rejeter des codes parfaitement acceptables. Par exemple, la détection statique que la partie checkpointée d'un code MPI est étanche à droite échouera sur la plupart des codes de grande taille. La réponse classique est à nouveau les directives de l'utilisateur final. Nous croyons qu'il serait profitable de développer un contrôle dynamique a posteriori des restrictions d'applicabilité. Par exemple, dans les boucles à point fixe, nous avons besoin d'une vérification dynamique de la stationnarité du flux de contrôle. Pour la question du checkpointing des codes MPI, la détection des paires `MPI_send/ MPI_recv` qui se correspondent ne peut être effectuée en général que dynamiquement. Nous pensons que ceci est une direction de recherche intéressante.

Ce travail a été mené dans le contexte des outils de DA par transformation de source, créant un code adjoint qui de base sur l'approche Store-All pour pouvoir inverser le flux de données. À notre connaissance, seul l'outil de DA TAF [16] utilise une stratégie basée sur l'approche Recompute-All. Nous sommes conscients que TAF implémente des stratégies adaptées à des questions proches de celles que nous avons étudiées. Outre le fait que les stratégies de TAF sont insuffisamment documentées (TAF étant un outil propriétaire), il nous semble que nos techniques sont légèrement plus développées et pourraient inspirer quelques améliorations à TAF.

Étendre plus loin ce travail à la DA basée sur la surcharge des opérateurs [40], nous pensons que la complexité du checkpointing dans ce contexte rend irréaliste l'application de nos propositions sur les codes MPI. En revanche, l'adjoint des boucles à point fixe Deux-Phases raffinée semble prometteuse, surtout que la DA avec surcharge des opérateurs est connue par sa grande consommation mémoire et que le principal atout de la méthode Deux-Phases raffinée est précisément sa faible consommation en mémoire.

Chapter 6

Conclusion (english)

This work on adjoint Algorithmic Differentiation has focused on two problems of particular importance, especially for industrial applications, in which code sizes are huge and run-time and memory efficiency are crucial. The AboutFlow European project that boosted this research provided the motivation for these two problems, as well as the application codes. These two issues are to select and implement an adapted adjoint algorithm for Fixed-Point iterations on the one hand, and on the other hand to study limitations imposed by MPI parallel architecture on the adjoint trade-off mechanism known as checkpointing.

Although the link between theses two questions is not obvious at first sight, they share their context of adjoint AD and the problem of data-flow reversal, the quest for time and memory efficiency through checkpointing, and the need to detect and to take advantage of the particular code organization.

An issue that come up in a similar manner for both questions is the applicability restrictions. An accurate specification of these restrictions is still to be found. Still, significant parts of these restrictions might be lifted by further work. For instance the Two-Phases strategy for Fixed-Point loops can certainly be extended to loops with multiple entries or exits (see section 3.4). Similarly, there are situations where a checkpointed part of an MPI execution is not "right-tight" and we can still apply message-resending to it (see section 4.4). Independently of the particular question addressed, these applicability restrictions call for helping tools. One is the possibility to transform the original code so that it meets the restrictions (think of loop peeling). Should the end-user alone be in charge of these transformations? We believe a source-transformation AD tool is an appropriate framework to perform such transformation. Still, for every intrusive transformation such as loop unrolling, this must be controlled by the end-user through directives.

The main issue about applicability restriction is how to check for them. Whenever possible, a static checking, on the code source, is preferable. We saw however that static detection may be highly inaccurate, leading the tool to reject perfectly acceptable codes. For instance, detecting statically that an MPI code's checkpointed part is right-right will fail on most large codes. The classic answer is again end-user directives. We believe it would be profitable to develop dynamic checking of applicability restrictions. For instance, in FP loops we need a dynamic verification for stationnarity of the flow of control. In MPI checkpointing, only dynamic verification can find matching send/receive pairs. We believe this is a useful research direction.

This strategy was conducted in the context of Source-Transformation AD tools, building adjoint code with store-all data-flow reversal. To our knowledge, only TAF [16] uses a recompute-all reversal strategy. We are aware that TAF implements adapted strategies for questions close to the ones we studied. Still, we observed that these strategies in TAF are slightly less developed than ours, at least for the available documentation, and so could be improved. Extending further to overloading-based AD, we believe that the complexity of checkpointing in this context makes it unrealistic to apply our proposals about MPI codes. However, Fixed-Point Two-Phases adjoint seems a promising approach, particularly since overloading AD is known to use a lot of memory and the main strength of the Two-Phases method is its low memory consumption.

Bibliography

[1] Adol-C, 2016.

[2] OpenAD, 2016.

[3] M. Araya-Polo and L. Hascoët. Data flow algorithms in the tapenade tool for automatic differentiation. In *Proceedings of 4^{th} European Congress on Computational Methods, ECCOMAS'2004, Jyvaskyla, Finland*, 2004.

[4] T. Bosse. Augmenting the one-shot framework by additional constraints. *Optimization Methods and Software*, 31(6), 2016.

[5] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra. Reasons for a pessimistic or optimistic message logging protocol in MPI uncoordinated failure, recovery. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA*, pages 1–9, 2009.

[6] F. Capello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: A 2014 update. 2014.

[7] A. Carle and M. Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Department of Computational and Applied Mathematics, Rice University, 2000.

[8] A. Carle and M. Fagan. Automatically differentiating MPI-1 datatypes: The complete story. In George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 25, pages 215–222. Springer, New York, NY, 2002.

[9] F. Christakopoulos. *Sensitivity computation and shape optimisation in aerodynamics using the adjoint methodology and Automatic Differentiation.* PhD thesis, Queen Mary University of London, 2013.

[10] B. Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3:311–326, 1994.

[11] B. Christianson. Reverse accumulation and implicit functions. *Optimization Methods and Software*, 9(4):307–322, 1998.

[12] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, June 1996.

[13] Z. Dastouri, S. M. Gezgin, and U. Naumann. A mixed operator overloading and source transformation approach for adjoint cfd computation. In *Proceedings of European Congress on Computational Methods, ECCOMAS Congress 2016, Greece*, 2016.

[14] B. Dauvergne and L. Hascoët. The data-flow equations of checkpointing in reverse automatic differentiation. In *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part IV*, pages 566–573, 2006.

[15] About Flow, 2016.

[16] R. Giering. *Tangent Linear and Adjoint Model Compiler, Users Manual.* Center for Global Change Sciences, Department of Earth, Atmospheric, and Planetary Science, MIT, Cambridge, MA, December 1997. Unpublished.

[17] R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software*, 24(4):437–474, 1998.

[18] R. Giering and T. Kaminski. Towards an optimal trade off between recalculations and taping in reverse mode ad. In *Automatic Differentiation of Algorithms: From Simulation to Optimization.* 2001.

[19] R. Giering, T. Kaminski, and T. Slawig. Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil. *Future Generation Computer Systems*, 21(8):1345–1355, 2005.

[20] J. C. Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1(1):13–21, 1992.

[21] D. N. Goldberg, S. H. K. Narayanan, L. Hascoët, and J. Utke. An optimized treatment for algorithmic differentiation of an important glaciological fixed-point problem. *Geosci. Model Dev.*, 9:1891–1904, 2016.

[22] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992.

[23] A. Griewank and C. Faure. Reduced functions, gradients and hessians from fixed-point iterations for state equations. *Numerical Algorithms*, 30:113–139, 2002.

[24] A. Griewank and C. Faure. Piggyback differentiation and optimization. In Biegler et al., editor, *Large-scale PDE-constrained optimization*, pages 148–164. Springer, LNCSE #30, 2003.

[25] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Other Titles in Applied Mathematics, #105. SIAM, 2008.

[26] L. Hascoët. *Analyses statiques et transformations de programmes: de la parallélisation a la différentiation*. Habilitation, Université de Nice Sophia-Antipolis, 2005.

[27] L. Hascoët and M. Araya-Polo. The adjoint data-flow analyses: Formalization, properties, and applications. In *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Computational Science and Engineering. Springer, 2005. Selected papers from AD2004 Chicago, July 2005.

[28] L. Hascoët and M. Araya-polo. Enabling user-driven checkpointing strategies in reverse-mode automatic differentiation, 2006.

[29] L. Hascoët, S. Fidanova, and C. Held. Adjoining independent computations. In *Automatic Differentiation of Algorithms, from Simulation to Optimization*, Computer and Information Science, pages 299–304. Springer, 2001. selected papers from the AD2000 conference, Nice, France.

[30] L. Hascoët, U. Naumann, and V. Pascual. "to be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Comp. Syst.*, 21(8):1401–1417, 2005.

[31] L. Hascoët and V. Pascual. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software*, 39(3), 2013.

[32] P. Heimbach, C. Hill, and R. Giering. An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation. *Future Generation Comp. Syst.*, 21(8):1356–1371, 2005.

[33] P. D. Hovland. *Automatic Differentiation of Parallel Programs*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, May 1997.

[34] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. Number 16 in Frontiers in Applied Mathematics. SIAM, 1995.

[35] J. G. Kim and P. D. Hovland. Sensitivity analysis and parameter tuning of a sea-ice model. In George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and

U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 9, pages 91–98. Springer, New York, NY, 2002.

[36] A. Kowarz and A. Walther. Optimal checkpointing for time-stepping procedures in ADOL-C. In V. N. Alexandrov, G. D. Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 541–549, Heidelberg, 2006. Springer.

[37] B. Mohammadi, J.M. Malé, and N. Rostaing-Schmidt. Automatic differentiation in direct and reverse modes: Application to optimum shapes design in fluid mechanics. In Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 309–318. SIAM, Philadelphia, PA, 1996.

[38] U. Naumann. *Call Tree Reversal is NP-Complete*, pages 13–22. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[39] U. Naumann. Dag reversal is np-complete. *J. of Discrete Algorithms*, 7(4):402–410, December 2009.

[40] U. Naumann. *The Art of Differentiating Computer Programs - An Introduction to Algorithmic Differentiation*, volume 24 of *Software, environments, tools*. SIAM, 2012.

[41] U. Naumann, L. Hascoët, C. Hill, P. D. Hovland, J. Riehme, and J. Utke. A framework for proving correctness of adjoint message-passing programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting, Dublin, Ireland, September 7-10, 2008. Proceedings*, pages 316–321, 2008.

[42] U. Naumann, J. Utke, A. Lyons, and M. Fagan. Control flow reversal for adjoint code generation. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 55–64, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[43] Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara. Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations. *ACM Trans. Math. Softw.*, 41(4):26:1–26:21, October 2015.

[44] V. Pascual and L. Hascoët. Native handling of message-passing communication in data-flow analysis. In *Recent Advances in Algorithmic Differentiation*, volume 87

of *Lecture Notes in Computational Science and Engineering*, pages 83–92. Springer, Berlin, 2012.

[45] V. Pascual and L. Hascoët. Mixed-language automatic differentiation. In *AD2016-Programme and Abstracts, Oxford, UK*, 2016.

[46] J. M. Restrepo, G. K. Leaf, and A. Griewank. Circumventing storage limitations in variational data assimilation studies. *SIAM J. Scientific Computing*, 19(5):1586–1605, 1998.

[47] J. Reuther. Aerodynamic shape optimization of supersonic aircraft configurations via anadjoint formulation on distributed memory parallel computers. *Computers and Fluids*, 28(4–5):675–700, 1999.

[48] M. Schanen. *Semantics Driven Adjoints of the Message Passing Interface.* PhD thesis, RWTH Aachen University, October 2014.

[49] M. Schanen, U. Naumann, L. Hascoët, and J. Utke. Interpretative adjoints for numerical simulation codes using MPI. In *Proceedings of the International Conference on Computational Science, ICCS 2010, University of Amsterdam, The Netherlands, May 31 - June 2, 2010*, number 1, pages 1825–1833, 2010.

[50] S. Schlenkrich, A. Walther, N. R. Gauger, and R. Heinrich. Differentiating fixed point iterations with ADOL-C: gradient calculation for fluid dynamics. In *Modeling, Simulation and Optimization of Complex Processes, Proceedings of the Third International Conference on High Performance Scientific Computing, March 6-10, 2006, Hanoi, Vietnam*, pages 499–508, 2006.

[51] D. A. Schmidt. Programming language semantics. In *Computing Handbook, Third Edition: Computer Science and Software Engineering*, pages 69: 1–19. 2014.

[52] M. Snir, S. Otto, and S. Huss-Lederman. *MPI : the complete reference. Volume 1. , The MPI core.* Scientific and engineering computation. Cambridge, Mass. MIT Press, 1998.

[53] M. Towara, M. Schanen, and U. Naumann. Mpi-parallel discrete adjoint openfoam. In *Proceedings of the International Conference on Computational Science, ICCS 2015, Computational Science at the Gates of Nature, Reykjavík, Iceland, 1-3 June, 2015, 2014*, pages 19–28, 2015.

[54] J. Utke, L. Hascoët, P. Heimbach, C. Hill, P. D. Hovland, and U. Naumann. Toward adjoinable MPI. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–8, 2009.

[55] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.