CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS

École Doctorale EDITE

ÉQUIPES de RECHERCHE : MMSA au LOCEAN et MSDMA au CEDRIC

THÈSE présentée par :

Luigi NARDI

soutenue le : 8 mars 2011

pour obtenir le grade de : Docteur du Conservatoire National des Arts et Métiers

Spécialité : informatique

Formalisation et automatisation de YAO, générateur de code pour l'assimilation variationnelle de données

THÈSE DIRIGÉE PAR :

BADRAN Fouad	Professeur des universités, CNAM
THIRIA Sylvie	Professeur des universités, Versailles Saint Quentin en Yvelines

RAPPORTEURS :

CHEVALLIER Frédéric	Chercheur, Laboratoire des Sciences du Climat et de l'Environnement
SIARRY Patrick	Professeur des Universités, Paris-Est Créteil

JURY :

PICOULEAU Christophe	Professeur des universités, CNAM Président du jury
BASTOUL Cédric	Maître de conférence, Université Paris-Sud
HERLIN Isabelle	Directeur de recherche, Institut National de Recherche en Informatique et en Automatique
MOULIN Cyril	Directeur Adjoint, Laboratoire des Sciences du Climat et de l'Environnement

Remerciements

Pour exprimer ma gratitude envers ceux qui m'ont aidé et encouragé durant ces quelques années, il m'aurait fallu plus que ces quelques lignes. J'essayerai de n'oublier personne, mais la tâche est rude !

Je commence bien évidemment par Fouad Badran, pour son encadrement et son soutien quotidien durant ces trois dernières années. Il a su me faire profiter de son expérience scientifique avec bienveillance. Je le remercie de sa confiance et de tout ce qu'il m'a transmis. Je tiens à exprimer ma profonde gratitude à Sylvie Thiria d'avoir co-encadré cette thèse. Ses conseils et son énergie ont toujours su rassurer mon chemin de recherche. Je salue la souplesse et l'ouverture d'esprit de mes directeurs de thèse qui ont su me laisser une large marge de liberté pour mener à bien ce travail.

Je remercie Frédérick Chévallier et Patrick Siarry d'avoir accepté d'être les rapporteurs de ma thèse. Je remercie également le président Christophe Picouleau et les membres du jury Cédric Bastoul, Isabelle Herlin, Cyril Moulin pour m'avoir fait l'honneur d'examiner mes travaux. Cette thèse pluridisciplinaire a pu bénéficier de leur évaluation minutieuse grâce à leurs compétences variées et complémentaires.

Au cours de la thèse, dans ce cadre si particulier qu'était le LOCEAN et le campus de Jussieu, j'ai été marqué par de nombreuses rencontres qui se sont parfois transformées en collaborations. Aux premiers rangs, je tiens à saluer et remercier Julien Brajard et Pierre Fortin, qui ont fait office de grands frères dans mes explorations en assimilation de données et en calcul haute performance. Leur soutien a été déterminant à certains moments et je leur dois énormément. Je suis heureux d'avoir pu croiser leur chemin.

Ensuite, et je ne l'écartais que parce qu'il « habitait » deux laboratoires plus loin, je tiens à remercier Joseph Salmon. Ce grand chercheur/ami qui est toujours prêt a donner un bon conseil a qui a perdu son chemin. Les maths prennent du charme quand elles sortent de sa bouche. Ma découverte du monde de la recherche aura été un vrai plaisir à ses côtés, par sa joie de vivre et sa curiosité intellectuelle.

Je voudrais remercier Carlos Mejia de me faire l'honneur d'être un cher ami et collègue de bureau. J'ai découvert énormément grâce à ses remarques. Les problèmes s'en fuient quand Carlos arrive. Je le remercie de sa bienveillance à mon égard.

Je tiens à remercier mes collaborateurs du LIP6. Merci à Fabienne Jézéquel, Stef Graillat, Dominique Béréziat et Jéan-Luc Lamotte, j'admire votre vision de la science et je vous remercie

d'avoir ouvert ces magnifiques perspectives à mes recherches.

Un grand merci à mes collègues de bureau Houda, Manel et Anastase, pour avoir rendu agréables les moments pas toujours faciles passés au laboratoire. Cette ambiance que vous avez la capacité de créer et dans laquelle vous enveloppez tout au tour de vous fait envoler les esprits et rend paisible le travail en lui donnant de la légèreté.

Un énorme merci au comité relecture de m'avoir aidé à rendre ce manuscrit le plus lisible possible. Sabine, Anne Charlotte, Guillaume, Joseph, Houda, Alice, Francesco, je ne sais vraiment pas comment j'avais fait si vous n'étiez pas là, toujours prêt à corriger les fautes et à rectifier la forme.

Un merci à Michel Crépon d'avoir été toujours prêt à donner un bon conseil. Son avis rassure les esprits et ses remarques donnent de la rigueur aux publications. Son aide a était très importante, merci encore.

Un grand merci aussi à Mohamed Berrada, Abdou Kane, Hector Simon Benavides et Awa Niang pour avoir permis, avec leurs efforts, l'avancement du projet YAO. Vous êtes des collaborateurs infatigables et je suis heureux d'avoir pu travailler avec vous.

Enfin un grand merci aux collaborateurs juniors, les stagiaires Abdouramane, Alexis, Christophe et Guillaume qui ont soutenu activement l'évolution de YAO en donnant une contribution fondamentale à cet enthousiasmant projet.

Une pensée va aussi à notre magnifique équipe réseau qui a su supporter mes caprices et mes exigences tout au long de la thèse et spécialement lors des simulations sur les serveurs de calcul. Pierre, Julien et Paul votre travail est l'épine dorsale des recherches faites au laboratoire, j'espère que vous continuerez à assurer cette tâche avec la même intensité et énergie.

Je remercie aussi l'ensemble de l'équipe administrative du laboratoire, plus particulièrement Florence et Nelly, toujours à l'écoute et prêtes à nous aider avec efficacité. Un grand merci aussi à notre directrice Laurence pour son dévouement à la science et son regard attentif au bien être des doctorants.

Je salue les thésards (et les autres aussi !) du LOCEAN et en particulier Agathe, Manu, Elena, Adrien, Maïté, Fernanda, Benoît, Leticia, Dorotea, Simona, Yves, Philippe, Laetitia, Pascaline, Antoine, pour les discussions échangées, le soutien et l'encouragement. La petite pause goûter avec vous restera un incontournable dans mes souvenirs, cette richesse d'échanges autour d'un simple thé. Merci à vous tous pour avoir égayer nos repas pendant si longtemps.

Une pensée va aussi à Christine, Natalie et Alice pour leur confiance durant l'expédition en antarctique et leur enthousiasme. Cette expérience scientifique m'a beaucoup marqué et reste très vive dans mes souvenirs.

Enfin, je tiens à remercier ma famille, qui a toujours su m'apporter un soutien sans faille. Merci pour la logistique, qui comme chacun sait est le nerf de la guerre.

Dulcis in fundo, tout cela n'aurait pas pu être possible sans le soutien moral de ma chère Alice. C'est à elle que je dédie ce travail, pour sa patience et sa lumière.

REMERCIEMENTS

REMERCIEMENTS

Résumé

L'assimilation variationnelle de données 4D-Var est une technique très utilisée en géophysique, notamment en météorologie et océanographie. Elle consiste à estimer des paramètres d'un modèle numérique direct, en minimisant une fonction de coût mesurant l'écart entre les sorties du modèle et les mesures observées. La minimisation, qui est basée sur une méthode de gradient, nécessite le calcul du modèle adjoint (produit de la transposée de la matrice jacobienne avec le vecteur dérivé de la fonction de coût aux points d'observation). Lors de la mise en œuvre de l'AD 4D-Var, il faut faire face à des problèmes d'implémentation informatique complexes, notamment concernant le modèle adjoint, la parallélisation du code et la gestion efficace de la mémoire.

Afin d'aider au développement d'applications d'AD 4D-Var, le logiciel YAO qui a été développé au LOCEAN, propose de modéliser le modèle direct sous la forme d'un graphe de flot de calcul appelé graphe modulaire. Les modules représentent des unités de calcul et les arcs décrivent les transferts des données entre ces modules. YAO est doté de directives de description qui permettent à un utilisateur de décrire son modèle direct, ce qui lui permet de générer ensuite le graphe modulaire associé à ce modèle. Deux algorithmes, le premier de type propagation sur le graphe et le second de type rétropropagation sur le graphe permettent, respectivement, de calculer les sorties du modèle direct ainsi que celles de son modèle adjoint. YAO génère alors le code du modèle direct et de son adjoint. En plus, il permet d'implémenter divers scénarios pour la mise en œuvre de sessions d'assimilation.

Au cours de cette thèse, un travail de recherche en informatique a été entrepris dans le cadre du logiciel YAO. Nous avons d'abord formalisé d'une manière plus générale les spécifications de YAO. Par la suite, des algorithmes permettant l'automatisation de certaines tâches importantes ont été proposés tels que la génération automatique d'un parcours "optimal" de l'ordre des calculs et la parallélisation automatique en mémoire partagée du code généré en utilisant des directives OpenMP. L'objectif à moyen terme, des résultats de cette thèse, est d'établir les bases permettant de faire évoluer YAO vers une plateforme générale et opérationnelle pour l'assimilation de données 4D-Var, capable de traiter des applications réelles et de grandes tailles.

Mots clés : assimilation variationnelle de données, modèle numérique, modèle adjoint, génération automatique, parallélisation automatique, mémoire partagée, OpenMP.

RÉSUMÉ

Abstract

Variational data assimilation 4D-Var is a well-known technique used in geophysics, and in particular in meteorology and oceanography. This technique consists in estimating the control parameters of a direct numerical model, by minimizing a cost function which measures the misfit between the forecast values and some actual observations. The minimization, which is based on a gradient method, requires the computation of the adjoint model (product of the transpose Jacobian matrix and the derivative vector of the cost function at the observation points). In order to perform the 4D-Var technique, we have to cope with complex program implementations, in particular concerning the adjoint model, the parallelization of the code and an efficient memory management.

To address these difficulties and to facilitate the implementation of 4D-Var applications, LOCEAN is developing the YAO framework. YAO proposes to represent a direct model with a computation flow graph called modular graph. Modules depict computation units and edges between modules represent data transfer. Description directives proper to YAO allow a user to describe its direct model and to generate the modular graph associated to this model. YAO contains two core algorithms. The first one is a forward propagation algorithm on the graph that computes the output of the numerical model ; the second one is a back propagation algorithm on the graph that computates the adjoint model. The main advantage of the YAO framework, is that the direct and adjoint model programming codes are automatically generated once the modular graph has been conceived by the user. Moreover, YAO allows to cope with many scenarios for running different data assimilation sessions.

This thesis introduces a computer science research on the YAO framework. In a first step, we have formalized in a more general way the existing YAO specifications. Then algorithms allowing the automatization of some tasks have been proposed such as the automatic generation of an “optimal” computational ordering and the automatic parallelization of the generated code on shared memory architectures using OpenMP directives. This thesis permits to lay the foundations which, at medium term, will make of YAO a general and operational platform for data assimilation 4D-Var, allowing to process applications of high dimensions.

Keywords : variational data assimilation, numerical model, adjoint model, automatic generation, automatic parallelization, shared memory, OpenMP.

ABSTRACT

Table des matières

Chapitre 1

Introduction

La modélisation numérique est de plus en plus utilisée comme complément à l'étude d'un phénomène physique se déroulant dans le temps, dans l'espace à une, deux ou trois dimensions (1D, 2D, 3D) ou bien tout à la fois dans le temps et l'espace. Dans une première étape le modèle numérique permet de choisir et d'adapter les équations permettant de reproduire les lois physiques qui sous-tendent le phénomène. Une fois vérifié le bon comportement de ces lois, il est alors utilisé pour l'étude et la simulation du phénomène lui-même. Cependant, les modèles numériques (modèles directs) ainsi créés restent toujours imparfaits. La paramétrisation des modèles, les discrétisations, les incertitudes sur les conditions initiales et les conditions aux limites introduisent des approximations dont il faudra tenir compte au moment d'étudier le comportement du phénomène ou de prédire ses évolutions. De nouvelles techniques, qui utilisent à la fois la modélisation et la résolution du problème inverse, ont vu le jour, elles se regroupent sous le nom d'*assimilation de données* (AD). Celles-ci tentent de faire évoluer les paramètres du modèle numérique ou ceux qui le contraignent (conditions initiales, ...) en introduisant un nouveau type d'information : les observations. Elles utilisent alors le modèle numérique du phénomène, ou modèle direct, et se servent des observations pour contrôler les modifications nécessaires. Il existe deux grandes familles de techniques pour faire de l'assimilation de données : les méthodes séquentielles et les méthodes variationnelles [Bouttier et Courtier, 1997, Le Dimet et Talagrand, 1986, Louvel, 1999, Sportisse et Quélo, 2004, Talagrand, 1991, Talagrand, 1997], la seconde étant plus adaptée, si l'on veut prendre en compte d'une manière globale les variations du phénomène pendant une période de temps donnée. Toutes sont aujourd'hui de plus en plus utilisées, car elles permettent de faire interagir l'ensemble des informations disponibles (connaissances théoriques sur le phénomène et observations de celui-ci) dans l'espace et dans le temps.

L'efficacité de ces méthodes est visible si l'on considère leurs bonnes performances sur des problèmes réels. On peut faire référence, parmi d'autres, à l'amélioration actuelle obtenue sur les prévisions des modèles météorologiques qui assimilent les observations satellitaires et in situ depuis une vingtaine d'années. La technique choisie par les centres de prévision météorologique est, en général, l'assimilation variationnelle dite *4D-Var*, qui permet la prise en compte des observations mesurées en divers points de l'espace et à n'importe quel moment durant le temps

d'assimilation. Le principe de l'AD 4D-Var repose sur la minimisation d'une fonction de coût, celle-ci mesure l'adéquation entre les observations et leurs équivalents modèles, elle tient compte des incertitudes liées aux mesures et à des informations a priori sur les paramètres à contrôler. La minimisation de la fonction de coût, par une méthode de gradient, nécessite le calcul du modèle numérique direct, du gradient de la fonction de coût par rapport aux variables à contrôler (modèle adjoint) et parfois l'utilisation d'une version linéarisée du modèle direct (modèle linéaire tangent). L'implémentation informatique de ce calcul nécessite donc d'avoir accès au code du modèle direct, de son adjoint et de son linéaire tangent. Ceux-ci nécessitent un investissement de programmation important, notamment en ce qui concerne le modèle adjoint. D'autre part, des difficultés importantes résident dans leur mise en œuvre, car à cela s'ajoute qu'il faut gérer les observations qui peuvent intervenir en n'importe quel lieu et quel que soit le temps, gérer les différentes fonctions de coût, ainsi que les matrices de variance-covariance d'erreur qui interviennent dans le processus d'assimilation. D'autre part, des développements théoriques ont été apportés aux méthodes AD 4D-Var, il s'agit de la prise en compte : des erreurs liées aux modèles, de l'information relative aux problèmes multi-échelles, du couplage de modèles numériques, des données structurées (images). D'autre part, des scénarios de minimisation complexes ont été proposés : méthode incrémentale, méthode quasi-statique. Tous ces développements introduisent des complexités importantes au niveau des algorithmes 4D-Var et des méthodes de minimisation associées. L'implémentation d'une méthode AD 4D-Var nécessite :

a. d'avoir le code du modèle numérique direct, de son adjoint et souvent de son linéaire tangent.
b. Ayant toutes ces implémentations, il faut alors, en fonction de la méthode choisie, ordonnancer les différents calculs suivant un certain scénario. En plus, dans certains cas plus complexes, il faut gérer le couplage entre modèles.

Tenant compte des complexités de ces méthodes et de l'investissement important qu'ils demandent au niveau de leur implémentation informatique, plusieurs projets ont vu le jour. L'objectif de ces projets est de fournir des outils logiciels permettant d'assister un utilisateur dans sa tâche de programmation. Historiquement, les outils logiciels d'aide qui ont été proposés visent à répondre d'une manière spécifique à l'une ou l'autre de ces deux tâches. Concernant le point (a), on trouve les logiciels de génération automatique du code adjoint à partir du code du modèle direct. Ces logiciels répondent à un besoin actuel qui consiste à exploiter de très gros codes existants dont on est sûr du bon fonctionnement. Le code adjoint généré de cette manière contient des instructions qui ne sont pas nécessaires et doit être "nettoyé" avant son utilisation, sous peine de payer un surcoût de calcul important. Cette opération doit d'autre part être recommencée si l'on fait évoluer le code direct et si l'on désire faire évoluer également son adjoint. Les logiciels qui mettent en œuvre cette approche sont connus sous le nom de *dérivateurs automatiques* ou *adjointiseurs* [Griewank et Walther, 2008]. On trouve dans cette famille les logiciels *TAPENADE*, *TAF*, *OpenAD*, etc. [Hascoët et Pascual, 2004, Giering et Kaminski, 1998, Naumann *et al.*, 2006]. Concernant le point (b), on trouve les logiciels *PALM* et *OASIS*, qui sont développés au *Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique* (*CERFACS* à Toulouse) [Fouilloux et Piacentini, 1999, Redler *et al.*, 2010]. Ces logiciels supposent que l'utilisateur dispose de programmes qui calculent le modèle numérique direct, son adjoint et si nécessaire son

linéaire tangent. A partir de ces programmes de calculs, ces logiciels permettent d'implémenter et d'exécuter divers schémas d'assimilation de données en couplant leurs différentes composantes algorithmiques d'une façon modulaire. D'autre part, une bibliothèque de programme, intitulée *VERDANDI*, est développée à l'INRIA pour aider à la réalisation de programme en assimilation de données par des non spécialistes, elle concerne actuellement les méthodes d'assimilations séquentielles et ne contient pas, dans sa version actuelle, d'aide dédiée à l'assimilation 4D-Var.

Le logiciel YAO [Nardi *et al.*, 2009], qui est développé actuellement au laboratoire *LOCEAN* (*Université Pierre et Marie Curie*) dans l'équipe *Modélisation et Méthodes Statistiques Avancées* (*MMSA*), traite simultanément les deux problématiques précédentes (a) et (b). Il est dédié aux méthodes d'assimilation variationnelle de données 4D-Var. YAO propose de modéliser le modèle direct sous la forme d'un graphe de flot de calcul appelé *graphe modulaire*. L'utilisateur n'écrit pas explicitement un programme qui implémente son modèle direct. YAO lui propose un langage de spécification qui lui permet de spécifier les entités de calculs élémentaires et leurs dépendances, il génère alors le graphe modulaire associé. Ce formalisme de graphe modulaire permet de gérer les divers scénarios de minimisation, dont les plus classiques (4D-Var, incrémental) sont préprogrammés dans la version actuelle de YAO. Le graphe est défini localement, il est donc assez simple pour un utilisateur de le conceptualiser. En effet, la tâche de l'utilisateur consiste à définir les fonctions élémentaires de calculs et leurs dépendances, YAO permet un dépliement automatique de ce graphe en temps et en espace, ce qui permet de gérer simplement des applications à une, deux ou trois dimensions. Ce formalisme donne à YAO une souplesse importante, puisque les codes du modèle direct, de son adjoint et de son linéaire tangent s'obtiennent automatiquement par l'application d'algorithmes spécifiques sur le graphe modulaire généré. Ainsi, YAO permet à l'utilisateur de réduire sa part de programmation et de se concentrer sur la spécification de son problème, en particulier sur les dépendances qui existent entre les différentes variables qui constituent le phénomène étudié. L'avantage de cette méthode repose sur sa flexibilité et la facilité avec laquelle il est possible de modifier le modèle pour le faire évoluer. Cette méthodologie a été testée sur différents modèles numériques utilisés en géophysique. Elle a montré sa capacité à produire facilement des modèles adjoints et à permettre l'assimilation des données 4D-Var. YAO s'est donc avéré être une solution efficace pour surmonter plus aisément l'investissement nécessaire à la réalisation des modèles (direct, tangent et adjoint) dont il est reconnu qu'il s'agit habituellement d'un travail lourd et contraignant. Cette souplesse de YAO le distingue des logiciels existants, elle en fait un candidat privilégié pour être une plateforme conviviale permettant de réaliser des expériences d'AD 4D-Var, de tester les modèles et de suivre leur évolution. La version actuelle de YAO a évolué progressivement, son développement a été guidé par des nécessités pratiques des utilisateurs. L'ajout d'une nouvelle fonctionnalité était suggéré par un besoin concret. Ainsi, ses spécifications et son formalisme ont été définis au fur et à mesure des applications traitées. Cette version est opérationnelle sur des applications de tailles moyennes. Mais, l'objectif à moyen terme est de faire évoluer YAO vers une plateforme complète et très générale qui doit être capable de :

- traiter des applications réelles qui nécessitent un formalisme complexe, telles que la prise en compte des erreurs modèles et des données structurées, du multi-échelle et des couplages de modèles ;
- générer automatiquement un code parallèle efficace (en temps de calcul et en gestion mé-

moire) et opérationnel sur de grandes applications, notamment en environnement.
L'objectif de notre travail est de préparer l'évolution de la version actuelle de YAO, vers cette plateforme. Ainsi, au cours de cette thèse, tout en continuant le développement et la maintenance de YAO, nous avons entrepris un travail de recherche à deux niveaux :

1. La formalisation, il s'agit de reprendre les spécifications et les directives de la version actuelle de YAO afin de les formaliser d'une manière plus générale.
2. L'algorithmique, il s'agit de proposer des algorithmes permettant l'automatisation de certaines directives, la génération automatique de l'ordre des parcours des espaces de calculs et de proposer des algorithmes pour la parallélisation automatique du code généré.

Concernant le niveau 1, notre travail a consisté à présenter et redéfinir d'une manière unifiée le formalisme sous-jacent à la version actuelle de YAO, afin de préparer le terrain pour son évolution au niveau conceptuel. L'évolution au niveau conceptuel de YAO doit continuer en collaboration étroite avec des mathématiciens et des chercheurs qui développent des méthodes en AD variationnelle. En effet, il s'agit de bien cadrer les spécifications de YAO et de proposer, relativement aux fonctionnalités attendues, un formalisme général et réaliste. Le travail de formalisation que nous avons entamé dans cette thèse ouvre la voie pour de telles collaborations. Concernant le niveau 2, nous avons proposé des algorithmes, notamment en ce qui concerne les boucles imbriquées, qui sont au cœur du calcul, afin de bien définir les stratégies de leur génération, de leur fusion et de leur décomposition, ainsi que de leur parallélisation automatique. Ce travail a permis de faire le lien entre les concepts sous-jacents à YAO et ceux qui ont été développés dans le cadre de la parallélisation automatique de boucles imbriquées, qui est un domaine de recherche bien avancé. Nous avons présenté des adaptations de ces algorithmes à YAO. Ce travail de synthèse et d'unification de concepts et des notations avec ce domaine de recherche ouvre la voie à l'intégration, dans YAO, de techniques plus avancées, notamment concernant les approches polyédriques en parallélisation automatique des boucles. L'évolution de YAO, dans le sens des deux objectifs cités ci-dessus, nécessite aussi la collaboration avec des compétences complémentaires : en modélisation numérique, en parallélisation automatique et en architectures des machines. Nous avons souhaité que ce document de synthèse serve comme outil de travail dans le cadre de ces coopérations.

Cette thèse se situe dans un contexte pluridisciplinaire, à la fois en AD et en informatique. Elle a été réalisée dans le cadre d'une collaboration pluridisciplinaire entre le LOCEAN (Laboratoire d'Océanographie et du Climat : Expérimentation et Approches Numériques) et le CEDRIC (Centre d'Etude et De Recherche en Informatique du CNAM). Plusieurs stages d'ingénieur ont été menés, dans le cadre de collaborations avec d'autres organismes, qui ont porté sur des sujets divers de YAO. En 2009 et 2010 la participation au projet SPIRALES (Soutien aux Projets Informatiques dans les Equipes Scientifiques) de l'IRD (Institut de Recherche pour le Développement) a eu pour objectif le développement de Visual YAO, une interface graphique de YAO. Le projet réunissait les partenaires suivant : LOCEAN, LSCE-CEA (Laboratoire des Sciences du Climat et de l'Environnement-Commissariat à l'Energie Atomique et aux énergies alternatives) et LTI-UCAD (Laboratoire de Traitement de l'Information-Université Cheikh Anta Diop de Dakar). Deux étudiants ingénieur (de Dakar) ont réalisé leur stage de six mois sur ce sujet, leur travail a été poursuivi pendant un an par un ingénieur. Le résultat de ce travail est une première version de l'interface graphique qui rendra YAO plus convivial et plus facile à utiliser par la communauté

de géophysiciens. D'autre part, dans la même période, un stage d'un an d'ingénieur CNAM, était mené sur l'implémentation d'une partie des algorithmes proposés dans le cadre du niveau 2 (ci-dessus). Un second stage d'ingénieur CNAM a démarré début octobre 2010, l'objectif de ce stage est d'optimiser le code généré par YAO et de revoir certaines de ces fonctionnalité afin d'implémenter des méthodes plus avancées d'AD variationnelle, telle la méthode duale. Les travaux de cette thèse ont permis aussi la collaboration avec l'équipe PEQUAN (PErformance et QUalité des Algorithmes Numériques) du LIP6 (Laboratoire d'Informatique de Paris 6). Cette collaboration a ouvert des perspectives importantes pour le développement futur de YAO.

Au chapitre 2, nous présentons les principes théoriques et le cadre mathématique général de l'assimilation variationnelle 4D-Var, qui est la méthode d'AD gérée par YAO. Des méthodes plus avancées, telles que la méthode incrémentale et la méthode duale, sont rappelées brièvement à la fin du chapitre afin de montrer la complexité de celles-ci.

Au chapitre 3, nous introduisons le formalisme du graphe modulaire et le logiciel YAO. Nous présentons l'algorithme *forward*, qui correspond à la propagation des calculs dans le graphe modulaire et qui permet de calculer le modèle direct, et l'algorithme *backward*, qui permet le calcul du modèle adjoint. Une présentation approfondie des directives YAO est donnée dans ce chapitre, celles-ci permettent la description des dépendances entre les entités de calcul qui sont à la base du graphe modulaire généré par YAO, ainsi que la descruption des directives qui permettent la traversée de ce graphe. Nous présentons ensuite une application simple sous YAO, qui correspond au modèle numérique du *Shallow-water*, il s'agit d'illustrer par un exemple les directives de YAO. A la fin du chapitre nous montrons la facilité d'implémentation des méthodes incrémentale et duale sous YAO, ce qui permet de les automatiser. Ce chapitre ne correspond pas à une présentation complète de YAO, nous limitons la présentation aux aspects qui permettront de bien comprendre les chapitres qui suivent.

Au chapitre 4, nous présentons une méthodologie afin de vérifier la cohérence des directives, spécifiées par un utilisateur. En effet, le traitement par YAO d'un problème, pour lequel l'on dispose du code de son modèle direct, nécessite l'écriture de celui-ci sous le formalisme de YAO. Il s'agit de définir l'ordre dans lequel les calculs sont effectués. Ceci est fait en utilisant deux types de directives qui redéfinissent les dépendances entre entités de calculs et l'ordre de parcours du graphe modulaire qui sera généré. Il est alors intéressant de vérifier les cohérences de ces deux types de directives : l'ordre de parcours du graphe permet-il d'exécuter toutes les entités de calculs souhaitées en respectant les dépendances imposées ? En effet, l'existence de ce type d'incohérence se répercute directement sur les résultats de l'assimilation. Nous présentons dans ce chapitre un algorithme qui permet la détection de ces incohérences.

L'écriture des directives qui permettent un parcours cohérent du graphe modulaire est une phase importante et souvent source d'erreurs, notamment quand il s'agit de grandes applications complexes. Il est alors souhaitable d'assister l'utilisateur en générant automatiquement les directives qui permettent ce parcours. Nous présentons au chapitre 5 une méthode permettant la génération automatique d'un ordre de parcours cohérent, relativement aux contraintes définies par le graphe modulaire généré. La méthode proposée permet de générer plusieurs parcours possibles, elle permet donc de proposer des parcours optimisés suivant certains critères.

Au chapitre 6, nous traitons le problème de la parallélisation automatique, en mémoire partagée, du code généré par YAO. Ce chapitre explore les algorithmes de détection des parties parallélisables du code et présente des stratégies de décomposition, de choix et de regroupement qu'il est possible d'opérer.

Enfin, au chapitre 7 nous concluons en présentant les perspectives des évolutions possibles de YAO.

Trois annexes sont également présentées à la fin de ce document. La première aborde d'une manière détaillée la thématique du *checkpointing* dans le cadre de l'AD et d'une éventuelle application à YAO. Cette annexe pose cette problématique, mais ne propose pas une solution définitive. Elle analyse aussi la taille des structures de données générées par YAO. Cette analyse permet de comprendre jusqu'où les applications peuvent être poussées en assurant le non débordement de la mémoire allouée. La deuxième annexe donne une démonstration de la procédure *backward*, qui sera présentée en détail au chapitre 3. Enfin la troisième annexe présente des considérations pratiques sur le stockage des matrices et l'implication que ceux-ci ont sur les performances d'un programme.

Chapitre 2

Principes théoriques de l'assimilation variationnelle

2.1 Introduction et notations

Ce chapitre présente l'ensemble des notions mathématiques nécessaires à la mise en œuvre des techniques d'assimilation variationnelle de données. Traditionnellement on classe l'assimilation variationnelle en différents types, selon le nombre de dimensions que l'on considère : 3D-Var signifie que l'on considère un phénomène physique décrit dans l'espace à une, deux ou trois dimensions et 4D-Var que l'on observe aussi son évolution dans le temps. L'assimilation variationnelle demande la connaissance d'un modèle numérique, ou modèle direct M, qui décrit l'évolution temporelle du phénomène physique que l'on étudie. Si l'on prend par exemple un problème géophysique, le modèle direct permet, entre autres, de relier les variables géophysiques que l'on étudie aux observations. En faisant varier certaines des variables géophysiques (les variables de contrôle) l'assimilation cherche à inférer les variables physiques qui ont engendré les valeurs d'observation. Ces variables physiques peuvent être, par exemple, des conditions initiales ou des paramètres mal connus du modèle M. Les variations considérées se font à partir d'un ensemble de valeurs initiales données aux variables de contrôle. Souvent, elles ne s'autorisent qu'à explorer un domaine limité en introduisant des valeurs d'ébauche, desquelles il ne faut pas trop s'écarter. D'une manière générale, il s'agit de déterminer le minimum d'une fonction de coût J qui mesure l'écart entre les observations et les valeurs calculées par le modèle M. Le minimum recherché est toujours obtenu à partir de la méthode du gradient. Selon les algorithmes que l'on considère, il faut alors utiliser des modèles qui se déduisent de M : le linéaire tangent et l'adjoint. Si le modèle est continu et dérivable, ces deux modèles se déduisent des équations du modèle M.

Le linéaire tangent revient à étudier, pour une des variables de contrôle données (*point de référence*), la sensibilité des valeurs de sortie du modèle M à des petites perturbations de ces variables. Il se calcule en un point de référence en linéarisant le modèle M, calcul qui correspond à la valeur de la matrice jacobienne en ce point.

Le modèle adjoint revient, quant à lui, à étudier les variations des variables de contrôle en

réponse à une perturbation des valeurs de sortie calculées par le modèle M. Il faut donc dérouler en sens inverse les calculs du linéaire tangent, ce qui revient à utiliser la transposée de la matrice jacobienne.

Lorsque des observations sont disponibles, ces modèles permettent de faire de l'assimilation variationnelle de données, en minimisant la fonction J, et de retrouver les valeurs des variables de contrôle. Nous présentons maintenant de manière formelle les notations mathématiques et les algorithmes les plus classiques utilisés actuellement. Dans cette présentation, nous avons adopté le formalisme et les notations présentés dans l'article [Ide *et al.*, 1997]. Nous notons :

- M : le modèle direct décrivant l'évolution (en général non linéaire) entre 2 temps de discrétisation t_i et t_{i+1}. Désignons par n le nombre de pas de temps.
- $\mathbf{x}(t_0)$: le vecteur d'état initial en entrée du modèle et qui est à contrôler. On suppose qu'il est de dimension N.
- $M_i(\mathbf{x}(t_0))$ ou $M(t_0,t_i)$ correspond à l'état du modèle au temps t_i à partir de l'état du modèle à t_0. On notera $\mathbf{x}(t_i) = M_i(\mathbf{x}(t_0))$.
- $\mathbf{M}(t_i,t_{i+1})$: le linéaire tangent, qui correspond à la matrice jacobienne du modèle M calculée en $\mathbf{x}(t_i)$.
- On définit le linéaire tangent du modèle M_i calculé en $\mathbf{x}(t_0)$:

$$\mathbf{M}_i(\mathbf{x}(t_0)) = \prod_{j=i-1}^{0} \mathbf{M}(t_j,t_{j+1}).$$

- L'adjoint du modèle M_i calculé en $\mathbf{x}(t_0)$ est la matrice tranposée du linéaire tangent :

$$\mathbf{M}_i^T(\mathbf{x}(t_0)) = \prod_{j=0}^{i-1} \mathbf{M}(t_j,t_{j+1})^T.$$

- $\mathbf{x}^b$: un vecteur d'ébauche, qui correspond à une estimation a priori du vecteur $\mathbf{x}(t_0)$.
- $\mathbf{y}^o$: l'ensemble des observations sur différents pas de temps. Le vecteur $\mathbf{y}_i^o$ correspond donc aux observations au temps t_i, ce vecteur peut être vide s'il n'y a pas d'observation sur ce pas de temps.

Le modèle M permet d'estimer des quantités, qui sont le plus souvent observées à partir d'un opérateur H dit d'observation. Dans le domaine de la géophysique, cet opérateur permet, par exemple, de comparer les sorties du modèle M, qui calcule la température à la surface de la mer, à des observations, enregistrées par un radiomètre qui se trouve à bord d'un satellite. On comprend également que les observations ne peuvent pas être obtenues en tout point de l'espace étudié, ni à tout instant. Il est nécessaire de connaître l'opérateur qui permet de transformer les valeurs de sortie de M en valeurs observables. C'est l'opérateur H qui permet donc de comparer, là ou les observations sont disponibles, les valeurs observées et celles calculées par la composition $H \circ M$. La fonction de coût J va donc être définie en fonction des observations, il sera nécessaire d'exprimer le linéaire tangent et l'adjoint de l'opérateur H. On note donc :

- H_i : l'opérateur d'observation qui permet de calculer les variables d'observation $\mathbf{y}_i$ au temps t_i à partir du vecteur d'état $\mathbf{x}(t_i)$. On suppose par la suite que : $\mathbf{y}_i^o = H_i(M_i(\mathbf{x}(t_0))) + \varepsilon_i$ (ε_i étant une variable aléatoire de moyenne nulle). Par la suite, ε_i est appelé erreur d'observation.

– $\mathbf{H}_i$: la matrice du modèle linéaire tangent de l'opérateur H_i calculé en $\mathbf{x}(t_i)$.

Enfin, l'assimilation des données cherche à résoudre des problèmes qui sont le plus souvent sous contraintes. Il faut, pour obtenir des solutions réalistes, apporter le plus possible d'information pour maintenir la cohérence existant entre les différentes variables de contrôle retrouvées et des quantités estimées par le modèle M. En général, l'ajustement des paramètres de contrôle consiste à maximiser la probabilité $P(\mathbf{x}(t_0)|\mathbf{y}^o)$. D'après la règle de Bayes, elle s'écrit :

$$P(\mathbf{x}(t_0)|\mathbf{y}^o) = \frac{P(\mathbf{y}^o|\mathbf{x}(t_0))P(\mathbf{x}(t_0))}{P(\mathbf{y}^o)},$$

ce qui revient à maximiser le numérateur de cette expression, car son dénominateur $P(\mathbf{y}^o)$ est constant par rapport à $\mathbf{x}(t_0)$. Si l'on définit la fonction de coût $J(\mathbf{x}(t_0)) = -\ln P(\mathbf{y}^o|\mathbf{x}(t_0)) - \ln P(\mathbf{x}(t_0))$, le problème se ramène à minimiser J par rapport à $\mathbf{x}(t_0)$. Généralement, on fait l'hypothèse que :

– Les paramètres de contrôle $\mathbf{x}(t_0)$ suivent une loi normale de moyenne $\mathbf{x}^b$, que nous appelons par la suite l'ébauche, et de matrice de variance-covariance $\mathbf{B}$. Cette hypothèse représente l'a priori que l'on suppose sur les paramètres de contrôle $\mathbf{x}(t_0)$.
– Pour tout couple de pas de temps t_i et t_j, les variables conditionnelles $\mathbf{y}_i^o|\mathbf{x}(t_0)$ et $\mathbf{y}_j^o|\mathbf{x}(t_0)$ sont indépendantes, et pour tout i la variable aléatoire conditionnelle $\mathbf{y}_i^o|\mathbf{x}(t_0)$ suit une loi normale de moyenne $\mathbf{y}_i = H_i(\mathbf{x}(t_i))$ et de matrice de variance-covariance $\mathbf{R}_i$. Cette hypothèse suppose que le modèle est parfait et que $\mathbf{y}_i^o|\mathbf{x}(t_0)$, qui modélise la loi de l'erreur ε_i des mesures par rapport à leur équivalent modèle, suit une loi normale.

Sous ces deux hypothèses, la fonction de coût J se présente alors de la manière suivante :

$$J(\mathbf{x}(t_0)) = \frac{1}{2}\sum_{i=1}^{n}(\mathbf{y}_i - \mathbf{y}_i^o)^T\mathbf{R}_i^{-1}(\mathbf{y}_i - \mathbf{y}_i^o) + \frac{1}{2}(\mathbf{x}(t_0) - \mathbf{x}^b)^T\mathbf{B}^{-1}(\mathbf{x}(t_0) - \mathbf{x}^b). \tag{2.1}$$

2.2 Calcul de l'adjoint

D'une manière générale, soit J une fonction scalaire qui mesure l'écart entre les sorties $\mathbf{y}$ d'un modèle G et les observations ; J dépend donc de $\mathbf{y}$. Pour minimiser J, on est amené à calculer le gradient de J par rapport aux variables $\mathbf{x}$ du modèle tel que $\mathbf{y} = G(\mathbf{x})$. Si l'on note $\nabla_\mathbf{y}$ (respectivement $\nabla_\mathbf{x}$) le vecteur gradient de J de l'expression (2.1) par rapport à $\mathbf{y}$ (respectivement par rapport à $\mathbf{x}$), on a :

$$\nabla_\mathbf{x} J = \mathbf{G_x}^T \nabla_\mathbf{y} J.$$

Ceci permet le calcul de $\nabla_\mathbf{x} J$ connaissant $\nabla_\mathbf{y} J$ sous la forme d'un produit matriciel de celui-ci par la matrice $\mathbf{G_x}^T$ qui est la transposée de la jacobienne $\mathbf{G_x}$.

L'assimilation variationnelle de données consiste à minimiser la fonction J par rapport à l'état initial $\mathbf{x}(t_0)$ en utilisant la méthode du gradient. Le développement du calcul donne l'expression du gradient $\nabla_{\mathbf{x}(t_0)} J$:

$$\nabla_{\mathbf{x}(t_0)} J = \mathbf{G}^T_{\mathbf{x}(t_0)} \nabla_\mathbf{y} J + \mathbf{B}^{-1}(\mathbf{x}(t_0) - \mathbf{x}^b) = \sum_{i=1}^{n} \mathbf{M}_i^T(\mathbf{x}(t_0))\mathbf{H}_i^T[\mathbf{R}_i^{-1}(\mathbf{y}_i - \mathbf{y}_i^o)] + \mathbf{B}^{-1}(\mathbf{x}(t_0) - \mathbf{x}^b). \tag{2.2}$$

La procédure de minimisation est réalisée en choisissant un algorithme d'optimisation parmi l'ensemble de ceux proposés par les techniques d'optimisation. Dans le domaine de la géophysique des minimiseurs très utilisés pour l'assimilation variationnelle sont les minimiseurs M1QN3 et M2QN1 [Gilbert et Lemaréchal, 1989], développés par l'INRIA. On peut également, de manière à favoriser pour certains problèmes la convergence et l'efficacité, utiliser une méthode approchée de descente de gradient, appelée *algorithme incrémental* [Weaver *et al.*, 2005, Courtier *et al.*, 1994]. Procéder de la sorte revient à modifier la fonction J. La présentation de cette variante fait l'objet de la section suivante.

En général, les paramètres de contrôle sont ajustés plusieurs fois avant d'arriver à un résultat acceptable. Les itérations de la méthode du gradient permettent ainsi de s'approcher de la solution cherchée jusqu'à satisfaire un critère d'arrêt qui pourrait être, par exemple, le fait que le gradient soit inférieur à une certaine valeur. La figure 2.1 montre ce processus d'ajustement progressif.

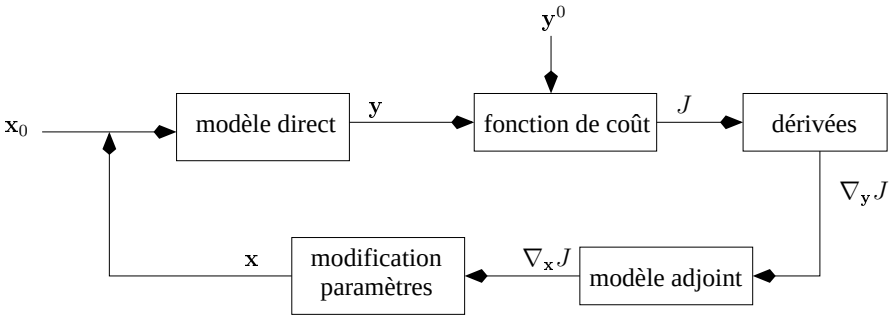

FIG. 2.1 – Itération de base de l'assimilation variationnelle de données. $\mathbf{x}_0$ est le vecteur $\mathbf{x}(t_0)$ à la première itération. L'écart entre la sortie du modèle direct $\mathbf{y}$ et les observations $\mathbf{y}^o$ est mesuré à l'aide de la fonction de coût J. Comme montré en (2.2), pour minimiser cette fonctionnelle on est amené à calculer $\nabla_{\mathbf{y}} J$ qui, par le biais du modèle adjoint, nous donne $\nabla_{\mathbf{x}} J$. Le carré modification paramètres contient la procédure de minimisation souhaitée. Ce schéma ne montre pas le deuxième terme de (2.2), celui-ci étant facile à calculer.

2.3 Méthode incrémentale

La minimisation de la fonction J par rapport à $\mathbf{x}(t_0)$ suppose que l'on positionne l'algorithme de minimisation à un état initial $\mathbf{x}^g(t_0)$, ainsi on a $\mathbf{x}(t_0) = \mathbf{x}^g(t_0) + \boldsymbol{\delta}\mathbf{x}$. On utilise l'approximation par le linéaire tangent :

$$\mathbf{y}_i = H_i(M_i(\mathbf{x}(t_0))) \simeq H_i(M_i(\mathbf{x}^g(t_0))) + \mathbf{H}_i\mathbf{M}_i(\mathbf{x}^g(t_0))\boldsymbol{\delta}\mathbf{x}. \tag{2.3}$$

En posant en plus $\mathbf{d}_i = \mathbf{y}_i^o - H_i(M_i(\mathbf{x}^g(t_0)))$ on peut exprimer J par :

$$J[\boldsymbol{\delta}\mathbf{x}] = \frac{1}{2}\sum_{i=1}^{n}[\mathbf{H}_i\mathbf{M}_i(\mathbf{x}^g(t_0))\boldsymbol{\delta}\mathbf{x} - \mathbf{d}_i]^T\mathbf{R}_i^{-1}[\mathbf{H}_i\mathbf{M}_i(\mathbf{x}^g(t_0))\boldsymbol{\delta}\mathbf{x} - \mathbf{d}_i] \tag{2.4}$$

$$+\frac{1}{2}[\boldsymbol{\delta}\mathbf{x}-(\mathbf{x}^b-\mathbf{x}^g(t_0))]^T\mathbf{B}^{-1}[\boldsymbol{\delta}\mathbf{x}-(\mathbf{x}^b-\mathbf{x}^g(t_0))].$$

Cette nouvelle formulation de J représente une expression quadratique en $\boldsymbol{\delta}\mathbf{x}$ et correspond à une approximation de $J(\mathbf{x}(t_0))$ autour de $\mathbf{x}^g(t_0)$. Le problème est de minimiser $J(\boldsymbol{\delta}\mathbf{x})$ par rapport à la seule variable $\boldsymbol{\delta}\mathbf{x}$, les autres termes étant constants. Cette phase de minimisation sera dite la *boucle interne*. Elle correspond à quelques itérations de la méthode du gradient. A chaque itération de la *boucle interne*, il faut donc calculer le gradient $\nabla_{\boldsymbol{\delta}\mathbf{x}}J$:

$$\nabla_{\boldsymbol{\delta}\mathbf{x}}J=\sum_{i=1}^{n}\mathbf{M}_i^T(\mathbf{x}^g(t_0))\mathbf{H}_i^T\mathbf{R}_i^{-1}[\mathbf{H}_i\mathbf{M}_i(\mathbf{x}^g(t_0))\boldsymbol{\delta}\mathbf{x}-\mathbf{d}_i]+\mathbf{B}^{-1}[\boldsymbol{\delta}\mathbf{x}-(\mathbf{x}^b)-\mathbf{x}^g(t_0))]. \tag{2.5}$$

Au cours des itérations de la *boucle interne*, l'approximation par la formule (2.3) risque de devenir trop grossière. Pour cette raison, on fait un nombre limité d'itérations, puis on recalcule la vraie valeur de $\mathbf{y}_i=H_i(M_i(\mathbf{x}^g(t_0)))$. La phase de préparation de la boucle interne est appelée *boucle externe*. La boucle interne est donc imbriquée dans la boucle externe. Nous pouvons résumer les calculs à chaque itération de chacune de ces boucles de la façon suivante :

Boucle externe :
- Remplacer $\mathbf{x}^g(t_0)$ par $\mathbf{x}^g(t_0)+\boldsymbol{\delta}\mathbf{x}$. Il est à noter qu'à la toute première itération $\mathbf{x}^g(t_0)$ doit être initialisé (en général égal à l'ébauche).
- Calculer les $\mathbf{d}_i$ pour la nouvelle expression de $J[\boldsymbol{\delta}\mathbf{x}]$, en calculant d'abord les $\mathbf{y}_i=H_i(M_i(\mathbf{x}^g(t_0)))$.
- Initialiser la boucle interne $\boldsymbol{\delta}\mathbf{x}=0$.

Boucle interne :
- Calculer $\nabla_{\boldsymbol{\delta}\mathbf{x}}J$ et $J[\boldsymbol{\delta}\mathbf{x}]$ selon les formules (2.4) et (2.5).
- Corriger $\boldsymbol{\delta}\mathbf{x}$ avec une méthode de minimisation adéquate (la boucle interne calcule une nouvelle valeur de $\boldsymbol{\delta}\mathbf{x}$).

La figure 2.2 illustre l'algorithme incrémental, en montrant comment progresse la minimisation par l'imbrication de ces deux boucles.

2.4 Quelques extensions

2.4.1 Transformation du vecteur d'état

En prenant $\mathbf{v}=\sqrt{\mathbf{B}^{-1}}\mathbf{x}=\mathbf{U}^{-1}\mathbf{x}$, l'expression $\frac{1}{2}(\mathbf{x}(t_0)-\mathbf{x}^b)^T\mathbf{B}^{-1}(\mathbf{x}(t_0)-\mathbf{x}^b)$ de la fonction de coût devient $(\mathbf{v}(t_0)-\mathbf{v}^b)^T(\mathbf{v}(t_0)-\mathbf{v}^b)$. Dans cette formulation, le nouveau vecteur de contrôle admet la matrice identité comme matrice de variance-covariance. Cette formulation présente plusieurs avantages, notamment en ce qui concerne la convergence de l'algorithme de minimisation de la *boucle interne*. Lorsque le vecteur $\mathbf{x}$ est fortement corrélé, certaines méthodes utilisent l'ACP, ce qui permet de proposer des nouveaux vecteurs de contrôle de dimension p très réduite ($p \ll N$), une présentation rigoureuse de cette méthode a été proposée dans [Berrada *et al.*, 2009].

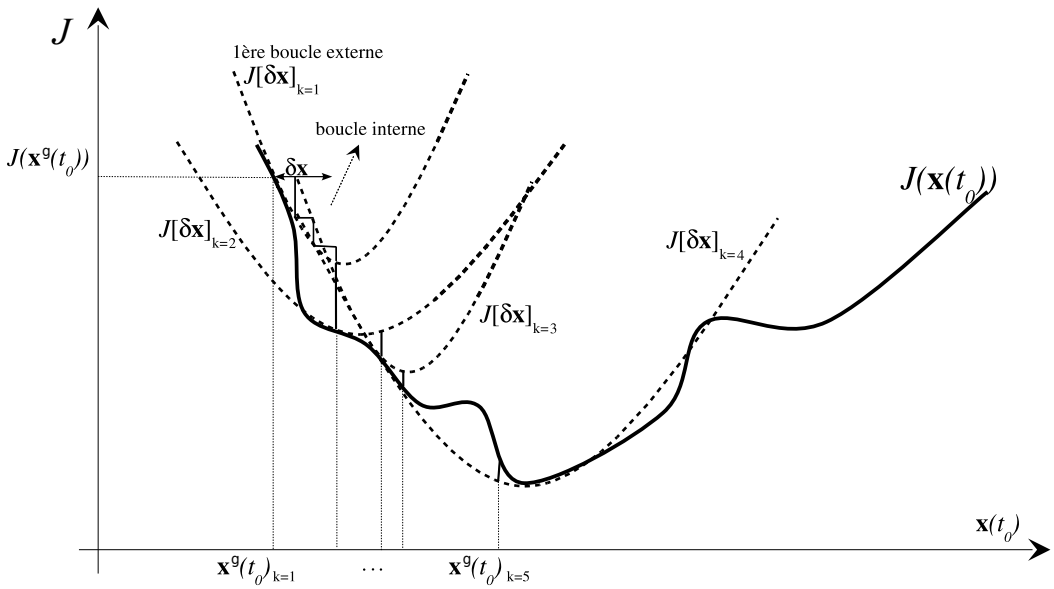

FIG. 2.2 – La courbe en trait plein correspond à la fonction de coût $J(\mathbf{x}(t_0))$. Chacune des paraboles en pointillés correspond à la fonction $J[\boldsymbol{\delta}\mathbf{x}]$ initialisée à chaque itération k de la *boucle externe.* La *boucle interne* réalise une descente de gradient sur la parabole correspondant à l'approximation. La *boucle externe*, quant à elle, permet d'avancer en se repositionnant sur la fonction $J(\mathbf{x}(t_0))$ (courbe pleine) et de faire une nouvelle approximation correspondant à la parabole suivante dans la minimisation.

Certaines méthodes [Weaver *et al.*, 2005] proposent de définir directement $\mathbf{U}^{-1}$, par exemple en définissant $\mathbf{U}^{-1} = \mathbf{F}^{-1}\mathbf{D}^{-1}\mathbf{K}^{-1}$, où :

- $\mathbf{K}^{-1}$produit un ensemble de variables 2 à 2 non corrélées, par exemple en retirant toute relation dynamique physique.
- $\mathbf{D}^{-1}$ correspond à une matrice de normalisation.
- $\mathbf{F}^{-1}$ est l'inverse d'un opérateur de lissage spécial qui agit séparément sur chacune des variables (non corrélées).

L'implication de cette formulation et la méthode incrémentale qui lui est associée suppose une estimation de $\mathbf{x}^l$ à la fin de la $l^{ième}$ itération de la *boucle externe* par : $\mathbf{x}^l = \mathbf{x}^{l-1} + \boldsymbol{\delta}\mathbf{x}^l$, avec $\boldsymbol{\delta}\mathbf{x}^l = \mathbf{U}\boldsymbol{\delta}\mathbf{v}^l$, où $\mathbf{U}$ correspond au linéaire tangent de l'opérateur $\mathbf{U}$. La boucle interne aura comme objectif l'optimisation de la fonction de coût par rapport à $\boldsymbol{\delta}\mathbf{v}$.

2.4.2 Prise en compte du bruit du modèle, méthode duale

On présente dans cette section une méthode dite *duale* qui permet de prendre en compte les incertitudes sur les modèles [Louvel, 2001]. Cette présentation sera très succincte, car l'objectif est d'illustrer la complexité des formules nécessaires à la mise en place de cette méthode. Le modèle n'étant pas une reproduction parfaite de la réalité, il est donc entaché d'erreur. Ainsi, si l'on note

$\mathbf{x}^t$ le vecteur qui correspond à la situation réelle ($\mathbf{x}$ true) et l'on note $\mathbf{x}^f$ le vecteur prévu par le modèle ($\mathbf{x}$ forecast), en posant $\mathbf{x}^t(t_0) = \mathbf{x}(t_0)$ on a alors :

$$\mathbf{x}^f(t_i) = M(\mathbf{x}^t(t_{i-1})) \quad \text{et} \quad \mathbf{x}^t(t_i) = \mathbf{x}^f(t_i) + \boldsymbol{\eta}_{i-1}.$$

On suppose que le vecteur $\boldsymbol{\eta}_i$ correspond au bruit du modèle au temps t_i (appelé aussi erreur de représentativité), et qu'il admet une matrice de variance-covariance $\mathbf{Q}_i$. On suppose aussi que, pour tout i et j, les variables $\boldsymbol{\eta}_i$ et $\boldsymbol{\eta}_j$ sont indépendantes. Il s'agit alors d'estimer, en plus des variables de contrôle, les différents vecteurs $\boldsymbol{\eta}_i$. Ainsi, la nouvelle fonction J s'écrit :

$$J(\mathbf{x}(t_0), \boldsymbol{\eta}_1, \boldsymbol{\eta}_2, \ldots, \boldsymbol{\eta}_n) = \frac{1}{2}\sum_{i=1}^{n}(H_i(\mathbf{x}^t(t_i)) - \mathbf{y}_i^o)^T \mathbf{R}_i^{-1}(H_i(\mathbf{x}^t(t_i)) - \mathbf{y}_i^o)$$

$$+\frac{1}{2}(\mathbf{x}(t_0) - \mathbf{x}^b)^T \mathbf{B}^{-1}(\mathbf{x}(t_0) - \mathbf{x}^b) + \frac{1}{2}\sum_{i=1}^{n}\boldsymbol{\eta}_i^T \mathbf{Q}_i^{-1}\boldsymbol{\eta}_i, \tag{2.6}$$

avec $\mathbf{x}^t(t_i) = M(\mathbf{x}^t(t_{i-1})) + \boldsymbol{\eta}_{i-1}$. On note par la suite $\mathbf{x}(t_0) = \mathbf{x}^b$, $\mathbf{x}(t_i) = M_i(\mathbf{x}^b)$ et pour simplifier on note $\mathbf{M}(t_i, t_{i+1}) = \mathbf{M}(t_i)$. On procède alors par approximations successives, en utilisant le linéaire tangent et en définissant :

- les vecteurs $\mathbf{x}^t(t_i) \simeq M_{i-1}(\mathbf{x}^b) + \mathbf{M}(t_{i-1})\ldots\mathbf{M}(t_0)\boldsymbol{\delta}\mathbf{x} + \sum_{k=1}^{i-1}\mathbf{M}(t_{i-1})\ldots\mathbf{M}(t_k)\boldsymbol{\eta}_{k-1} + \boldsymbol{\eta}_{i-1}$,
- les vecteurs d'innovation $\mathbf{d}_i = \mathbf{y}_i^o - H_i M_{i-1}(\mathbf{x}^b)$,
- les vecteurs $\mathbf{z}_i = \mathbf{H}_i\mathbf{M}(t_{i-1})\ldots\mathbf{M}(t_0)\boldsymbol{\delta}\mathbf{x} + \sum_{k=1}^{i-1}\mathbf{H}_i\mathbf{M}(t_{i-1})\ldots\mathbf{M}(t_k)\boldsymbol{\eta}_{k-1} + \mathbf{H}_i\boldsymbol{\eta}_{i-1}$,

où $\mathbf{M}(t_i)$ est la matrice linéaire tangent du modèle M calculée au vecteur $\mathbf{x}_i = M(t_0, t_i)$ (état du modèle au temps t_i à partir de t_0). La fonction de coût devient alors :

$$J(\boldsymbol{\delta}\mathbf{x}, \boldsymbol{\eta}, \mathbf{z}) = \frac{1}{2}\sum_{i=1}^{n}(\mathbf{z}_i - \mathbf{d}_i)^T \mathbf{R}_i^{-1}(\mathbf{z}_i - \mathbf{d}_i) + \frac{1}{2}\boldsymbol{\delta}\mathbf{x}^T\mathbf{B}^{-1}\boldsymbol{\delta}\mathbf{x} + \frac{1}{2}\sum_{i=1}^{n}\boldsymbol{\eta}_i^T\mathbf{Q}_i^{-1}\boldsymbol{\eta}_i, \tag{2.7}$$

qu'il faut minimiser sous les n contraintes linéaires :

$$\mathbf{z}_i = \mathbf{H}_i\mathbf{M}(t_{i-1})\ldots\mathbf{M}(t_0)\boldsymbol{\delta}\mathbf{x} + \sum_{k=1}^{i-1}\mathbf{H}_i\mathbf{M}(t_{i-1})\ldots\mathbf{M}(t_k)\boldsymbol{\eta}_{k-1} + \mathbf{H}_i\boldsymbol{\eta}_{i-1}. \tag{2.8}$$

Il s'agit d'un problème de programmation quadratique sous des contraintes linéaires.

En introduisant les multiplicateurs de Lagrange, en écrivant les conditions de Karush-Kuhn-Tucker (KKT) et en introduisant des substitutions intermédiaires, on obtient le problème dual qui se ramène à une fonction quadratique $G(\mathbf{m})$ à minimiser sans contraintes, relativement à $\mathbf{m}$. Les coefficients de Lagrange forment un vecteur $\mathbf{m}^T = (\mathbf{m}_1^T, \mathbf{m}_2^T \ldots, \mathbf{m}_n^T)$, où $\mathbf{m}_i$ est un sous-vecteur défini en chaque point d'observation au temps t_i.

D'autre part, les formules de transformation données par les conditions KKT sont :

$$\boldsymbol{\delta}\mathbf{x} = \mathbf{B}\sum_{k=0}^{n-1}\mathbf{M}^T(t_0)\ldots\mathbf{M}^T(t_k)\mathbf{H}_{k+1}\mathbf{m}_{k+1} \tag{2.9}$$

et

$$\eta_k = \mathbf{Q}_k^T [\mathbf{H}_k^T \mathbf{m}_k + \sum_{i=k}^{n-1} \mathbf{M}^T(t_i) \dots \mathbf{M}^T(t_{n-1}) \mathbf{H}_{i+1}^T \mathbf{m}_{i+1}]. \tag{2.10}$$

D'autre part, on a

$$\mathbf{m}_i = \mathbf{R}_i^{-1}(\mathbf{d}_i - \mathbf{y}_i). \tag{2.11}$$

En calculant le gradient $\nabla_{\mathbf{m}} G$ et en utilisant les formules de transformation on obtient :

$$\nabla_{\mathbf{m}_i} G \simeq \mathbf{H}_i \mathbf{M}(t_{i-1}) \dots \mathbf{M}(t_0) \delta\mathbf{x} + \mathbf{H}_i \sum_{k=1}^{i-1} \mathbf{M}^T(t_{i-1}) \cdots \mathbf{M}^T(t_k) \eta_k + \mathbf{H}_i \eta_i + \mathbf{R}_i \mathbf{m}_i - \mathbf{d}_i. \tag{2.12}$$

L'application de ces formules permet alternativement de calculer $\delta\mathbf{x}$, η, $\nabla_{\mathbf{m}} G$ et de faire une itérations de la méthode du gradient pour recalculer une nouvelle valeur de $\mathbf{m}$. On peut donc proposer l'algorithme suivant :

Initialisation :
- Prendre $\eta_i = 0$ pour tout $i = 1, \dots, n$.
- Faire quelques itération de la méthode incrémentale (en supposant donc $\forall i \ \eta_i = 0$). Ceci permet de calculer un vecteur $\delta\mathbf{x}$.
- Calculer $\mathbf{m}_i = \mathbf{R}_i^{-1}(\mathbf{d}_i - \mathbf{H}_i \mathbf{M}(t_i) \mathbf{M}(t_{i-1}) \dots \mathbf{M}(t_0) \delta\mathbf{x})$, ce qui est une manière d'initialiser $\mathbf{m}_i$.

Itérer les étapes suivantes :
- Calculer $\delta\mathbf{x}$ par la formule (2.9).
- Calculer $\mathbf{z}_i$ par la formule (2.8).
- Calculer η_i par la formule (2.10).
- Calculer $\nabla_{m_i} G$ par la formule (2.12).
- Modifier les paramètres $\mathbf{m}$ par la méthode du gradient.

2.5 Conclusion

Nous avons fait dans ce chapitre une présentation très succincte du formalisme de l'assimilation variationnelle de données. Nous avons aussi présenté la méthode incrémentale, qui correspond à un scenario possible pour la minimisation de la fonction de coût, ainsi que la méthode duale. Les calculs font apparaître des formules de complexité très importante, pour le calcul de l'adjoint (section 2.2) et l'application des autres méthodes. Ces formules posent de réels problèmes quant à leur mise en œuvre informatique.

Les logiciels existants, comme TAPENADE [Hascoët et Pascual, 2004], TAF [Giering et Kaminski, 1998] et OpenAD [Naumann *et al.*, 2006] sont des différentiateurs automatiques de code [Griewank et Walther, 2008]. Ils permettent de générer le code du modèle adjoint à partir du code du modèle direct. L'outil YAO, que nous présentons au chapitre suivant, est basé sur la décomposition d'un système complexe en *graphe modulaire*. Cette structure de

graphe modulaire permet d'avoir de façon quasi automatique le code direct, le code adjoint et le code du modèle linéaire tangent. Mais, en plus de l'aspect de génération de ces modèles, YAO constitue une plateforme qui permet de lancer des sessions d'assimilation variationnelle et de choisir, par exemple, la méthode incrémentale, ainsi que la méthode duale.

Chapitre 3

Le logiciel YAO

Dans ce chapitre, nous faisons une présentation générale de YAO. Nous présentons en détail le graphe modulaire, qui est la structure sous-jacente de YAO. Nous présentons ensuite les directives YAO, qui permettent la création de ce graphe et sa traversée. Avoir une maîtrise complète de ces aspects est essentiel pour comprendre le reste de ce manuscrit. L'objectif de cette thèse est de traiter l'automatisation de tâches telles que :

- le contrôle de la cohérence et
- la génération automatique de certaines spécifications de l'utilisateur, ainsi que
- la parallélisation automatique du code généré par YAO.

C'est à partir des directives YAO relatives au graphe modulaire et à son parcours que l'analyse de la génération automatique prend forme.

3.1 Concept de graphe modulaire

On définit les termes suivants :

- *Module* : un module F est une entité de calcul, il reçoit un vecteur en entrée et calcule un vecteur en sortie. Le graphe modulaire, qui est présenté ci-dessous, est formé de plusieurs modules, un module reçoit ses entrées d'autres modules ou du contexte extérieur[1] au graphe et transmet ses données en sortie à d'autres modules ou au contexte extérieur.
- *Connexion de base* : on schématise les transmissions de données entre modules par des connexions qui seront appelées par la suite les *connexions de base*. Une *connexion de base* qui lie la $i^{ème}$ sortie d'un module F_s (F source) à la $j^{ième}$ entrée d'un module F_d (F destination) indique que la $i^{ème}$ valeur calculée par F_s sera transmise à la $j^{ième}$ entrée de F_d. Les transmissions de données vers l'extérieur du graphe seront représentées par des *connexions de base* partant de sorties de certains modules et se terminant à des points particuliers que nous appelons *output data points*. De même la transmission de données du contexte extérieur vers les entrées de certains modules se fait par l'intermédiaire de *connexions de base*

[1] Le rôle du contexte extérieur au graphe sera plus clair par la suite. Pour l'instant, il est suffisant de savoir que le contexte extérieur initialise et récupère le calcul de certains modules.

qui débutent en des points qui seront dits *input data points*.

– *Graphe modulaire* : un graphe modulaire est un ensemble de modules interconnectés. Les modules représentent les sommets du graphe et un arc (orienté) du module F_s vers le module F_d signifie que F_s transmet une partie de ses sorties à F_d.

La figure 3.1a donne un exemple de modules interconnectés par des *connexions de base*. Ainsi, le module F_p transmet sa sortie y_{p1} à l'entrée x_{q1} de F_q et sa sortie y_{p2} à l'entrée x_{q2} de F_q, cette même sortie y_{p2} sera aussi transmise à l'entrée x_{l1} de F_l. La figure 3.1a représente donc les *connexions de base* décrivant la transmission de données, elle montre qu'une seule *connexion de base* peut aboutir à une entrée donnée, mais que plusieurs *connexions de base* peuvent être issues d'une même sortie. La figure 3.1b montre le graphe modulaire associé au graphe de figure 3.1a, en général un arc du graphe modulaire représente plusieurs *connexions de base*. Le graphe

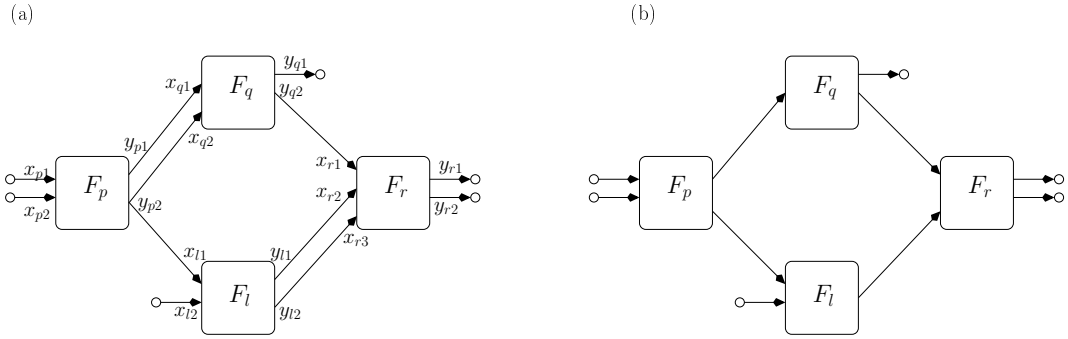

FIG. 3.1 – (a) *Connexions de base* entre modules ; les *input data points* forment le vecteur (x_{p1}, x_{p2}, x_{l2}) et les *output data points* le vecteur (y_{q1}, y_{r1}, y_{r2}). (b) Graphe modulaire correspondant.

modulaire décrit donc l'ordonnancement du calcul. Un arc de F_s vers F_d indique que le module F_d ne peut déclencher son calcul qu'après celui de F_s. Par contre, les *connexions de base* seront utilisées afin de transmettre les données.

Le graphe modulaire est un graphe sans circuit, il existe donc au moins un sommet d'entrée (sans prédécesseurs) et au moins un sommet de sortie (sans successeurs), on distingue donc trois types de modules :

– Les modules d'*entrée* (sans prédécesseurs) reçoivent leurs données uniquement du contexte extérieur du graphe et transmettent leur sorties à d'autres modules ou au contexte extérieur.
– Les modules de *sortie* (sans successeurs) transmettent leurs sorties uniquement au contexte extérieur du graphe et reçoivent des données d'autres modules ou du contexte extérieur.
– Les modules *internes* reçoivent nécessairement des entrées des modules prédécesseurs et éventuellement du contexte extérieur et transmettent les résultats aux modules successeurs et éventuellement au contexte extérieur.

Le graphe modulaire étant sans circuit, il est alors possible de numéroter les modules suivant un ordre appelé ordre topologique. Ainsi, relativement à cet ordre, l'existence d'un arc de F_s à F_d ($F_s \rightarrow F_d$) implique nécessairement que le numéro attribué à F_s est inférieur au numéro attribué à

F_d. L'ensemble des entrées d'un module F_s constitue un vecteur noté $\mathbf{x}_s$, l'ensemble de ses sorties constitue un vecteur noté $\mathbf{y}_s$ ($\mathbf{y}_s = F_s(\mathbf{x}_s)$). De ce qui précède, un module donné F_d ne peut être activé que s'il dispose de son vecteur d'entrée $\mathbf{x}_d$, ce qui implique que tous ses modules prédécesseurs F_s aient préalablement été activés. Les entrées correspondant à des *connexions de base* entrant de l'extérieur (à partir des *input data point*) sont initialisées par le contexte extérieur. Le tri topologique permet de propager correctement le calcul à travers le graphe des *input data points* aux *output data points*, tous les *input data points* du graphe sont initialisés par le contexte extérieur. La propagation des calculs intermédiaires qui suivent le tri topologique est appelée procédure *forward*.

Quand un modèle direct complexe peut être décomposé en entités de calcul plus petites, il est alors possible de le représenter par un graphe modulaire. La procédure *forward* permet de produire la valeur finale recherchée de l'ensemble du calcul de ce modèle direct. Si nous notons $\mathbf{x}$ le vecteur correspondant à toutes les valeurs *input data point* du graphe et $\mathbf{y}$ le vecteur correspondant à toutes les valeurs *output data point* du graphe, le graphe modulaire représente une fonction globale Γ et rend possible de calculer $\mathbf{y} = \Gamma(\mathbf{x})$. Ce calcul est effectué par l'algorithme *forward*.

Algorithme 1 Algorithme *forward*.

Nous supposons que, pour chaque module F_d nous pouvons calculer la fonction $F_d(\mathbf{x}_d)$, où $\mathbf{x}_d$ est son vecteur d'entrée.

1. Initialiser les *input data points*, on note $\mathbf{x}$ le vecteur défini par ses valeurs.
2. Traverser tous les modules en suivant l'ordre topologique. Pour chaque module F_d construire son vecteur d'entrée $\mathbf{x}_d$, en utilisant les *connexions de base*, et calculer $\mathbf{y}_d = F_d(\mathbf{x}_d)$.
3. Récupérer le vecteur résultat $\mathbf{y}$ sur les *output data points* du graphe. Ce vecteur représente les sorties de la fonction globale Γ.

Globalement, un graphe modulaire reçoit une partie de ses données sur les entrées de ses modules d'*entrée*, il propage le calcul dans ce graphe en récupérant éventuellement d'autres données du contexte extérieur. L'ensemble des données qui seront transmises au contexte extérieur constitue un vecteur $\mathbf{y}$, ce vecteur correspond aux résultats à restituer au contexte extérieur du graphe modulaire. Ainsi, un graphe modulaire peut être considéré lui-même comme un module F. La représentation par graphe modulaire s'applique donc à différents niveaux d'intégration, ce qui permet de former des modèles de plus en plus complexes.

La suite du chapitre montre comment, grâce à la notion de graphe modulaire, on calcule le tangent linéaire et l'adjoint d'un modèle. Les calculs nécessitent l'application de résultat de multiplications matricielles faisant intervenir, suivant les cas, les matrices jacobiennes de chaque module ou leurs transposées.

3.2 Tangent linéaire et adjoint de la fonction globale Γ

Une fois que le modèle direct complexe Γ est représenté par un graphe modulaire, il est possible de proposer deux algorithmes, qui permettent de calculer le modèle tangent linéaire et adjoint de Γ.

3.2.1 Modèle tangent linéaire d'un graphe modulaire

On suppose que, dans le cadre d'un graphe modulaire, qui implémente une fonction Γ, on sait calculer le tangent linéaire de chaque module F_p, celui-ci se calcule pour une perturbation $\mathbf{dx}_p$ (en un point de référence $\mathbf{x}_p$). On note $\mathbf{F}_p$ la matrice correspondante au tangent linéaire, qui est égale à la matrice jacobienne de F_p calculée en $\mathbf{x}_p$, la perturbation correspondante en sortie du module est égale à $\mathbf{dy}_p = \mathbf{F}_p\mathbf{dx}_p$. De la même façon que le vecteur d'entrée $\mathbf{x}_p$, le vecteur $\mathbf{dx}_p$ provient des perturbations en sorties des modules prédécesseurs de F_p ou des perturbations transmises par le contexte extérieur.

Le tangent linéaire du modèle global Γ peut se calculer par une propagation avant dans le graphe de façon similaire à la procédure *forward*. L'algorithme *linward* décrit cette procédure. Un exemple de ce calcul est donné en figure 3.2.

Algorithme 2 Algorithme *linward*.

Avant de déterminer le tangent linéaire de la fonction globale Γ pour une entrée donnée **x**, tous les vecteurs d'entrée $\mathbf{x}_p$ des différents modules F_p doivent être déterminés. Ceci est fait en exécutant la procédure *forward* avec **x** en entrée.

1. Initialiser la perturbation **dx** en attribuant à chaque *input data point i* du graphe sa perturbation correspondante $\mathbf{dx}_i$.
2. Traverser tous les modules en suivant le tri topologique. Pour chaque module F_p, considérer son vecteur de perturbation d'entrée $\mathbf{dx}_p$. Cela peut être fait en transmettant les perturbations calculées de la sortie de ses modules prédécesseurs, ou de ceux initialisés par le contexte extérieur pour les *input data points*. Ensuite calculer $\mathbf{dy}_p = \mathbf{F}_p\mathbf{dx}_p$ (la matrice jacobienne $\mathbf{F}_p$ est calculée au point de référence $\mathbf{x}_p$).
3. Récupérer le vecteur résultat **dy** sur les *output data points* du graphe. Ce vecteur représente le produit matriciel **Γdx**.

3.2.2 Modèle adjoint d'un graphe modulaire

Comme pour le modèle tangent linéaire, on suppose que l'on sait calculer le modèle adjoint de chaque module. Ainsi, pour un module F_p, ayant en entrée un vecteur $\mathbf{x}_p$, s'il reçoit un vecteur $\mathbf{dy}_p$ de même dimension que son vecteur de sortie, son modèle adjoint calculé en $\mathbf{x}_p$ est alors $\mathbf{dx}_p = \mathbf{F}_p^T\mathbf{dy}_p$, qui a la même dimension que le vecteur d'entrée de F_p. La matrice $\mathbf{F}_p^T$ est la transposée de la matrice jacobienne du module F_p calculée au point $\mathbf{x}_p$. On démontre alors (voir annexe B), que

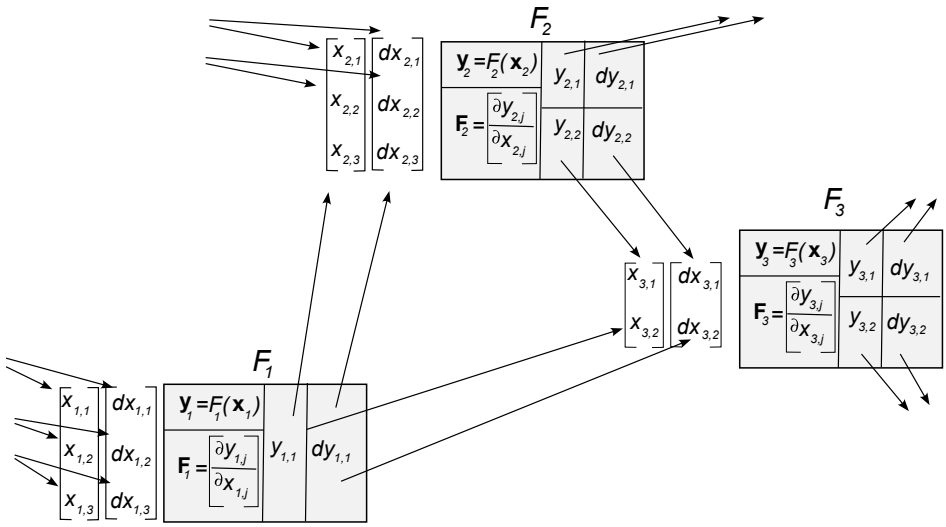

FIG. 3.2 – Calcul du modèle linéaire tangent dans un graphe modulaire. Soit F_p un module, pour lequel le vecteur $\mathbf{x}_p$ (de composante x_{pi}) a déjà été calculé, et $\mathbf{dx}_p$ sa perturbation (de composante dx_{pi}). On calcule $\mathbf{dy}_p = \mathbf{F}_p \mathbf{dx}_p$ ce qui correspond au calcul du linéaire tangent du module F_p. Les composantes dy_{pj} peuvent à leur tour être utilisées par les modules successeurs pour former leur propres composantes dx_{pi}.

l'on peut calculer le modèle adjoint du modèle global Γ pour chaque vecteur **dy** ayant la même dimension que son vecteur de sortie ($\mathbf{dx} = \mathbf{\Gamma}^T \mathbf{dy}$). On a besoin pour faire ce calcul de parcourir le graphe dans l'ordre topologique inverse, ce qui revient à utiliser les arcs en sens opposé, il s'agit du parcours en passe arrière, ou rétropropagation, du graphe modulaire.

Cet algorithme nécessite, pour chaque module source F_s, la définition de deux vecteurs $\boldsymbol{\alpha}_s$ et $\boldsymbol{\beta}_s$. Le vecteur $\boldsymbol{\alpha}_s$ a la même dimension que le vecteur d'entrée du module F_s et on note α_{si} son $i^{ème}$ élément. Le vecteur $\boldsymbol{\beta}_s$ a la même dimension que le vecteur de sortie du module F_s et on note β_{sj} son $j^{ième}$ élément. En outre, les paramètres α sont également définis pour les *output data points* et les paramètres β sont définis pour les *input data points*. Nous notons α_i le paramètre de l'*output data point i* et β_j le paramètre de l'*input data point j*. Si (s, j) est l'indice de la $j^{ième}$ sortie de F_s, on note :

- $SUCCM(s, j)$ l'ensemble d'indices (d, i), où i est la $i^{ème}$ entrée du module F_d, qui prennent la valeur de (s, j) en entrée.
- $SUCCO(s, j)$ l'ensemble de tous les *output data points*, qui prennent la valeur de (s, j) en entrée.

Si j est un *input data point*, nous notons $SUCCI(j)$ l'ensemble des indices (d, i), où i est la $i^{ème}$ entrée du module F_d, qui prend sa valeur de l'*input data point j*.

L'adjoint du modèle global Γ peut se calculer par une rétropropagation dans le graphe. L'algorithme *backward* décrit cette procédure. Un exemple de ce calcul est donné en figure 3.2.

Algorithme 3 Algorithme *backward*.

Avant d'exécuter cet algorithme, les vecteurs d'entrée $\mathbf{x}_s$ de tous les modules F_s doivent être calculés. Pour cela, il est nécessaire d'exécuter l'algorithme *forward* avec le vecteur d'entrée $\mathbf{x}$. Pour chaque *output data point* j, nous supposons que la perturbation correspondante est déjà définie par dy_j.

1. Initialiser les paramètres α_i relatifs aux *output data points* i du graphe en faisant $\alpha_i = dy_i$.
2. Traverser tous les modules dans le sens inverse du tri topologique. Pour chaque module F_s calculer $\boldsymbol{\beta}_s$ et $\boldsymbol{\alpha}_s$ comme suit :
   - Pour l'ensemble de ses indices de sortie (s, j), effectuer les opérations suivantes afin de calculer $\boldsymbol{\beta}_s$:
     a) affecter $\beta_{sj} = 0$,
     b) si $SUCCM(s, j)$ n'est pas vide alors calculer $\beta_{sj} = \sum_{(d,i)\in SUCCM(s,j)} \alpha_{di}$,
     c) si $SUCCO(s, j)$ n'est pas vide alors calculer $\beta_{sj} = \beta_{sj} + \sum_{i\in SUCCO(s,j)} \alpha_i$.
   - Calculer $\boldsymbol{\alpha}_s = \mathbf{F}_s^T \boldsymbol{\beta}_s$, où $\mathbf{F}_s^T$ est la transposée de la matrice jacobienne de F_s calculée au point de référence $\mathbf{x}_s$.
3. Pour chaque *input data point* j, calculer $\beta_j = \sum_{(d,i)\in SUCCI(j)} \alpha_{di}$.

Le vecteur $\mathbf{dx}$, dont les composantes sont les β_j de tous les *input data points* du graphe, vérifie $\mathbf{dx} = \boldsymbol{\Gamma}^T \mathbf{dy}$, où $\mathbf{dy}$ est le vecteur dont les composantes sont les valeurs dy_i définies aux *output data points* du graphe.

Remarque : Les deux algorithmes *linward* et *backward* supposent que l'on peut calculer le tangent linéaire et l'adjoint de chaque module F_s. Les modules peuvent avoir une très différente complexité. Dans un cas simple, où le module est une fonction analytique, on peut calculer la matrice jacobienne $\mathbf{F}_s$ explicitement et calculer les produits $\mathbf{F}_s \mathbf{dx}_s$ et $\mathbf{F}_s^T \mathbf{dy}_s$. En outre, il est important de définir le graphe modulaire, afin d'avoir des modules simples (petites entités de calcul) ainsi le calcul analytique de $\mathbf{F}_s$ devient facile. En ce qui concerne des modules plus complexes, l'utilisateur peut utiliser des logiciels qui font ces calculs, par exemple un code obtenu en utilisant un logiciel de différentiation automatique [Griewank et Walther, 2008, Hascoët et Pascual, 2004, Naumann *et al.*, 2006, Giering et Kaminski, 1998]. Un module lui même peut être un graphe modulaire, un réseau de neurones multicouches ou un programme précompilé. Ainsi le formalisme de graphe modulaire permet de combiner différentes méthodes pour construire des modèles de calcul complexes. Les algorithmes 2 et 3 permettent, grâce au parcours avant et arrière du graphe, de calculer directement les résultats du tangent linéaire et de l'adjoint pour des modèles complexes.

3.3 Représentation d'une application par graphe modulaire

Réaliser une assimilation variationnelle de données suppose que l'on est capable de prendre en compte des graphes modulaires de grande complexité et que l'on gère les échanges (entrées, sorties) avec l'extérieur du graphe. On peut repérer plusieurs parties du graphe présentant des compor-

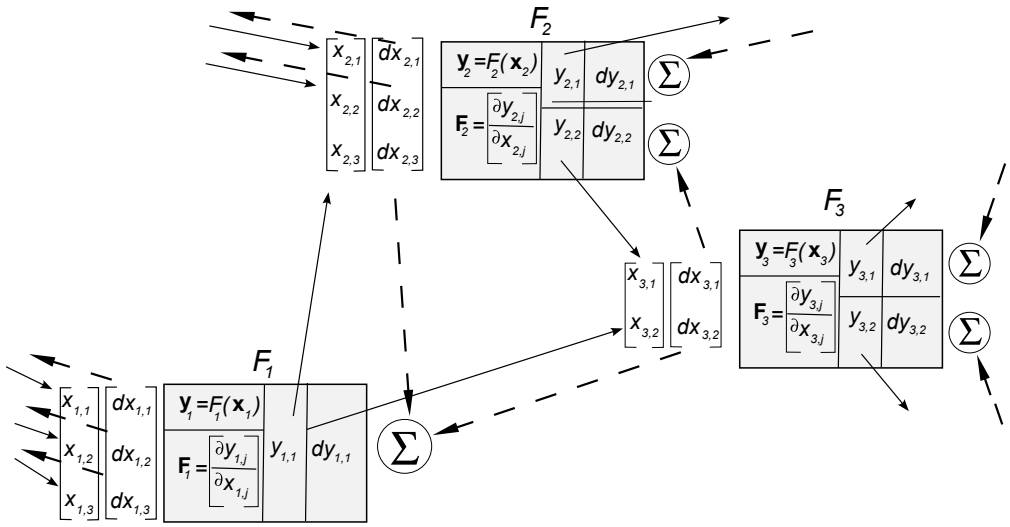

FIG. 3.3 – Calcul du modèle adjoint dans un graphe modulaire. Il s'agit du même graphe que celui de la figure 3.2. Les flèches en pointillés représentent le parcours en passe arrière pendant lequel le calcul adjoint est effectué. Pour ce calcul, on rappelle cependant que chaque composante dy_{pj} d'un module est constituée des sommes des données provenant de ses modules successeurs. Cette opération est représentée par le symbole Σ sur le graphe.

tements différents. On distingue en particulier le graphe qui correspond au modèle M qui permet de calculer l'état $\mathbf{x}(t_i)$ au temps t_i en fonction de l'état $\mathbf{x}(t_{i-1})$ au temps t_{i-1}, $\mathbf{x}(t_i) = M(\mathbf{x}(t_{i-1}))$, le graphe de M_i qui est la composition de M par elle-même i fois, qui permet de calculer l'état $\mathbf{x}(t_i)$ au temps t_i, en fonction de l'état $\mathbf{x}(t_0)$ au temps initial t_0, $\mathbf{x}(t_i) = M_i(\mathbf{x}(t_0)) = M \circ M \circ \ldots M(\mathbf{x}(t_0))$, et le graphe qui représente l'opérateur H_i. Ces éléments apparaissent dans les formules présentées au chapitre précédent. Nous noterons G_M, G_H les graphes de calcul correspondant à M et H. Le graphe représentant le problème dans son entier sera noté G_T. Il correspond à l'organisation globale de l'ensemble des opérations liant les différents graphes. Comme précédemment, le graphe G_T est défini par des échanges avec l'extérieur qui prennent en charge, par exemple, l'ensemble des problèmes liés à la fonction de coût (matrice de variance-covariance, expression moindres carrés, ...). Nous présentons ci-dessous cette organisation.

Réaliser, dans YAO, une simulation ou une assimilation de données dans un modèle géophysique opérationnel suppose que l'on est capable de définir le graphe modulaire représentant l'ensemble des calculs qui vont s'effectuer dans la fenêtre temporelle de travail. Cet ensemble de calculs présente, en général, une certaine répétitivité due au fait qu'un même phénomène physique se répète lorsque l'on se déplace dans l'espace ou dans le temps. L'ensemble de ces fonctions seront exécutées en chaque point de grille, point où les grandeurs géophysiques sont estimées. Ainsi, le graphe modulaire globale G_T va prendre en compte cette notion de répétitivité et permettre les échanges spatio-temporels. Le graphe modulaire G_T intègre 3 niveaux d'abstraction, comme illustré en figures 3.4 et 3.5 :

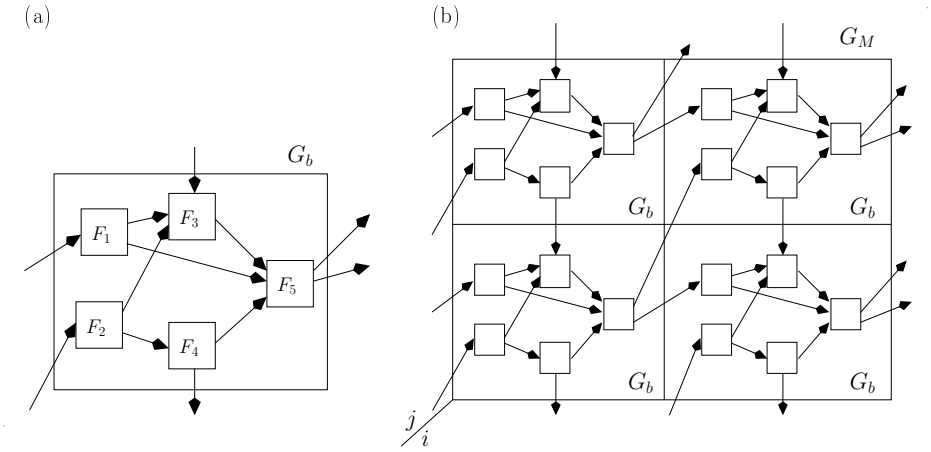

FIG. 3.4 – Deux niveaux d'abstraction de graphe : au niveau le plus bas (a), on construit le graphe dans un point de grille G_b. Au niveau spatial (b), le même graphe G_b est répliqué pour chaque point de grille, formant ainsi le graphe de l'espace G_M (en 2D dans cet exemple). Les connexions spatiales entre graphe G_b correspondent aux *connexions de base* entre modules appartenant à deux points de grille différents.

1. On détermine d'abord un sous-graphe modulaire dit de base G_b qui décrit l'ensemble des calculs intervenant à un point de grille (figure 3.4a).
2. La définition d'un modèle (1D, 2D ou 3D) à un instant t_i donné est donc un graphe modulaire dont les nœuds sont les sous-graphes modulaires G_b, les arcs décrivant les échanges spatiaux entre points de grilles. Cela définit le graphe G_M. On utilise, comme auparavant, des *connexions de base* entre modules de points de grille différents, pour décrire les calculs à effectuer (figure 3.4b).
3. La prise en compte du temps demande que l'on déplie le graphe G_M dans le temps. On obtient alors le graphe G_{M_i} au temps t_i, illustré en figure 3.5. Ce graphe peut être représenté par un graphe modulaire dont les nœuds sont représentés par les différents graphes modulaires G_M, les arcs ou *connexions de base* décrivent les échanges d'un pas de temps à l'autre.

Comme cela a été défini précédemment, on retrouve les mêmes notions d'entrée et de sortie pour chacun des graphes. Les *connexions de base* entrantes venant de l'extérieur peuvent provenir par exemple de l'initialisation ou des conditions aux limites. Les *connexions de base* sortantes de ce graphe modulaire vont communiquer leurs valeurs aux différents graphes de type G_H qui implémentent les opérateurs H. Ainsi, l'ensemble de cette articulation de graphes constitue le graphe modulaire qui permet de réaliser une session d'assimilation de données. Les points de sortie *output data points* de ce graphe correspondent aux points de mesure. YAO permet de gérer toute sorte de connexion non répétitive d'entrée/sortie.

Dans la suite, on appellera *fonction de base* la fonction réalisée par les modules en un point de grille et à un pas de temps t_i. Il s'agit de la même fonction qui est appliquée en tout point. Le terme module désignera la *fonction de base* à un point particulier de la grille et à un pas de temps

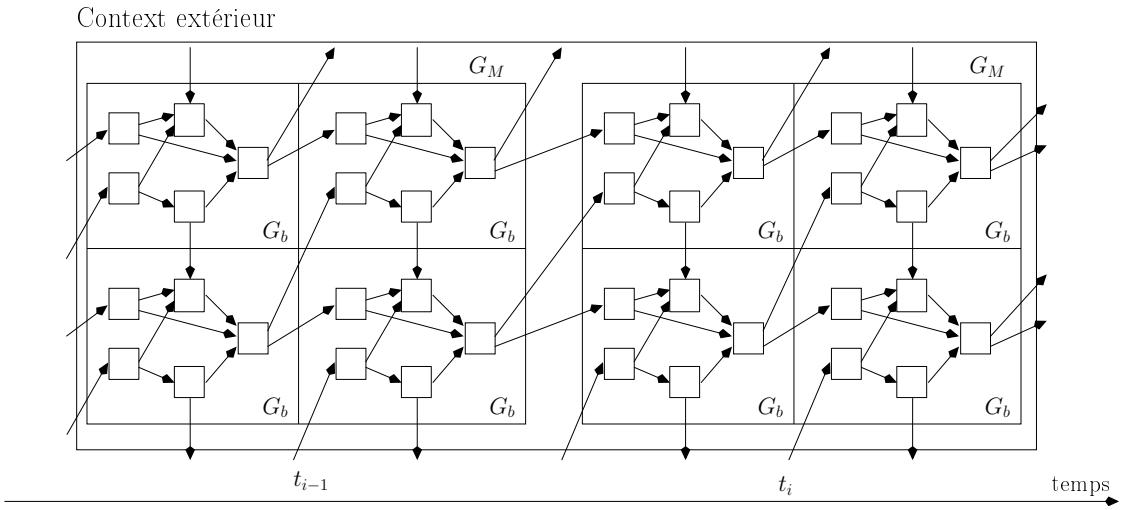

FIG. 3.5 – Le graphe modulaire composé par les G_M, qui est la répétition de G_b dans l'espace et dans le temps. Dans cet exemple spécifique, on a une fenêtre temporelle de deux pas de temps (t_{i-1} and t_i).

donné. Ainsi, la notation F_p désigne une *fonction de base* et $F_p(I,t)$ désigne un module au point de grille I et au pas de temps t. I est un vecteur à 1, 2 ou 3 dimensions, en fonction de la dimension de la grille de calcul, qui contient les coordonnées du point de grille.

Exemple de modèle et son graphe modulaire associé. L'exemple qui suit montre, sur un modèle 1D, illustré par la figure 3.6, le dépliement temporel qui permet de comprendre l'élaboration du graphe modulaire G_T. On considère un modèle, dont l'évolution de l'espace des états entre deux temps consécutifs est régie par un modèle numérique M. Prenons, pour simplifier, un modèle en colonne de coordonnée z, avec quatre variables d'état u, v, w et h. La discrétisation conduit à exprimer, pour chaque variable, sa dépendance fonctionnelle par rapport aux autres variables qui peuvent se situer sur des points de coordonnées quelconques. Supposons que la dicrétisation nous donne, par exemple, les équations suivantes (il s'agit ici d'un cas artificiel qui n'a pas de sens physique particulier) :

$$\begin{cases} u(z,t_i) = u(z,t_{i-1}) + h(z-1,t_i) & F_0 \\ v(z,t_i) = K\left(u(z,t_i)\right)^2 & F_1 \\ w(z,t_i) = u(z,t_i) + v(z,t_i) & F_2 \\ h(z,t_i) = v(z,t_i) + w(z,t_i) & F_3 \end{cases} \tag{3.1}$$

Ce système d'équations, lorsqu'il est appliqué, en chaque point de grille, pour passer d'un temps t_{i-1} au temps t_i, correspond au modèle M. Si, par exemple, la variable u est celle que nous voulons contrôler, l'ensemble des composantes $u(z,t_1)$ correspondrait à ce que nous avions noté $\mathbf{x}(t_0)$ au chapitre 2. Ainsi le vecteur d'état initial $u(z,t_1)$ en entrée du modèle est le vecteur à contrôler. Nous associons respectivement, à chacune des ce variables u, v, w et h, la *fonction de base* qui permet de la calculer. On définit pour cela les *fonctions de base* F_0, F_1, F_2 et F_3. F_0 calculera $u(z,t_i)$

en fonction de $u(z,t_{i-1})$ et de $h(z-1,t_i)$, c'est-à-dire avec la sortie de F_0 au même point de grille, mais au temps précédent et celle de F_3 au point $z-1$, et ainsi de suite pour F_1, F_2 et F_3. Sur la figure 3.6, le graphe en un point de grille représente cette situation. En chaque point, les *entrées* du graphe modulaire correspondent aux entrées de F_0 et les *sorties* correspondent aux sorties de F_3 (F_1 et F_2 sont des *fonctions de base internes*). Dans ce type de schéma, on est évidemment confronté à la phase d'initialisation et à la gestion des conditions aux limites. Sur notre figure, la prise en charge de ces éléments est représentée par ce que nous avons appelé le contexte extérieur du graphe. Ce premier graphe modulaire simple constitue notre graphe de base générique G_b. A partir de ce premier niveau d'abstraction, la figure 3.6 illustre les niveaux d'abstraction suivants G_M et G_T. Dans cet exemple, on considère qu'il n'y a pas de graphes G_H.

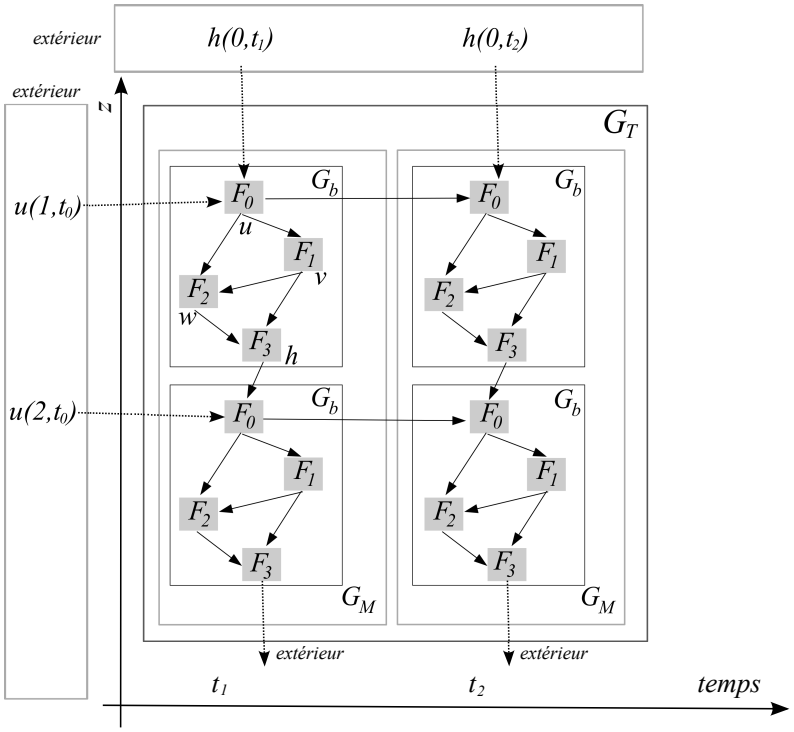

FIG. 3.6 – Graphe modulaire découlant des équations (3.1). Pour un point de grille : F_0 est la *fonction de base* d'*entrée*, F_1 et F_2 sont les *fonctions de base internes*, F_3 est la *fonction de base* de *sortie*. Calculer, en un temps t_i, tous les modules selon leur ordre dans le graphe correspond au calcul de $M_i(\mathbf{x}(t_0))$.

L'opérateur H_i exprime la partie physique liée à l'observation, son graphe correspondant G_{H_i} n'intervient qu'à certains pas de temps aux points où il y a des observations. Comme le montre la figure 3.7, les entrées/sorties se font de G_M vers G_H et de G_H vers l'extérieur pour participer, par exemple, au calcul de la fonction de coût. Ainsi, la figure 3.7 illustre une application complète.

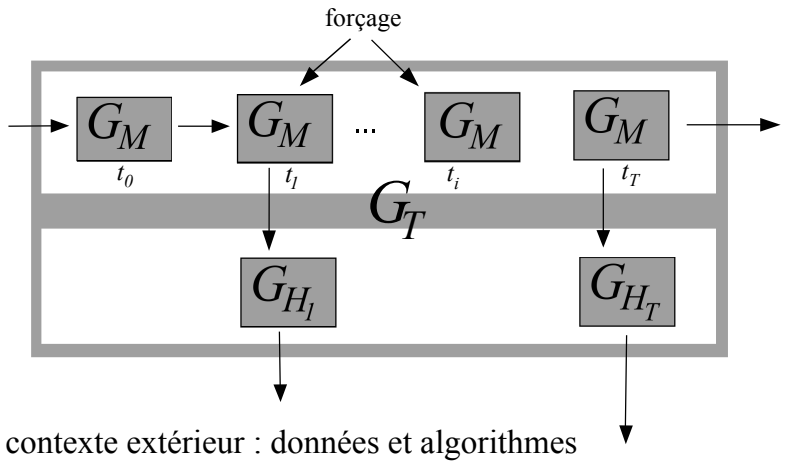

FIG. 3.7 – Graphe G_T d'une application.

Celle-ci est constituée du graphe représentant le problème physique G_T (zone foncée) et de la partie qui lui est extérieure (zone hachurée) qui contient les données et les algorithmes qui permettent d'effectuer toutes sortes de traitements sur ce graphe. Par exemple, on peut prendre en compte des forçages, charger des observations, calculer la fonction de coût, en utilisant différents algorithmes (3D-Var, 4D-Var, incrémental), lancer des tests de validation (test de la fonction objectif, test de l'adjoint), récupérer des résultats pour les visualiser, etc..

3.4 Mise au point d'expériences de simulation et/ou d'assimilation

Le formalisme de graphe modulaire qui vient d'être présenté constitue la base du fonctionnement du logiciel YAO. Il procure à YAO des algorithmes efficaces pour obtenir les calculs relatifs au tangent linéaire et à l'adjoint pour l'assimilation 3D-Var et 4D-Var. Une expérience de simulation et/ou d'assimilation demande d'intégrer et d'organiser ces calculs selon les différentes méthodologies présentées au chapitre précédent et plus généralement dans la littérature. Pour cela, la modélisation géophysique doit être effectuée et les opérateurs d'observation spécifiés.

Il faut aussi définir les conditions dans lesquelles va s'effectuer l'expérience (conditions initiales, forçages, temps et lieux auxquels seront disponibles les observations) et prendre également en compte les matrices de variance-covariance sur les erreurs d'ébauche et d'observation qui permettent de lier les variables entre elles. Chaque application demande de choisir une fonction de coût et un algorithme de minimisation adapté au problème traité. Sur un autre registre, il faut aussi posséder des outils de test et de visualisation permettant de contrôler le déroulement de l'assimilation et de juger les résultats. L'ensemble de ces opérations se situe dans la partie définie aux sections précédentes comme étant l'extérieur du modèle. YAO organise ces différentes étapes sous forme de composants qui ont des rôles spécifiques :

- Gestion de différentes sortes de données nécessaires à l'application (paramètres, forçage, etc.).
- Choix de la fonction de coût : c'est à ce niveau que doivent être intégrées les matrices (ou opérateurs) de variance-covariance sur les erreurs d'ébauche et d'observation. Certaines fonctions de coût, comme celle présentée au chapitre 2 et sa version incrémentale sont directement disponibles dans YAO. Il est par ailleurs possible d'intégrer d'autres méthodes (par exemple la méthode duale) qui vont pouvoir communiquer avec le graphe modulaire de l'application.
- Choix du minimiseur : avec YAO il est possible d'utiliser les minimiseurs de l'INRIA, M1QN3 et M2QN1 [Gilbert et Lemaréchal, 1989]. D'autres algorithmes de minimisation peuvent être interfacés.
- Outils de vérification de l'exactitude des modèles adjoint et tangent linéaires : dans YAO il est nécessaire de savoir calculer l'adjoint et le tangent linéaire de chaque module. Ceci peut se faire de différentes manières et il est important de pouvoir vérifier qu'à ce stade tous les adjoints et tangents linéaires qui ont été calculés sont exacts. YAO facilite ces vérifications, en permettant d'accéder à différents tests de validité : test de l'exactitude des matrices jacobiennes par approximation des dérivées, test sur l'application globale du tangent linéaire, de l'adjoint et de la fonction de coût.
- Visualisation des résultats : la bonne conduite d'une expérience de simulation et/ou d'assimilation demande de visualiser le comportement des différentes variables, des gradients et des fonctions de coût pendant la période considérée. YAO permet d'accéder à l'ensemble de ces quantités et l'utilisateur peut donc, en communicant avec le graphe de l'application, les transmettre vers l'extérieur, pour écrire les programmes, ou utiliser les outils de visualisation qu'il juge utiles.

3.5 Présentation de YAO

Comme nous l'avons déjà signalé, YAO repose sur le formalisme du graphe modulaire et il est dédié aux modèles numériques. Ainsi, l'utilisateur doit :

- déclarer les différentes *fonctions de base* F_p, définir les codes qui permettent de calculer chaque fonction et son adjoint ;
- préciser les différentes dépendances qui permettent de générer les *connexions de base*. Il s'agit de déclarer le graphe modulaire associé à l'application ;
- préciser un ordre de parcours du graphe modulaire qui correspond à un ordre topologique.

Tenant compte de ces informations, YAO génère le graphe modulaire. Il peut alors appliquer les procédures *forward*, *linward* et *backward*, ce qui lui permet de calculer respectivement le modèle direct, son modèle tangent linéaire et son modèle adjoint. Pour cela YAO génère un code C++ lui permettant de faire tous ces calculs. YAO n'est donc pas un différentiateur automatique, car il ne génère pas le code adjoint à partir du code du modèle numérique direct. En effet, l'utilisateur doit reformuler son modèle direct dans le formalisme du graphe modulaire. Ainsi, les modèles direct et adjoint ne se présentent pas sous la forme de codes distincts, mais comme les résultats de l'application d'algorithmes sur le graphe modulaire. Cette propriété est très importante, car elle permet à

l'utilisateur de se concentrer sur la spécification de son modèle direct (son graphe modulaire) et de réduire d'une manière significative sa part de programmation, seule la programmation du code lié aux différentes *fonctions de base* reste sous sa responsabilité. Cet effort de spécification du modèle direct est très bénéfique pour la maintenance et le suivi des différentes modifications du modèle numérique. En effet, les chercheurs apportent en permanence des modifications aux modèles numériques, soit en raffinant les lois physiques sous-jacentes ou en passant à des résolutions spatiales plus fines. Il suffit alors à l'utilisateur d'apporter ces modifications aux spécifications relatives du graphe modulaire et le reste est géré par YAO. Il n'a donc pas à apporter ces modifications en rentrant dans les différents codes.

En plus de son aspect générateur du modèle direct et adjoint, YAO est aussi une plateforme complète pour lancer des sessions d'assimilation de données. En effet, l'utilisateur peut spécifier une méthode et un scénario pour la minimisation de la fonction de coût, spécifier des paramètres et sauvegarder des résultats sur des fichiers, etc.. Ainsi, YAO est un outil complet permettant de générer un code pour exécuter des sessions d'assimilation variationnelle de données.

La figure 3.8 affiche l'architecture YAO : le grand cadre rectangulaire contient les procédures YAO qui permettent de générer le graphe modulaire. Le fichier *description*, le fichier *chapeau*,

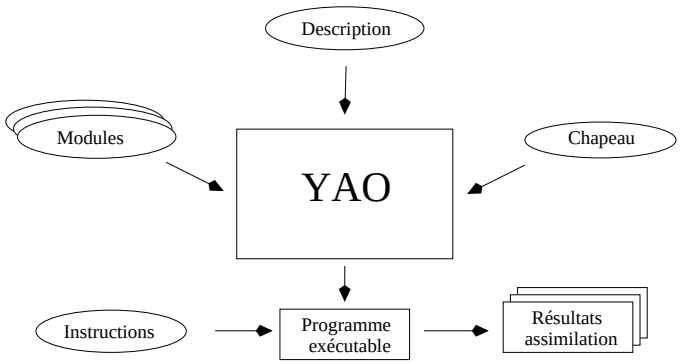

FIG. 3.8 – Représentation schématique de YAO. La génération de code commence à partir du fichier description, qui représente la spécification du modèle, et les fichiers module. Le code généré, le fichier exécutable, s'exécute par le biais d'un fichier instructions, qui permet de contrôler la production des résultats. Le fichier chapeau permet d'intégrer au code généré certains points d'entrée d'utilisateur, par la définition des fonctions utilisateur.

les fichiers *module* et le fichier *instruction* sont spécifiés par l'utilisateur. YAO utilise les fichiers description, chapeau et module pour générer le programme exécutable. Une fois le programme exécutable créé, le fichier instructions est utilisé pour exécuter les instructions de l'utilisateur. La modification d'un modèle consiste donc à modifier certains modules ou certaines directives YAO dans le fichier description. La modification de l'exécution d'une application YAO consiste à modifier les instructions YAO dans le fichier instructions.

3.5.1 Le fichier description

Le fichier description contient les directives YAO, qui définissent les caractéristiques du modèle numérique direct. Une directive YAO est une phrase reconnue par le *langage description* YAO. Elle est composée d'un ou plusieurs mot-clés et d'autres mots que nous appelons *options*. Les options permettent de spécifier les paramètres d'une directive.

L'objectif des directives est le suivant :
- Définir la fenêtre temporelle du modèle numérique, la *trajectoire*.
- Définir la grille de calcul (notée aussi *espace*) et sa dimension (1D, 2D, 3D).
- Définir les *fonctions de base*, en indiquant pour chacune le nombre d'entrées/sorties, sa participation au calcul de la fonction de coût, etc..
- Construire le graphe en définissant les *connexions de base* entre les modules.
- Indiquer l'ordre de calcul, l'ordre dans lequel les points de la grille doivent être parcourus.
- Définir un contexte pour l'application : le minimiseur utilisé, l'algorithme incrémental, le type de données utilisées (*flot* ou *double*), le test du gradient, etc..

Le fichier description contient toutes ces informations qui seront, utilisées par YAO pour générer le code exécutable. La section 3.7 présente un exemple de fichier description du modèle numérique *Shallow-water*.

3.5.2 Le fichier instructions

Des instructions YAO pour l'exécution du modèle dans une configuration particulière (durée de la simulation, valeurs initiales des paramètres, introduction d'une valeur d'ébauche pour la fonction de coût, etc.) sont contenues dans le fichier instructions. Comme pour le langage de description YAO, le *langage des instructions* permet d'identifier des phrases. Chaque phrase est une commande interprétée et exécutée par l'application générée. Toutes les commandes sont exécutées en séquentiel, le fichier instructions est une sorte de fichier *batch* et l'application est exécutée en mode *batch*. Parfois, l'utilisateur peut être intéressé par une exécution interactive. Ainsi, le fichier instructions n'est pas obligatoire, l'utilisateur peut utiliser l'option d'interactivité. Dans ce cas, le langage des instructions permet de contrôler le flot d'exécution, de modifier certains paramètres pendant l'exécution, d'écrire les résultats sur un fichier ou de prendre en considération un fichier d'observation. Le fichier instructions permet d'exécuter des expériences jumelles et des expériences réelles d'assimilation de données.

3.5.3 Les fichiers module

Ces fichiers contiennent le code source dans lequel les lois physiques du modèle numérique sont programmées ainsi que sa matrice jacobienne. Dans la pratique, l'utilisateur crée un fichier module pour chaque *fonction de base* F_p [2]. La *fonction de base* est exécutée, par le code YAO généré, pour chaque point de grille et pas de temps, ce qui correspond au module noté $F_p(I,t)$ du graphe modulaire. YAO génère une classe C++ pour chaque *fonction de base*. Les principales

[2] L'abus de langage "module" au lieu de "fonction de base" est utilisé dans YAO.

méthodes de la classe sont les méthodes dites *forward* et *backward*, qui permettent la mise en œuvre des algorithmes de propagation et rétropropagation présentés au début de ce chapitre.

Les notations YAO peuvent prêter à confusion, puisque les méthodes *forward* et *backward* des différentes classes ont les mêmes noms que les algorithmes du graphe modulaire. Une fois que ces méthodes sont implémentées par l'utilisateur, YAO s'occupe de la mise en œuvre et l'enchaînement des procédures *forward*, *linward* et *backward*[3], qui réalisent, elles, les algorithmes de même nom. Une part des pseudocodes de ces procédures est donnée au chapitre 6. Pour l'instant, il faut garder à l'esprit que les méthodes *forward* et *backward* des *fonctions de base* permettent le calcul des sorties et de sa matrice jacobienne de la *fonction de base* relative au fichier module[4]. Ces méthodes contiennent les seules informations que l'utilisateur doit spécifier pour avoir la mise en œuvre complète des algorithmes *ward*. L'utilisateur spécifie les méthodes de chaque fichier module et YAO les déplie sur tous les points (I,t). Ainsi, les méthodes *forward* et *backward* sont automatiquement intégrées par YAO dans les classes générées à partir des fichiers module.

Soit $f(x) = 2x_1^2 + x_2$ une fonction analytique implémentée par un fichier module. Alors, la méthode *forward* contient le code de programmation qui réalise le calcul $y = f(x)$. En revanche, la méthode *backward* contient le code qui réalise le calcul de la matrice jacobienne. Cette *fonction de base* admet deux variables en entrée (x_1, x_2) et une variable en sortie (y). Ainsi, la matrice jacobienne est une matrice de dimension 1×2 : ($4x_1$, 1). Cet exemple de fichier module souligne que YAO n'est pas un différentiateur automatique. La matrice jacobienne doit être écrite explicitement par l'utilisateur, dans les méthodes *backward*. Un outil de différentiation automatique peut être utilisé pour écrire automatiquement les méthodes *backward*, plutôt que de coder directement ces matrices[5]. Quoi qu'il en soit, l'écriture des matrices jacobiennes dans YAO n'est pas très coûteuse en temps car, en général, les *fonctions de base* sont petites et les matrices ne sont que de quelques lignes et colonnes.

En plus des deux méthodes, la classe module a des attributs tels que *state* et *gradient*. L'attribut *state* est un vecteur qui représente l'état d'un module $F_p(I,t)$ et correspond aux sorties du module. L'attribut *gradient* est aussi un vecteur et représente le calcul nécessaire à la rétropropagation, donné par le produit de la matrice jacobienne et le vecteur **dy** comme expliqué à la section 3.2. YAO écrit automatiquement dans les attributs *state* et *gradient*, fait le produit, lance la propagation ou la rétropropagation propres aux procédures *ward*.

3.5.4 Le fichier chapeau

Le fichier chapeau est une sorte de programme principal (*main program*) de l'application générée. L'utilisateur doit définir dans ce fichier les fonctions, qui permettent à l'utilisateur de prendre la main pendant l'exécution de l'application (au début de l'application, avant/après la procédure

[3]Quand on se réfère aux trois procédures *forward*, *linward* et *backward*, on utilise souvent l'expression "les procédures *ward*". De même pour les méthodes et les algorithmes relatifs à ces procédures.

[4]Les méthodes *forward* et *backward* seront aussi appelées les fonctions *forward* et *backward* locales, à ne pas confondre avec les procédures *forward* et *backward*, qui s'appliquent sur le graphe modulaire.

[5]Des travaux futurs intégreront probablement un différentiateur automatique dans le générateur YAO pour épargner cette tâche à l'utilisateur.

forward, avant/après la procédure *backward*, etc.). En outre, ce fichier permet à l'utilisateur de définir des variables globales et les fonctions utilisateur, qui sont spécifiques à l'application particulière.

3.5.5 Applications et quelques considérations techniques

Bien que YAO exploite les langages C/C++, il est néanmoins possible d'établir des liens avec d'autres langages de programmation. YAO fournit des fonctionnalités d'un outil intégré. Par exemple, il gère l'interface avec des minimiseurs tels que M1QN3 et M2QN1 [Gilbert et Lemaréchal, 1989][6], il peut gérer les perceptrons multicouches, il comprend une fonction de coût générale qui tient compte de l'ébauche et des opérateurs de variance-covariance. En outre, YAO gère des applications avec multi-trajectoires, multi-espaces et multi-dimensionnelles (jusqu'à 3D). YAO a déjà été testé avec succès sur plusieurs modèles en océanographie. Il a été appliqué, entre autres, dans les applications suivantes :

- **Couleur de l'océan** : inversion variationnelle des mesures multi-spectrales satellitaires de la couleur de l'océan pour la restitution de la chlorophylle-a [Brajard *et al.*, 2006].
- **Acoustique marine** : inversion variationnelle du profil de la vitesse du son et récupération de paramètres géoacoustiques (célérité, densité, atténuation, ...) [Hermand *et al.*, 2006, Badran *et al.*, 2008, Berrada, 2008].
- **PISCES** : assimilation de données variationnelle sur la couleur de l'océan dans un modèle biogéochimique [Kane *et al.*, 2006].
- **NEMO** (*Nucleus for European Modelling of the Ocean*) : modèle adjoint de la configuration GYRE [Madec, 2008].

3.6 Présentation de quelques directives de description

Cette section présente d'une manière détaillée cinq directives du fichier de description : les directives *trajectory*, *space*, *module*, *ctin* et *order*. Ces sont elles qui interviendront le plus dans le travail réalisé dans cette thèse. C'est par le biais de ces directives que l'utilisateur spécifie le graphe modulaire associé à son modèle numérique direct, ainsi que l'ordre de son parcours.

3.6.1 Directive *trajectory*

La forme générale de la directive *trajectory* se présente de la manière suivante :

trajectory *nom nb_temps*.

nom est le nom donné à la trajectoire. *nb_temps* est le nombre de pas de temps de la trajectoire discrétisée.

[6] Comme exemple de lien avec un code écrit dans un langage de programmation différent, les minimiseurs de l'INRIA M1QN3 et M2QN1 sont écrits en FORTRAN.

3.6.2 Directive *space*

Cette directive permet à l'utilisateur de définir les grilles de calcul, également appelées *espaces* (*space*). La forme générale de la directive *space* se présente de la manière suivante :

space *nom* $taille_i$ [$taille_j$ [$taille_k$]] *trajectoire*.

Un espace est défini par son *nom*, sa dimension et la $taille_l$ ($l \in \{i, j, k\}$) de chaque axe l. La dimension représente le nombre d'axes (1, 2 ou 3) et les tailles représentent la résolution, c'est-à-dire le nombre de points de grille de chaque axe. Ainsi, si uniquement la $taille_i$ est définie, la dimension sera 1, si le couple $taille_i$ et $taille_j$ est défini, la dimension sera 2, et si les trois $taille_i$, $taille_j$ et $taille_k$ sont définies, la dimension sera 3. La version actuelle de YAO gère des espaces de calcul de dimension trois au maximum. Les valeurs entre crochets sont optionnelles. L'utilisateur peut définir plusieurs espaces de calcul, qui correspondent à des degrés de résolution différents, les tailles des espaces sont différentes. Dans la suite, nous supposons que les trois axes sont ordonnés et nous les notons respectivement i, j et k. Nous notons N_i, N_j et N_k le nombre de points de grille ($taille_l$) de chacun des axes i, j et k. A partir de chaque espace de calcul, il est possible de définir des espaces de dimensions plus petites qui correspondent à des projections. Ainsi, si S est un espace de calcul de $dim\ N$ ($N \leq 3$), alors une projection S' de S aura la dimension $N-1$ ou $N-2$. Comme il s'agit d'une projection, les axes communs doivent avoir les mêmes résolutions. Dans YAO, S' sera défini par référence à S en le déclarant comme projection de S. D'autre part, un espace de calcul dans YAO est toujours associé à une trajectoire temporelle spécifiée par l'utilisateur. La trajectoire temporelle d'un espace de calcul se transmet à toutes ses projections.

3.6.3 Directive *module*

La forme générale de la directive *module* se présente de la manière suivante :

module *nom* **space** *nomEspace* **input** *n* **output** *m* options.

Cette directive spécifie une *fonction de base* appelée *nom* admettant un vecteur de dimension n en entrée et un vecteur de dimension m en sortie. Il s'agit d'une fonction particulière de $\mathbb{R}^n$ dans $\mathbb{R}^m$ dont le code sera spécifié dans un autre fichier, le fichier module. D'autre part, cette *fonction de base* est rattachée à l'espace appelé *nomEspace*. Cet espace peut être l'un des espace de calcul, déclarés par l'utilisateur, ou l'une de leurs projections. Les directives *module*, définies par l'utilisateur, permettent à YAO de générer tous les modules (les sommets) du graphe modulaire. Ainsi, pour une *fonction de base* F_p (définie par l'utilisateur) et qui est définie sur un espace S donné, YAO génère l'ensemble des modules du graphe modulaire qui lui sont associés et localisés en tous les points de S.

Nous rappelons qu'un module du graphe modulaire est le calcul de la *fonction de base* F_p dans un point spécifique I de son espace (à 1, 2 ou 3 dimensions) et au pas de temps t de sa trajectoire associée. Ce module est noté $F_p(I,t)$. Une *connexion de base* d'un module source $F_s(I',t')$ à un module destination $F_d(I,t)$ représente une transmission de données entre ces deux modules. L'utilisateur doit spécifier l'ensemble des *connexions de base* qui lui correspond. La directive *ctin*, que nous présentons en détail au paragraphe qui suit, lui permet d'introduire ce type de connexions. Dans la suite, on utilisera toujours la notation I' et I respectivement pour les vecteurs de localisa-

tion des modules source et destination.

Enfin, l'utilisateur doit préciser, à la fin de la directive, certaines options particulières. Dans la section 3.7, on verra en pratique l'utilisation des options *cost* et *target*.

3.6.4 Directive *ctin*

Références relatives

La directive *ctin* permet de spécifier des *connexions de base* entre deux modules du graphe modulaire. Ces deux modules sont rattachés à une même résolution spatiale, le nombre de points de grille est le même sur les axes communs. On suppose que l'espace S' rattaché au premier est de dimension plus petite ou égale à celui du second. La directive *ctin* se présente de la manière suivante :

ctin *moduleDestination u* **from** *moduleSource v* $\mathbf{i}+d_i$ [$\mathbf{j}+d_j$ [$\mathbf{k}+d_k$]] $\mathbf{t}+d_t$.

Cette directive précise que la $u^{ième}$ entrée du module *moduleDestination* récupère sa valeur de la $v^{ième}$ sortie du module *moduleSource*. La fin de cette instruction précise la localisation relative, dans l'espace et le temps, de ces deux modules, les valeurs entre crochets dépendent de la dimension de l'espace associé par l'utilisateur à la *fonction de base moduleSource*. Ainsi, dans le cas d'espace source de dimension 3 et pour une position au point de grille $I=(i,j,k)$ et au pas de temps t, de la *fonction de base moduleDestination*, la position relative du module source est localisée au point de grille $I'=(i+d_i, j+d_j, k+d_k)$ [7] et au pas de temps $t'=t+d_t$. Ainsi, d_i, d_j et d_k correspondent aux distances algébriques entre les deux points de grilles sur les différents axes, elles sont des entiers négatifs, positifs ou nuls. La quantité d_t représente le délai temporel entre les deux temps, il doit être un entier négatif ou nul[8]. Dans le cas d'espace source à dimension 2, la position relative du *moduleSource* est $(i+d_i, j+d_j)$ et au pas de temps $t'=t+d_t$. De même pour la dimension 1. Ce type de connexion correspond à ce que nous appellerons par la suite connexion avec *références relatives*, car la localisation du module source se définit relativement à la position du module destination. Ainsi, les cas cités ci-dessus correspondent à 3, 2 ou 1 références relatives.

Remarque : cas de projection de l'espace. Si S est un espace de dimension N ($N \leq 3$), et S' est une projection de S, les références relatives ne sont définies que pour les axes communs. Ainsi, si S est 3D et S' est 2D (défini sur les axes i, j), le point I' a la forme $(i+d_i, j+d_j)$ et le point I a la forme (i,j,k). Cette connexion permet la transmission de données de $F_s(i+d_i, j+d_j, t+d_t)$ de S' vers $F_d(i,j,k,t)$ de S pour toutes les valeurs de k. Une condition imposée par YAO à la définition des *ctin* est que $S' \subset S$ (inclus ou égal à). Ceci découle directement de l'expérience pratique et des modèles numériques qui ont été développés avec YAO.

[7] La dimension du vecteur I' dépend de l'espace de calcul. Il peut être donc à une dimension $I'=(i+d_i)$, à deux dimensions $I'=(i+d_i, j+d_j)$ ou à trois dimensions $I'=(i+d_i, j+d_j, k+d_k)$. De même pour le vecteur I.

[8] YAO n'autorise pas des valeurs $d_t > 0$, car le modèle évolue du passé vers le future.

Références absolues

Au lieu d'avoir une *référence relative*, YAO permet de définir une *référence absolue*, par rapport à un axe l ($l \in \{i, j, k\}$), qui se caractérise par une valeur constante de la composante l du point I'. Considérons une connexion entre deux *fonctions de base* F_s et F_d associées à deux espaces S' et S. Si l'on suppose que $S = S'$ et qu'ils sont de dimension trois, une *connexion de base* de $F_s(i+d_i, j+d_j, abs_k)$ à $F_d(i, j, k)$ signifie que F_d localisée au point $I = (i, j, k)$ reçoit une entrée de F_s localisée en $(i+d_i, j+d_j, abs_k)$. De même si l'on suppose S' de dimension 3 et S de dimension 2, une *connexion de base* de $F_s(i+d_i, j+d_j, abs_k)$ à $F_d(i, j)$ signifie que F_d localisé au point $I = (i, j)$ reçoit une entrée du module source F_s localisé au point $(i+d_i, j+d_j, abs_k)$, où $abs_k \in \mathbb{N}$ est une constante $1 \leq abs_k \leq N_k$, et N_k est la *taille* de l'axe k de l'espace S'. Ceci n'est pas en contradiction avec la condition $S' \subset S$ de la directive *ctin* puisque la valeur constante de l'axe k sélectionne un sous-espace affine 2D de l'espace du module source.

Dans l'exemple précédent, pour chaque point de la destination I, la composante qui concerne l'axe k du point de la source I' est constante. Toutefois, nous pouvons avoir plus d'une *référence absolue* dans une même directive *ctin*. La forme générale de la directive *ctin* est :

ctin *moduleDestination u* **from** *moduleSource v* (**i** $+ d_i$ ou abs_i) [($j + d_j$ ou abs_j) [($k + d_k$ ou abs_k)]] **t** $+ d_t$.
Ainsi, la même directive *ctin* peut contenir des références relatives et des références absolues. D'un point de vue géométrique, cette directive permet de définir des *connexions de base* qui concernent uniquement les modules sources localisés sur un sous-espace affine de S' (correspondant à certaines composantes constantes).

Remarque : La directive *ctin* définit des *connexions de base* entre modules localisés respectivement dans deux espaces S' et S qui correspondent à une même résolution de calcul (même taille de discrétisation sur chaque axe). Mais des *connexions de base* peuvent exister entre modules localisés en des espaces S' et S correspondant à des degrés de résolution différents. Cette situation concerne le couplage de modèles numériques. Chaque modèle numérique est défini sur un espace et le couplage permet la communication entre eux. En général, les tailles des axes des deux modèles ne sont pas les mêmes. Dans ce cas, la connexion entre les deux modules, à un pas de temps t, a toujours le même sens, à partir de S' à S. Ce type de connexion est un opérateur de *mapping*, défini entre deux espaces ; cet opérateur connecte un point de la grille S' à un point de la grille S. Dans ce travail, nous ne traitons pas l'opérateur de *mapping* qui ne fait pas encore partie de la version actuelle de YAO.

3.6.5 Directive *order*

L'application des procédures *forward, linward, backward* suppose que l'on dispose d'un ordre topologique, sur le graphe modulaire, permettant d'exécuter le calcul de tous ses modules. Dans le cas d'un graphe modulaire qui représente un modèle numérique, il est possible de traverser le graphe modulaire, en définissant des parcours sur les points de grilles de ses différents espaces de calculs et en exécutant à chaque fois des modules correspondant à des *fonctions de base* bien dé-

terminées. Ainsi, pour un pas de temps donné t, les directives *order* permettent le parcours de tous les modules du graphe correspondant au temps de calcul t. La directive *order* permet à l'utilisateur de définir de tels parcours. L'utilisateur peut spécifier les traversées, qui sont exprimées avec des boucles imbriquées sur les axes de la grille. Les boucles permettent d'écrire le parcours sous une forme compacte ; par ailleurs les boucles permettent d'avoir une idée géométrique visuelle de la traversée sur la grille. Les directives *order* permettent de définir ces boucles. Elle permettent de visiter tous les points de grille des différents espaces par la génération de boucles imbriquées. Un programme généré par YAO contient une boucle extérieure, que l'utilisateur ne peut pas gérer directement, et qui représente la coordonnée temporelle, la trajectoire. Une trajectoire est traversée par YAO dans un sens ascendant, du début à la fin. Dans la boucle extérieure relative au temps, YAO génère, en utilisant les directives *order*, les différentes boucles permettant de faire tous les calculs pour un pas de temps donné t. La directive *order* se présente de la manière suivante :

order *sensAxe listeInstructions*.

sensAxe : précise l'axe à parcourir et le sens de parcours de cet axe. Dans le formalisme YAO l'axe i est noté 1, l'axe j 2 et l'axe k 3. Ainsi, *YA1* (YAO *Afterward* axe *1*) signifie que nous gérons la boucle liée à l'axe i et nous traversons cet axe de manière ascendante. *YA2* signifie la même chose, mais pour l'axe j, en revanche *YB1* (YAO *Backward* axe *1*) signifie que nous traversons l'axe i de manière descendante.

listeInstructions : est une suite d'instructions qui peuvent correspondre à des *fonctions de base* ou à une nouvelle boucle définie par une autre directive *order*. Un exemple de directives *order* et son code généré par YAO est donné en figure 3.9. Dans cet exemple, l'instruction de la ligne 1 définit

```
1: order YA1
2:       A
3:       order YB2
4:             B C
```

```
pour i = 1 à N_i faire
   A(i)
   pour j = N_j à 1 faire
      B(i,j)
      C(i,j)
   fin pour
fin pour
```

FIG. 3.9 – Directives *order* qui contiennent des modules d'un espace S (B, C) et un module d'une projection S' de S (A). A droite, la génération automatique de YAO en boucle *for*.

une boucle ascendante sur l'axe i, cette boucle contient d'une part, l'instruction de la ligne 2 qui exécute la *fonction de base A* en tout point de l'axe i, et d'autre part l'instruction de la ligne 3 qui correspond à une autre boucle qui parcourt en sens décroissant l'axe j en exécutant les deux *fonctions de base B* et *C*. Ainsi, dans cet exemple, B et C sont définies sur un espace 2D S alors que A est définie sur un espace à 1D S', qui est une projection de S. L'axe i est commun pour les deux espaces.

3.7 Exemple numérique : le *Shallow-water*

Cette section présente les étapes nécessaires pour obtenir le graphe YAO du modèle direct, nous donnons un exemple de fichier de description. Nous choisissons le modèle du *Shallow-water* à deux dimensions dans le plan horizontal (x,y), également appelé modèle de Saint-Venant, qui découle de l'intégration verticale des équations de Navier-Stokes à trois dimensions. Ce modèle décrit le flux linéaire d'un fluide non visqueux en un environnement à eaux peu profondes et surface libre. L'évolution est décrite par le système d'équations aux dérivées partielles suivant :

$$\begin{aligned}\frac{\partial u}{\partial t} &= -g^* \cdot \frac{\partial h}{\partial x} + f \cdot v - \gamma \cdot u \\ \frac{\partial v}{\partial t} &= -g^* \cdot \frac{\partial h}{\partial y} - f \cdot u - \gamma \cdot v \\ \frac{\partial h}{\partial t} &= -H \cdot \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\end{aligned}$$

- u et v sont les vitesses horizontales sur les axes x, y.
- h est l'amplitude (la hauteur) de la surface libre.
- g^* est la gravité réduite.
- f est le paramètre de Coriolis.
- γ est un coefficient de dissipation.
- H est la hauteur moyenne de l'eau.

Le système d'équations aux dérivées partielles est résolu spatialement en utilisant la grille C d'Arakawa. Pour l'axe temporel, nous utilisons la discrétisation *saute mouton* suivie par le *filtre d'Asselin* pour assurer la stabilité. La discrétisation spatiale est basée sur une grille régulière 2D. Après l'initialisation, le modèle numérique direct est le suivant[9] :

- Variables dynamiques :

$$u_{ijt} = \hat{u}_{ijt-2} + 2\Delta t \left(\frac{-g^*}{\Delta x} \left[h_{i+1jt-1} - h_{ijt-1} \right] + \right. \tag{3.2}$$
$$\left. \frac{f}{4} \left[v_{ijt-1} + v_{ij+1t-1} + v_{i+1jt-1} + v_{i+1j+1t-1} \right] - \gamma \cdot \hat{u}_{ijt-2} \right)$$
$$v_{ijt} = \hat{v}_{ijt-2} + 2\Delta t \left(\frac{-g^*}{\Delta y} \left[h_{ijt-1} - h_{ij-1t-1} \right] - \right. \tag{3.3}$$
$$\left. \frac{f}{4} \left[u_{i-1j-1t-1} + u_{i-1jt-1} + u_{ij-1t-1} + u_{ijt-1} \right] - \gamma \cdot \hat{v}_{ijt-2} \right)$$
$$h_{ijt} = \hat{h}_{ijt-2} - 2\Delta t \cdot H \left(\frac{u_{ijt-1} - u_{i-1jt-1}}{\Delta x} + \frac{v_{ij+1t-1} - v_{ijt-1}}{\Delta y} \right). \tag{3.4}$$

- Filtre d'Asselin :

$$\hat{u}_{ijt} = u_{ijt} + \alpha(\hat{u}_{ijt-1} - 2u_{ijt} + u_{ijt+1}) \tag{3.5}$$
$$\hat{v}_{ijt} = v_{ijt} + \alpha(\hat{v}_{ijt-1} - 2v_{ijt} + v_{ijt+1}) \tag{3.6}$$
$$\hat{h}_{ijt} = h_{ijt} + \alpha(\hat{h}_{ijt-1} - 2h_{ijt} + h_{ijt+1}), \tag{3.7}$$

[9] Par commodité d'écriture, on utilise la notation F_{ijt}, qui correspond au module $F(i,j,t)$.

où $\hat{h}$, $\hat{u}$ et $\hat{v}$ sont des variables intermédiaires. Dans ce qui suit, nous montrons les directives du fichier de description YAO, qui définit les *connexions de base* relatives à (3.2)–(3.7) et permettent de générer le graphe modulaire *Shallow-water*. Nous supposons que le modèle est défini sur une grille 50×50 ($\Delta x = \Delta y = 5000$ mètres) et sur une trajectoire de 100 pas de temps ($\Delta t = 1500$ secondes, soit environ 1 jour et 17 heures). Dans cet exemple, les *fonctions de base Hphy*, *Uphy*, *Vphy*, *Hfil*, *Ufil*, *Vfil* représentent respectivement h, u, v, $\hat{h}$, $\hat{u}$, $\hat{v}$.

Dans la section 3.6.4, nous avons introduit les directives *ctin* et nous avons défini le délai temporel d_t négatif ou nul. Ceci est une contrainte naturelle puisque le transfert de données se fait du passé vers le futur. Le filtre d'Asselin garantit la stabilité numérique de notre modèle. Il introduit trois variables intermédiaires, qui font apparaître des dépendances avec un d_t positif. Ainsi, par exemple la *fonction de base Hfil* au temps t dépend de la *fonction de base Hphy* au temps $t+1$. Ceci n'est pas en contradiction avec la condition $d_t \leq 0$. En effet, étant donné que *Hfil*, *Ufil*, *Vfil* sont des variables intermédiaires qui ne relèvent d'aucun sens physique, leur calcul peut être décalé d'un pas de temps sans compromettre le calcul du modèle numérique. Ceci est illustré à la figure 3.10 sur la fonction *Hfil*, il en est de même pour les fonctions *Ufil* et *Vfil*.

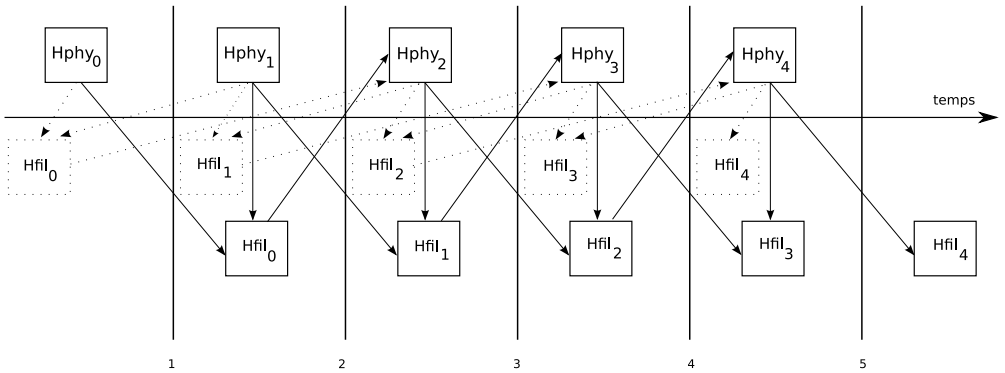

FIG. 3.10 – Décalage de la fonction *Hfil* d'un pas de temps sur la fenêtre temporelle. En pointillés, la figure montre la fonction *Hfil* au pas de temps original, en indice nous avons noté le pas de temps. Le même décalage concerne les fonctions *Ufil* et *Vfil*.

Le fichier de description est le suivant :

```
trajectory shallow_trajectory 100
space shallow_space 50 50 shallow_trajectory

module Hfil space shallow_space input 3 output 1 cost target
module Ufil space shallow_space input 3 output 1
module Vfil space shallow_space input 3 output 1
module Hphy space shallow_space input 5 output 1
module Uphy space shallow_space input 7 output 1
module Vphy space shallow_space input 7 output 1

ctin Hfil 1 from Hfil 1  i   j   t-1
ctin Hfil 2 from Hphy 1  i   j   t-1
```

```
ctin Hfil 3 from Hphy 1  i   j   t
ctin Ufil 1 from Ufil 1  i   j   t-1
ctin Ufil 2 from Uphy 1  i   j   t-1
ctin Ufil 3 from Uphy 1  i   j   t
ctin Vfil 1 from Vfil 1  i   j   t-1
ctin Vfil 2 from Vphy 1  i   j   t-1
ctin Vfil 3 from Vphy 1  i   j   t
ctin Hphy 1 from Hfil 1  i   j   t-1
ctin Hphy 2 from Uphy 1  i   j   t-1
ctin Hphy 3 from Uphy 1  i-1 j   t-1
ctin Hphy 4 from Vphy 1  i   j   t-1
ctin Hphy 5 from Vphy 1  i   j+1 t-1
ctin Uphy 1 from Ufil 1  i   j   t-1
ctin Uphy 2 from Hphy 1  i   j   t-1
ctin Uphy 3 from Hphy 1  i+1 j   t-1
ctin Uphy 4 from Vphy 1  i   j   t-1
ctin Uphy 5 from Vphy 1  i   j+1 t-1
ctin Uphy 6 from Vphy 1  i+1 j   t-1
ctin Uphy 7 from Vphy 1  i+1 j+1 t-1
ctin Vphy 1 from Vfil 1  i   j   t-1
ctin Vphy 2 from Hphy 1  i   j-1 t-1
ctin Vphy 3 from Hphy 1  i   j   t-1
ctin Vphy 4 from Uphy 1  i-1 j-1 t-1
ctin Vphy 5 from Uphy 1  i-1 j   t-1
ctin Vphy 6 from Uphy 1  i   j   t-1
ctin Vphy 7 from Uphy 1  i   j-1 t-1

  order YA1
      order YA2
          Hphy Uphy Vphy Hfil Ufil Vfil
```

La directive *trajectory* définit la trajectoire appelée *shallow_trajectory* composée d'un ensemble de 100 pas de temps dans lesquels le modèle tourne.

La directive *space* définit un espace appelé *shallow_space*. L'espace est défini avec une dimension de 50×50 ($N_i = 50$ et $N_j = 50$) et il est rattaché à une trajectoire *shallow_trajectory*.

Les directives *module* permettent de déclarer les *fonctions de base* F_p. Comme déjà dit, le mot-clé *module* est suivi par le nom de la *fonction de base* (par exemple *Hfil*) et puis le mot-clé *space* suivi par le nom de l'espace rattaché à la *fonction de base*. *input* et *output* permettent de spécifier le nombre d'entrées et sorties d'une *fonction de base*. La *fonction de base Hfil* a 3 entrées et 1 sortie. Le mot-clé *target* permet de contrôler les sorties de cette *fonction de base*, elle est la cible de notre processus d'assimilation. Le terme *cost* signifie que les sorties de cette *fonction de base* sont comparées à des observations et elles participeront au calcul de la fonction de coût. La mise en œuvre des *fonctions de base* se fait dans les fichiers module, comme montré à la page 44. Dans cet exemple les *fonctions de base Hphy*, *Uphy*, *Vphy*, *Hfil*, *Ufil*, *Vfil* réalisent les équations (3.2)–(3.7).

La directive *ctin* est utilisée pour créer les *connexions de base* du graphe modulaire. Le dernier *ctin* entre *Vphy* et *Uphy* permet de transférer la 1ère (et la seule) sortie de *Uphy* au point $(i, j-1, t-1)$ à la 7^e entrée de *Vphy* au point (i, j, t). Il s'agit de références relatives. En outre, le décalage des

fonctions *Hfil*, *Ufil* et *Vfil* permet leur calcul. A titre d'exemple, le 2^e et le 3^e *ctin* donnent en entrée à la fonction *Hfil* les sorties de *Hphy* respectivement aux pas de temps $t-1$ et t. Avec le décalage de *Hfil*, celle-ci correspondent au pas de temps t et $t+1$ de l'équation (3.7).

La directive *order* permet de coordonner le calcul des divers modules, c'est-à-dire de calculer un module seulement si toutes ses entrées provenant du module prédécesseur ont déjà été calculées. Les axes référencés par *YA1* pour *i* et *YA2* pour *j* sont fixés et traversés dans l'ordre mentionné. Ces directives *order* traversent l'espace dans un sens ascendant (sur *i* et *j*) et calculent les *fonctions de base* dans l'ordre spécifié. Ceci correspond à un tri topologique.

3.8 Implémentation des méthodes incrémentales et duales dans YAO

Lorsque le graphe modulaire d'une application est spécifié et généré par YAO, la mise en œuvre des différentes formules de calcul, qui, selon le modèle, peuvent être de complexité importante, est facilement implémentable dans YAO. Afin d'illustrer ceci, nous présentons sous des formes algorithmiques le calcul de l'adjoint et l'implémentation des méthodes incrémentale et duale.

Le calcul de l'adjoint de la formule (2.2), consiste à :
- Appliquer d'abord l'algorithme *forward*, qui permet de calculer tous les vecteurs d'entrées des modules du graphe, ainsi que les vecteurs de sorties $\mathbf{y}_i$ en chaque temps t_i et point de mesure.
- Calculer $\mathbf{B}^{-1}(\mathbf{x}(t_0)-\mathbf{x}^b)$.
- Calculer pour chaque temps t_i, le produit $\mathbf{m}_i = \mathbf{R}_i^{-1}(\mathbf{y}_i-\mathbf{y}_i^o)$.
- Initialiser pour chaque temps t_i, par les vecteurs $\mathbf{m}_i$, les points de mesure qui sont les *output data points* du graphe modulaire.
- Appliquer l'algorithme *backward* (algorithme 3), qui permet alors de calculer aux points d'entrées *input data point* le premier terme du gradient de la formule (2.2).
- Ajouter au calcul précédent le terme $\mathbf{B}^{-1}(\mathbf{x}(t_0)-\mathbf{x}^b)$ pour obtenir le gradient.

Les différents algorithmes *ward* de YAO permettent aussi une mise en œuvre simple de la méthode incrémentale. L'algorithme de la méthode incrémentale de la section 2.3 se présente, dans le formalisme de YAO, de la manière suivante :

Initialisation :
- Prendre $\mathbf{x}^g(t_0)=\mathbf{x}^b$ et $\boldsymbol{\delta}\mathbf{x}=0$

Boucle externe :
- Remplacer $\mathbf{x}^g(t_0)$ par $\mathbf{x}^g(t_0)+\boldsymbol{\delta}\mathbf{x}$.
- Initialiser les *input data points* du graphe par $\mathbf{x}^g(t_0)$ et appliquer l'algorithme *forward* qui permet de calculer $\mathbf{y}_i = H_i(M_i(\mathbf{x}^g(t_0)))$, puis calculer les vecteurs $\mathbf{d}_i = \mathbf{y}_i^o - \mathbf{y}_i$.
- Initialiser pour la boucle interne $\boldsymbol{\delta}\mathbf{x}=0$.

Boucle interne :
- Initialiser les entrées du graphe modulaire par le vecteur $\boldsymbol{\delta}\mathbf{x}$, puis appliquer l'algorithme *linward* afin de calculer pour chaque temps t_i les vecteurs $\boldsymbol{\delta}\mathbf{y}_i = \mathbf{H}_i\mathbf{M}_i(\mathbf{x}^g(t_0))\boldsymbol{\delta}\mathbf{x}$ et les vecteurs

$\mathbf{m}_i = \mathbf{H}_i\mathbf{M}_i(\mathbf{x}^g(t_0))\boldsymbol{\delta}\mathbf{x} - \mathbf{d}_i$ qui seront affectés aux points de mesure *output data points* du graphe.
- Appliquer l'algorithme *backward*, ce qui permet de calculer le premier terme de la formule (2.5).
- Calculer le gradient en ajoutant le second terme de (2.5).
- Corriger $\boldsymbol{\delta}\mathbf{x}$ avec une méthode de minimisation adéquate (la boucle interne calcule une nouvelle valeur de $\boldsymbol{\delta}\mathbf{x}$).

Il est possible aussi de présenter l'algorithme dual dans le formalisme de YAO en utilisant les notations de la section 2.4.2. Nous définissons pour cela deux vecteurs $\mathbf{p}(t_i)$ et $\boldsymbol{\delta}\mathbf{x}(t_i)$ qui sont définis pour tout point de l'espace de calcul. Ces deux vecteurs sont définis par les formules de récurrence :
- $\mathbf{p}(t_i) = \mathbf{M}^T(t_i)\mathbf{p}(t_i+1) + \mathbf{H}_i^T\mathbf{m}_i$ pour $i = 0,1,\ldots,n-1$.
- $\boldsymbol{\delta}\mathbf{x}_i = \mathbf{M}(t_{i-1})\boldsymbol{\delta}\mathbf{x}_{i-1} + \mathbf{Q}_i\mathbf{p}(t_i)$ pour $i = 0,1,\ldots,n$.

Initialisation :
- Prendre, pour tout temps $t_i = 0$, $\boldsymbol{\eta}_i$ ($\boldsymbol{\eta}_i$ étant un vecteur défini en tout point de la grille de calcul).
- Initialiser les points d'entrée du graphe (*input data points*) par le vecteur $\mathbf{x}^b$, appliquer la procédure *forward* afin de calculer $\mathbf{y}_i$ puis calculer $\mathbf{d}_i = \mathbf{y}_i^o - \mathbf{y}_i$.
- Faire quelques itérations de la méthode incrémentale, ce qui permet de calculer un vecteur $\boldsymbol{\delta}\mathbf{x}$.
- Initialiser les *input data points* du graphe modulaire par le vecteur $\boldsymbol{\delta}\mathbf{x}$. Appliquer la procédure *linward* afin de calculer $\boldsymbol{\delta}\mathbf{y}_i = \mathbf{H}_i\mathbf{M}(t_i)\ldots\mathbf{M}(t_0)\boldsymbol{\delta}\mathbf{x}$, calculer ensuite $\mathbf{m}_i = \mathbf{R}_{i-1}(\mathbf{y}_i^o - \boldsymbol{\delta}\mathbf{y}_i)$ (c'est une manière d'initialiser le vecteur $\mathbf{m}_i$ qui est défini en tout point d'observation).
- Prendre $\mathbf{p}(t_n) = 0$.

Itérer les étapes suivantes :
- Initialiser, pour tout temps t_i les points de sortie (*output data points*) par le vecteur $\mathbf{m}_i$, appliquer l'algorithme *backward* en conservant les valeurs intermédiaires des vecteurs $\boldsymbol{\alpha}_s$ (définis à l'algorithme 3) des différents modules. Ces vecteurs permettent de calculer les vecteurs $\mathbf{p}(t_i)$ pour $i = 0,1,\ldots,n-1$.
- Calculer le produit $\boldsymbol{\delta}\mathbf{x} = \mathbf{B}\mathbf{p}(t_0)$ (ce qui correspond à l'application de la formule (2.9)).
- Calculer, pour tout $i > 0$, le produit $\boldsymbol{\eta}_i = \mathbf{Q}_i^T\mathbf{p}(t_i)$ (formule (2.10)).
- En modifiant légèrement la procédure *linward*, on suppose que les entrées du graphe ne se limitent pas aux points de l'espace de calcul E_0 au temps t_0, mais s'étendent à tous les espaces de calcul E_i pour tout temps t_i. On initialise les *input data points* de la méthode *linward* de la manière suivante : les points de l'espace E_0 sont initialisés par $\boldsymbol{\delta}\mathbf{x}$, et les points de l'espace E_i sont initialisés par $\boldsymbol{\eta}_i$, puis on lance la procédure *linward*. Avec cette légère modification, la procédure *linward*, en chaque point de E_i, cumule la valeur courante $\boldsymbol{\eta}_i$ avec le résultat du calcul déjà réalisé en ce point lors de la propagation de l'algorithme. On note $\boldsymbol{\delta}\mathbf{x}_i$ le vecteur calculé par cette procédure sur les points de l'espace E_i.
- Calculer le gradient $\nabla_{\mathbf{m}_i}G = \mathbf{H}_i(\boldsymbol{\delta}\mathbf{x}_i) + \mathbf{R}_i\mathbf{m}_i - \mathbf{d}_i$ (ce qui correspond à l'application de la formule (2.12)).
- Modifier les paramètres $\mathbf{m}$ avec une méthode de minimisation adéquate.

3.9 Conclusion

Nous avons présenté dans ce chapitre le graphe modulaire, qui est la structure de base du logiciel YAO. Nous avons montré comment cette structure de graphe modulaire, qui correspond à un flot de calcul, permet de modéliser des fonctions complexes représentées par des modèles numériques discrétisés. Nous avons aussi montré que le modèle direct, le tangent linéaire et l'adjoint se calculent par des algorithmes de traversée du graphe modulaire et de son graphe inverse. Cet aspect algorithmique donne à YAO une flexibilité importante, il lui permet d'implémenter des méthodes comme la méthode incrémentale, pour la minimisation de la fonction de coût, et la méthode duale, pour la prise en compte des erreurs du modèle. Nous avons présenté, à la fin de ce chapitre, deux algorithmes permettant d'illustrer la façon d'implémenter, dans YAO, ces deux méthodes, qui manipulent des formules de calcul de complexités importantes. Ainsi, YAO constitue aussi une plateforme complète pour le lancement de sessions d'assimilation de données. D'autre part, il constitue un logiciel de simulation, puisqu'il met à la disposition de l'utilisateur les outils et les méthodes lui permettant de lancer plusieurs sessions d'assimilation de données, de tester les différentes méthodes de calcul des matrices de variance-covariance, de faire appel à des méthodes complexes, afin de tester l'apport quelles pourraient apporter à son problème.

Comme nous l'avons signalé, YAO n'est donc pas un différentiateur automatique du programme d'un modèle direct déjà écrit. En effet, quand il dispose d'un tel programme, l'utilisateur doit le transcrire dans le langage de description de YAO. Il doit donc analyser les différents calculs aux points de grilles, afin de pouvoir écrire les différentes directives *ctin*. Il doit aussi analyser les différentes boucles du programme, afin de pouvoir écrire les directives *order* qui leur correspondent. Parfois, quand il s'agit de modèles directs complexes, ces programmes pourraient contenir des anomalies, notamment au niveau des boucles de calculs, qui pourraient présenter des incohérences avec les dépendances des calculs, entre modules, en chaque point de grille. L'objectif du chapitre qui suit est de proposer un algorithme qui permet la détection des anomalies possibles entre les directives *ctin* et les directives *order*, afin que celles-ci puissent garantir la traversée complète du graphe modulaire et le lancement du calcul de tous les modules du graphe.

Chapitre 4

Cohérence dans l'ordre de calcul

4.1 Introduction

Les directives *ctin* et *order* sont à la base du langage de description YAO. Les *ctin* décrivent les dépendances dans le graphe modulaire et l'*order* génère les boucles imbriquées, qui permettent de traverser en même temps les espaces et d'appeler les *fonctions de base* dans un ordre particulier. Un espace peut être traversé de plusieurs façons et les directives *order* permettent à l'utilisateur de définir facilement une telle traversée. Parfois le choix de ce parcours s'avère assez difficile et, dans des applications YAO de grandes dimensions, l'utilisateur peut faire des erreurs dans la définition des directives *order*. Mal définir un parcours du graphe modulaire signifie que lorsqu'un module est calculé, ses entrées ne sont pas prêtes parce que ses prédécesseurs n'ont pas encore été tous calculés. La répercussion de ces erreurs affecte directement les résultats numériques du processus d'assimilation de données.

Une directive *ctin* est représentée par un couple de *fonctions de base* F_s (source) et F_d (destination ou cible). Si S' et S sont les espaces respectivement associés à F_s et F_d, nous notons L' l'ensemble des axes de l'espace S' et L celui de S [1] ; nous savons que S' est un sous-espace de S, dans le sens $L' \subset L$. Ainsi si (I,t) est une position courante de F_d, nous notons (I',t') la position correspondante de F_s. On note d le vecteur distance qui est défini par $d = I' - \hat{I}$, où $\hat{I}$ est la projection de I sur les axes de L'. Ainsi, d a la même dimension que I' et on note $d_l \in \mathbb{Z}$ sa composante suivant un axe l de L' et par $d_t = t' - t$ (≤ 0) le retard entre les pas de temps t' et t. Ces notations omettent le numéro de l'entrée de F_d et de la sortie de F_s, numéros qui sont spécifiés lors de l'écriture de la directive *ctin* (section 3.6.4). Vérifier que toutes les connexions, définies par les directives *ctin*, peuvent être correctement calculées est l'objectif de ce chapitre. Afin de vérifier la validité de la directive *order*, nous vérifions la cohérence de chaque directive *ctin* par rapport aux directives *order* définies par l'utilisateur. Cette vérification de cohérence n'a pas besoin du numéro

[1] Le vecteur itération I peut être défini à une ($I = (i)$), deux ($I = (i, j)$) ou trois dimensions ($I = (i, j, k)$) en fonction de l'espace. De même le vecteur I' qui est respectivement $I' = (i + d_i)$, $I' = (i + d_i, j + d_j)$ et $I' = (i + d_i, j + d_j, k + d_k)$ en fonction de l'espace S'. Le vecteur distance d aura une dimension qui correspond au nombre de composantes communes entre S et S'. Si par exemple S' est 2D et S est 3D, le vecteur distance sera 2D égal à (d_i, d_j).

d'entrée et de sortie, définis dans le *ctin*, ainsi la notation $F_s(I',t') \rightarrow F_d(I,t)$ est convenablement utilisée dans la suite. Le concept de cohérence d'une directive *ctin* est défini comme suit.

Définition 1. *Soit* $F_s(I',t') \rightarrow F_d(I,t)$ *une directive* ctin *où les* fonctions de base F_s *et* F_d *sont déclarées dans un même bloc de directives* order. *On dit que cette directive* ctin *est* ***cohérente*** *si, pour chaque* (I,t)*, les directives* order *assurent que la* fonction de base F_s *a déjà été calculée au point de grille* (I',t')*. Autrement, la connexion est dite* ***incohérente***[2].

La figure 4.1 montre un exemple de graphe modulaire composé de 4 *fonctions de base* (*A*, *B*, *C*, et *D*). Les flèches en pointillés représentent les connexions qui viennent de et qui vont vers l'extérieur de la grille. La figure 4.2a montre deux directives *order* imbriquées, qui rendent cohérentes chacune des connexions définies en figure 4.1. On peut vérifier que, pour chaque connexion, et pour chaque vecteur d'itération *I*, l'ordre de calcul donné par les boucles imbriquées est toujours satisfait. Dans le cas où la connexion vient de l'extérieur de la grille vers un module $F_d(I,t)$ (les flèches en pointillés de la figure 4.1), YAO considère que le module $F_d(I,t)$ est correctement alimenté par cette connexion. La cohérence de ce type de connexion est donc toujours vérifiée. La figure 4.2b, en revanche, montre deux directives *order* imbriquées, qui rendent au moins une connexion définie en figure 4.1 incohérente. La seule différence entre 4.2a et 4.2b est le sens de la seconde boucle qui est, dans le premier cas, ascendante et, dans le second, descendante. Cette différence rend la connexion $B(i,j-1,t) \rightarrow D(i,j,t)$ incohérente, c'est-à-dire que le module $D(1,2,t)$ est alimenté par $B(1,1,t)$ qui n'est pas encore calculé par cette traversée particulière. Le lecteur peut considérer la grille et la traversée imposée par les directives *order*, et ainsi immédiatement visualiser cette situation d'incohérence.

```
A(i-1, j, t) → A(i, j, t)
A(i+1, j, t) → B(i, j, t)
B(i, j-1, t) → D(i, j, t)
C(i, j-1, t) → C(i, j, t)
C(i, j+1, t) → D(i, j, t)
```

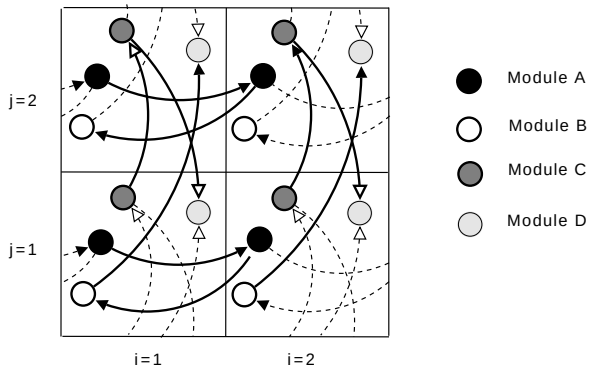

FIG. 4.1 – Graphe modulaire 2D (grille 2×2) avec 4 *fonctions de base* (*A*, *B*, *C* et *D*) et 5 connexions. Les flèches en pointillés représentent les connexions qui viennent de et qui vont vers l'extérieur de la grille.

Nous présentons dans ce chapitre les règles permettant de tester la cohérence d'une directive *ctin*. Le cas où les deux *fonctions de base* F_s et F_d sont calculées au même pas de temps ($t = t'$)

[2]L'abus de langage cohérence et incohérence des directives order est parfois utilisé dans ce texte.

```
    (a)

order YA1
    order YA2
        A C

order YA1
    order YA2
        B D
```

```
    (b)

order YA1
    order YA2
        A C

order YA1
    order YB2
        B D
```

FIG. 4.2 – (a) cohérence et (b) incohérence des directives *order* pour le graphe modulaire de la figure 4.1. La connexion incohérente relative aux *order* imbriqués est $B(i, j-1, t) \rightarrow D(i, j, t)$.

et par deux boucles extérieures différentes, relativement aux axes de L', représente le cas le plus simple. En effet, il suffit dans ce cas d'appliquer la règle suivante :

Règle 1. *Soit $F_s(I', t) \rightarrow F_d(I, t)$ ($t' = t$) un* ctin *entre les* fonctions de base *F_s et F_d. On suppose que F_s et F_d appartiennent à deux boucles extérieures différentes. Si la boucle extérieure contenant F_s est écrite avant celle qui contient F_d, alors la directive* ctin *est cohérente, sinon la directive* ctin *n'est pas cohérente.*

La vérification de la cohérence est plus délicate lorsque les deux *fonctions de base* F_s et F_d se présentent dans une même boucle extérieure. Dans la prochaine section, nous donnons deux règles, qui permettent de déterminer la cohérence d'une directive *ctin* par rapport à un ensemble de directives *order*. Dans la section 4.3, on présente un algorithme de vérification général. Nous nous limiterons, dans ces deux prochaines sections, aux cas des références relatives. Les références absolues seront traitées par la suite.

4.2 Règles de vérification : cas des références relatives

Règle 2. *Soit $F_s(I', t') \rightarrow F_d(I, t)$ une connexion entre deux* fonctions de base *contenues dans une boucle extérieure $l \in L' \cup \{t\}$, avec distance $d_l \neq 0$. Si $d_l < 0$ (respectivement $d_l > 0$) et la boucle l est ascendante (respectivement descendante), alors cette connexion est cohérente. De la même façon, si $d_l < 0$ (respectivement $d_l > 0$) et la boucle l est descendante (respectivement ascendante), alors cette connexion est incohérente.*

Justification. Supposons que la boucle l soit ascendante. Considérons un point I de l'espace S et supposons que l_I soit sa composante sur l'axe l. Considérons également un point I' de l'espace S' et supposons que $l_{I'}$ soit sa composante sur l'axe l. Si l est la boucle extérieure, alors, au moment du calcul de I, l'ensemble des points de grille avec $l_{I'} < l_I$ à déjà été calculé, grâce au sens ascendant de la boucle. En effet, lorsque les boucles imbriquées calculent l'itération l_I, toutes les instructions,

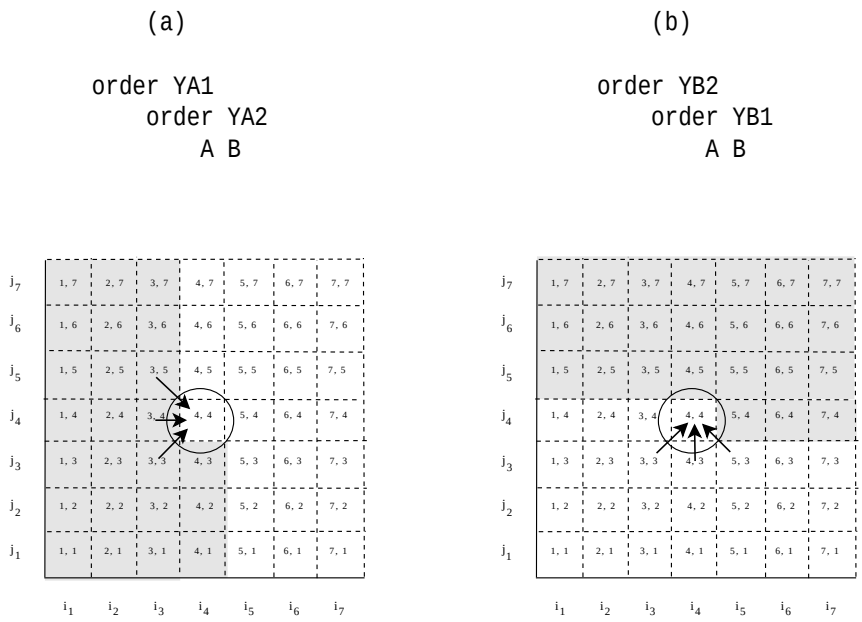

FIG. 4.3 – Parcours donné par deux directives *order* imbriquées et sa visualisation sur un espace de dimension 7×7. Le point de grille $I = (i, j) = (4,4)$ est le point de l'itération courante. La partie de l'espace en gris et celle en blanc représentent respectivement les points de grille calculés et pas encore calculés. Les flèches sont des connexions cohérentes pour (a) et des connexions incohérentes pour (b).

qui correspondent à un vecteur d'itération I', avec une composante $l_{I'}$ inférieure de l_I, ont déjà été calculées par la boucle l. Si la distance d_l est $d_l < 0$, alors le vecteur d'itération I' admet la composante $l_{I'} = l_I + d_l < l_I$, qui démontre la cohérence de la connexion. En conséquence, si $d_l > 0$, alors le module $F_s(I', t')$ n'a pas encore été calculé et la connexion est incohérente. Ce résultat est symétrique et ainsi valable aussi pour une boucle descendante.

La figure 4.3a illustre un parcours sur un espace 2D, où les *fonctions de base* A et B sont définies. Le point (i, j), encerclé dans la figure, se réfère au point de calcul courant. Les points de grille calculés dans les itérations précédentes sont colorés en gris. Le reste de la grille (en blanc) représente les itérations qui ne sont pas encore calculées. Les flèches représentent des connexions cohérentes par rapport à ces directives *order* imbriquées. Il s'agit des connexions $F_s(i-1, j+1, t) \rightarrow F_d(I,t)$, $F_s(i-1, j, t) \rightarrow F_d(I,t)$, $F_s(i-1, j-1, t) \rightarrow F_d(I,t)$. Les trois éléments qui assurent le calcul sont la boucle extérieure (l'axe i), le sens YA1 (ascendant) et le signe (-) de d_i, comme le résultat 2 le montre. A noter que F_s et F_d peuvent être indistinctement A ou B [3]. La figure 4.3b illustre un autre exemple de traversée 2D avec une boucle extérieure j descendante.

[3] $A(i-1, j+1, t) \rightarrow A(I,t)$, $B(i-1, j, t) \rightarrow A(I,t)$ et $B(i-1, j-1, t) \rightarrow B(I,t)$ sont aussi des connexions cohérentes.

Remarque : Ainsi, étant donné que la boucle la plus extérieure concerne la trajectoire temporelle, la règle 2 indique que si $d_t = t' - t < 0$ alors la directive *ctin* est cohérente quelles que soit les directives *order* relatives aux axes des espaces de calcul. C'est pourquoi dans la suite nous considérerons uniquement le cas où $t' = t$ ($d_t = 0$).

Règle 3. *Étant donnée une boucle extérieure relative à un axe $l \in \{i, j, k\}$ associé à une directive* order. *Supposons que $F_s(I', t) \rightarrow F_d(I, t)$ est une connexion entre deux* fonctions de base *contenues par cette boucle extérieure et avec une distance $d_l = 0$. Pour tester la cohérence, nous devons supprimer la boucle extérieure et garder le reste de ses instructions (boucles et* fonctions de base*). Nous pouvons avoir deux cas pour les instructions restantes :*

- *Les deux* fonctions de base *sont intégrées dans la même nouvelle boucle extérieure, alors nous appliquons la règle 2 ou 3 récursivement.*
- *Les deux* fonctions de base *sont dans deux instructions différentes, on applique dans ce cas, la règle 1 (l'instruction qui contient F_s doit précéder l'instruction qui contient F_d).*

Justification. Dans le cas $d_l = 0$ les *fonctions de base* F_s et F_d sont calculées lors d'une même itération de la boucle suivant l'axe l. Or cette boucle exécute une ou plusieurs instructions qui sont soit le calcul d'une *fonction de base*, soit l'exécution d'une boucle (imbriquée dans la première). Ainsi, si les *fonctions de base* F_s et F_d sont calculées par une même boucle imbriquée dans la première, alors il faut vérifier la cohérence relative à cette seconde boucle, pour cela il faut lui appliquer la règle 2 ou 3. Par contre, si les *fonctions de base* de type F_s et F_d sont calculées par deux instructions différentes de la liste des instructions de la boucle l, c'est la règle 1 qui s'applique (l'instruction qui contient la *fonction de base* F_s doit être exécutée avant celle qui contient la *fonction de base* F_d).

Exemple 1

```
A(i, j-1, t) → B(I, t)

  order YA1
      order YA2
          A B
```

On applique la règle 3 et, après avoir supprimé la boucle la plus extérieure t puis i (puisque $d_t = 0$ et $d_i = 0$), on a :

```
order YA2
    A B
```

Puisque $d_j < 0$ et la boucle est ascendante, la règle 2 assure que la connexion est cohérente.

Exemple 2

A(i, j+d_j, t) → B(I, t) avec d_j = -1 ou 0 ou +1

```
order YA1
    order YB2
        A
    order YB2
        B
```

On applique la règle 3 et après avoir supprimé la boucle la plus extérieure *t* puis *i* on a :

```
order YB2
    A
order YB2
    B
```

Puisque *A* et *B* sont contenues dans deux boucles extérieures différentes et que *A* précède *B*, la connexion est cohérente (règle 3). Pour le troisième exemple, la même connexion est incohérente :

```
order YA1
    order YB2
        B
    order YB2
        A
```

Remarque : Comme nous l'avons signalé, le programme généré par YAO contient toujours une boucle extérieure relative aux pas de temps (la trajectoire) ; à l'intérieur de cette boucle, les instructions permettent de calculer toutes les *fonctions de base* à un même pas de temps. On doit donc commencer par vérifier la cohérence par rapport à cette boucle extérieure (temps). Etant donné que cette boucle (temps) est toujours ascendante et que les délais d_t sont toujours ≤ 0 dans YAO, la vérification de la cohérence se réduit tout simplement à tester la valeur de d_t. Ainsi, si une directive *ctin* vérifie $d_t \neq 0$, alors elle est cohérente.

4.3 Algorithme de cohérence

Dans la section précédente, nous avons présenté les éléments qui caractérisent la cohérence d'une connexion *ctin* que nous avons notée $F_s(I',t) \rightarrow F_d(I,t)$. La boucle, le sens de la boucle, l'ordonnancement des *fonctions de base* et le vecteur distance *d* sont les acteurs du processus de vérification. La vérification de la cohérence globale des directives *order*, revient à tester la cohérence de chaque directive *ctin*. L'algorithme analyse chaque connexion indépendamment. D'autre part, en tenant compte de la remarque précédente, nous savons que toute directive *ctin* qui vérifient $d_t \neq 0$ est cohérente. C'est pourquoi, nous nous limitons à la vérification de cohérence des directives *ctin* qui vérifie $d_t = 0$. Par souci de simplification, ces directives seront donc représentées

dans la suite par la connexion générique $F_s(I') \rightarrow F_d(I)$, I' et I étant la localisation dans les espaces S' et S qui ne sont pas de même dimension. Etant donné que la dimension de S' est toujours inférieure ou égale à celle de S ($S' \subset S$), nous avons noté $\hat{I}$ la projection de I sur les axes de S'. Puisque nous considérons uniquement les références relatives, le vecteur distance $I' - \hat{I}$ a la même dimension que S' et sa composante suivant un axe l est égale à la distance algébrique d_l. Ainsi, si S' est de dimension 1, alors $I' - \hat{I} = (d_l)$, si elle est de dimension 2, alors $I' - \hat{I} = (d_l, d_m)$, où $l, m \in i, j, k$, et si elle est de dimension 3, alors $I' - \hat{I} = (d_i, d_j, d_k)$.

Pour introduire l'algorithme de cohérence, nous présentons en figure 4.4 un exemple de directives *order*. Dans cet exemple, l'utilisateur spécifie trois blocs de directives *order*. Chaque

```
order YA1
    A
    order YA2
        B C

order YA2
    order YA1
        D
        order YB3
            E
        order YA3
            F

order YB2
    order YB3
        order YA1
            G
```

FIG. 4.4 – Exemple de directives *order*, directives définies par l'utilisateur. Les *fonctions de base* E, F, G sont rattachées à des espaces à trois dimensions, B, C, D à des espaces à deux dimensions et A à un espace à une dimension.

bloc se réfère à un espace dans lequel les *fonctions de base* associées à cet espace peuvent être calculées. Ainsi, cette séquence de blocs *order* divise l'ensemble total des *fonctions de base* en sous-ensembles disjoints. L'ordre de calcul des blocs est donné par l'ordre de déclarations de l'utilisateur. Dans l'exemple, le calcul de la procédure *forward* commence par le calcul du bloc contenant *A*, *B* et *C* ; ensuite le bloc qui contient les *fonctions de base D*, *E* et *F* est calculé ; enfin le bloc de la *fonction de base G* termine le calcul de la procédure. Chaque bloc de directives *order* est composé par une boucle extérieure, décrite par le paramètre YXl, avec $X \in \{A, B\}$ et $l \in \{1, 2, 3\}$ qui correspond à $\{i, j, k\}$. Le corps de la boucle extérieure est composé de trois types de listes d'instructions :

- une liste de boucles,
- une liste de *fonctions de base*,
- une liste composée d'un mélange de boucles et de *fonctions de base*.

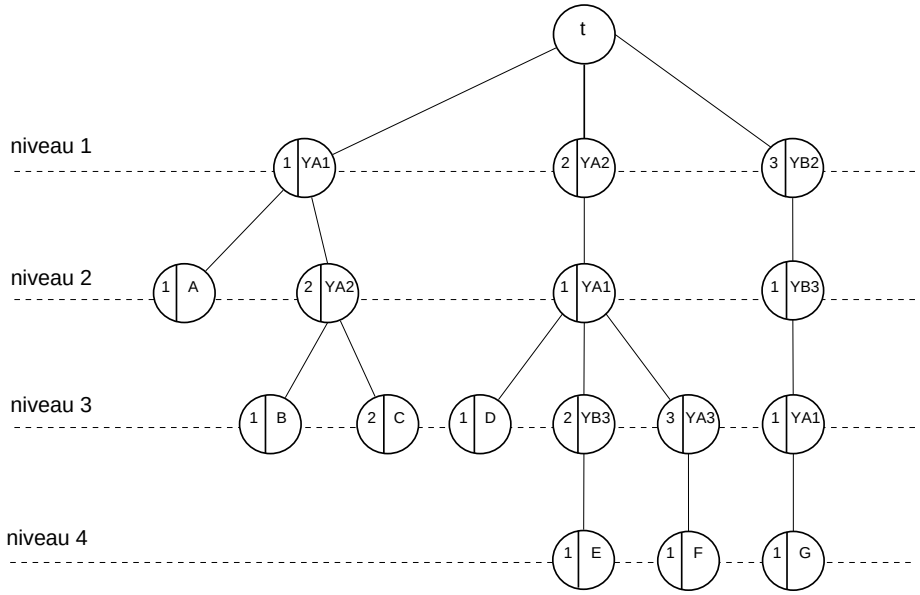

FIG. 4.5 – Cette arborescence correspond aux directives *order* de la figure 4.4. Les feuilles représentent les *fonctions de base* et tous les autres nœuds (les nœuds internes) représentent les boucles, ils sont caractérisés par leurs paramètres YXl. Tous les nœuds, sauf la racine, se caractérisent par num_fils.

Comme dans la théorie de la compilation [Aho *et al.*, 2006], il est possible d'organiser des directives *order* par un *Abstract Syntax Tree* (AST). On appellera cet AST *arbre de directives order*. Les nœuds fils de la racine sont les *order* extérieurs (trois dans l'exemple) ; ces nœuds correspondent au niveau 1 (la racine étant au niveau 0). Le développement de l'arbre continue à partir de ces nœuds fils. D'une manière générale, chaque nœud de l'arborescence correspond à une instruction. Cette instruction peut être soit une boucle relative à une directive *order*, soit le calcul d'une *fonction de base*. Un nœud qui calcule une *fonction de base* n'a pas de fils et correspond donc à une feuille de l'arbre. Un nœud qui correspond à une boucle aura autant de fils que d'instructions dans sa boucle, ces fils seront situés dans le niveau consécutif au niveau du père (niveau du père +1).

Dans cette représentation, les *fonctions de base* sont les feuilles de l'arbre, et chaque nœud interne (nœud qui n'est pas une feuille ou la racine) représente une boucle définie par son axe et son sens. Puisque la version actuelle de YAO ne permet qu'un maximum de trois dimensions pour un espace, l'arborescence a au plus 4 niveaux, étant donné que la racine est le niveau 0. Les fils d'un nœud sont numérotés (à partir de 1) dans l'ordre de leur déclaration par l'utilisateur. Nous noterons par la suite ce numéro num_fils. Un nœud interne de l'arborescence contient le paramètre YXl ($X \in \{A,B\}$ et $l \in \{1,2,3\}$), qui spécifie l'axe et le sens de la boucle. Une feuille contient le nom de la *fonction de base*. La figure 4.5 montre l'arborescence relative à l'exemple précédent (figure 4.4).

Avec cette représentation, si nous voulons caractériser les boucles imbriquées qui contiennent le calcul d'une *fonction de base*, il suffit de déterminer le chemin de la racine à la feuille qui représente la *fonction de base*. Les nœuds internes de ce chemin représentent les boucles imbriquées qui permettent le calcul de la *fonction de base*. Si nous ne considérons pas la racine, le premier nœud du chemin correspond à la boucle extérieure et le dernier nœud correspond à la *fonction de base*. Ainsi, pour chaque *fonction de base*, nous pouvons créer une liste qui représente le chemin, composé au plus de 3 nœuds internes intermédiaires. Chaque nœud interne contient les champs suivants :

- *num_fils*, ordre de gauche à droite de la déclaration des frères à l'égard du nœud parent,
- *axe*, indice de la boucle ($axe \in \{i, j, k\}$),
- *sens*, ascendant ou descendant de la boucle.

L'*axe* et le *sens* sont représentés dans la figure par le paramètre YXl. Avec cette arborescence, on peut vérifier la cohérence d'une directive *ctin* particulière. L'algorithme 4 explique le processus de cohérence.

Nous nous limitons ici à l'explication de l'idée générale en présentant quelques exemples d'exécution de l'algorithme de vérification. L'objectif de l'algorithme de cohérence est de détecter les incohérences sur les directives *order*, en supposant que les directives *ctin* sont globalement cohérentes. Nous allons discuter de l'incohérence des directives *ctin* au chapitre suivant.

À titre d'exemple, on montre une exécution de l'algorithme. Nous testons la cohérence de la connexion $B(i-1, j) \rightarrow C(i, j)$, où $d_i = -1$ et $d_j = 0$, sur l'arbre de figure 4.5. La figure 4.6a montre les chemins C_b et C_c pour les *fonctions de base B* et *C*, ils ont trois niveaux, $n = 3$. A la

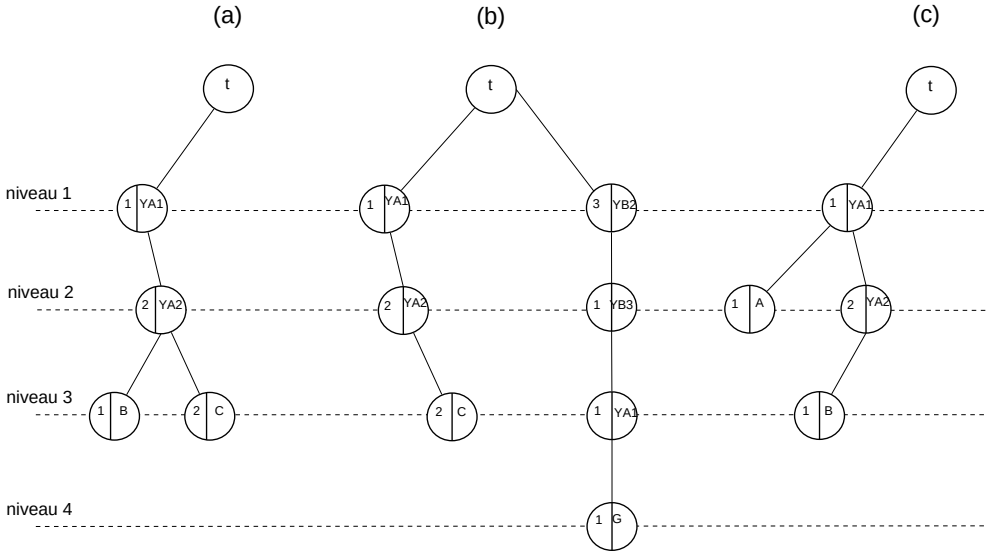

FIG. 4.6 – Trois exemples de chemins C_s et C_d pour l'arbre de la figure 4.5 et les *fonctions de base* : (a) B et C ; (b) C et G ; (c) A et B.

première itération $m = 1$, le *num_fils*, l'*axe* et le sens du nœud de N_s et N_d sont 1 et YA1. Les

Algorithme 4 Vérification de la cohérence des directives *order* par rapport à une directive *ctin*.

ENTRÉES: On note $F_s(I',t') \rightarrow F_d(I,t)$ la connexion qui représente la directive *ctin*. d_l est la composante du vecteur $I' - \hat{I}$ relativement à l'axe l.

SORTIES: *vrai* si le *ctin* est cohérent, *faux* autrement.

1: **si** $d_t \leq 0$ **alors**
2: **retourner vrai**
3: **finsi**
4: Trouver les deux chemins C_s et C_d de la racine aux feuilles F_s et F_d.
5: Soit n la longueur minimale de C_s et C_d.
6: **pour** $m = 1$ **à** n **faire**
7: Déterminer au niveau m les deux nœuds N_s et N_d de l'arbre qui sont situés respectivement sur les deux chemins C_s et C_d.
8: //Les deux nœuds N_s et N_d sont soit confondus soit deux frères.
9: **si** num_fils de $N_s <$ num_fils de N_d **alors**
10: **retourner vrai**
11: **finsi**
12: **si** num_fils de $N_s >$ num_fils de N_d **alors**
13: **retourner faux**
14: **finsi**
15: //A ce stade, num_fils de $N_s =$ num_fils de N_d. Les deux nœuds sont identiques et correspondent à une boucle.
16: Soit l l'axe relatif à la boucle commune.
17: **si** $d_l \neq 0$ **alors**
18: **retourner** le résultat de la règle 2.
19: **finsi**
20: // On continue la boucle au niveau successif ce qui correspond à l'application de la règle 3.
21: $m \leftarrow m+1$
22: **fin pour**
23: **retourner faux**

conditions des lignes 9 et 12 de l'algorithme 4 sont *faux*, nous sommes dans la même boucle. Puisque $sens = ascendant$ et $d_i < 0$, la règle 2 de la ligne 18 a comme résultat que le *ctin* est cohérent, donc l'algorithme se termine en retournant *vrai*.

En deuxième exemple (figure 4.6b), nous testons la cohérence de la connexion $C(i+d_i, j+d_j) \rightarrow G(i,j)$, où d_i et d_j ont une valeur quelconque. C_s et C_d, montrés en figure 4.6b, ont respectivement 3 et 4 niveaux. A la première itération $m = 1$, le num_fils de N_s et N_d ont respectivement les valeurs 1 et 3, ce qui rend vraie la condition à la ligne 9. En effet, C et G ont respectivement i et j comme boucles extérieures, les *fonctions de base* n'appartiennent pas à la même boucle de niveau 1. Puisque le num_fils de N_s est inférieur au num_fils de N_d, l'algorithme retourne *vrai* (il est évident que si l'on cherchait à vérifier la connexion $G(i+d_i, j+d_j) \rightarrow C(i,j)$ l'algorithme renverrait la réponse *faux*).

Un dernier exemple (figure 4.6c) montre l'execution de l'algorithme sur la connexion $A(i) \rightarrow B(i,j)$. C'est le cas du transfert de données entre les espaces qui ont une relation de projection : A et B sont des *fonctions de base* rattachées respectivement à des espaces 1D et 2D. C_s et C_d ont respectivement 2 et 3 niveaux (figure 4.6c). A l'itération $m = 1$, les *fonctions de base* sont dans la même boucle (les conditions des lignes 9 et 12 sont fausses) et, puisque $d_i = 0$, m est incrémenté. A l'itération $m = 2$, la condition à la ligne 9 est *vrai*, car nous avons, d'une part, le nœud de la *fonction de base A* et, d'autre part, la boucle j contenant B. La seule chose à tester est la précédence de A par rapport à B, ce qui est donné par le num_fils de N_s et N_d. L'algorithme retourne *vrai*.

Remarque : Notons que la boucle *for*, dans cet algorithme, se termine quand $I' - I = 0$, $t = t'$ et $F_s = F_d$ (ceci correspond à une connexion $F_s(I,t) \rightarrow F_s(I,t)$). Cette situation représente une anomalie dans la définition de cette directive *ctin* particulière, puisque cette connexion n'est jamais calculable. L'algorithme retourne *faux*, mais ce résultat ne signifie pas que des directives *order* incohérentes soient détectées. Ce cas particulier ne figure pas dans la pratique, puisque YAO n'accepte pas ce type de directive *ctin*. On discutera de ce type d'anomalie dans le chapitre suivant.

4.4 Références absolues

Dans la section 3.6.4, nous avons introduit les *références absolues*. Nous faisons maintenant une analyse plus détaillée de ce type de transfert de données. La section précédente traite la vérification de la cohérence des directives *order* pour les connexions qui ont des références relatives, le vecteur d'itération I' de la source est relatif par rapport au point de grille générique de destination I. Dans ce cas, la distance entre deux points de grille est donnée par le vecteur distance d, à 1, 2 ou 3 dimensions en fonction de la dimension du vecteur I'. Le processus de vérification de la cohérence doit vérifier également les références absolues, qui sont parfois utilisées dans la pratique. Afin de mieux comprendre ce genre de références, on considère la connexion $F_s(abs_i) \rightarrow F_d(I)$, où F_s et F_d sont des *fonctions de base* rattachées à deux espaces 1D ; abs_i est une valeur constante, $abs_i \in \mathbb{N}$ et $1 \leq abs_i \leq N_i$. Cette connexion signifie que les entrées pour le calcul de $F_d(i)$ sont toujours alimentées par le point de grille (abs_i) de la *fonction de base* F_s. Comme cas particulier de abs_i, on a $abs_i = 1$ et $abs_i = N_i$, respectivement en figures 4.7a et 4.7b, où la référence absolue correspond à un bord de la grille. Ce sont, en effet, les cas d'utilisation des références absolues les plus fréquents. La valeur $1 < abs_i < N_i$, est un cas particulier que nous décrivons à la fin de cette section. Pour l'instant nous limitons l'analyse à des valeurs constantes d'abs_i correspondantes à un bord de la grille.

Les références absolues sont utilisées principalement pour l'initialisation d'une *fonction de base* tout au long de son espace S (ou bien tout au long d'un sous-espace affine de S dans le cas pluridimensionnel). Si les *fonctions de base* F_s et F_d ne sont pas dans la même boucle, la règle 1 des références relatives peut être également utilisée. Il est clair que si les *fonctions de base* sont calculées séparément il suffit que F_s soit calculée avant F_d dans les deux exemples des figures 4.7a et 4.7b.

Dans la figure 4.7a, le calcul se fait dans le sens ascendant. Si F_s et F_d sont dans la même

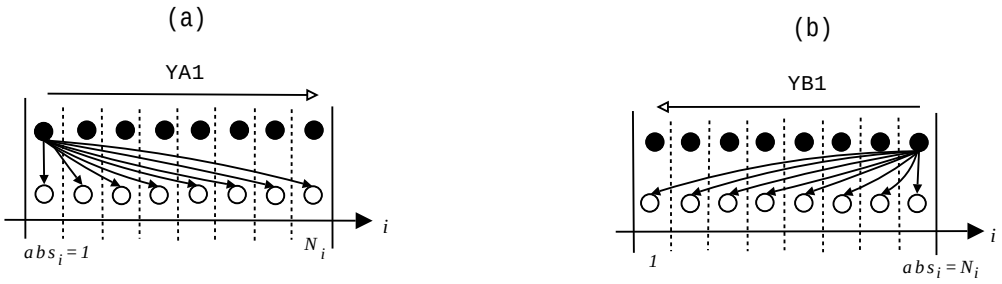

FIG. 4.7 – Connexion $F_s(abs_i) \rightarrow F_d(I)$ entre deux *fonctions de base* rattachées à deux espaces 1D avec une référence absolue abs_i. Les cercles en noir et blanc représentent respectivement les modules source et destination. (a) $abs_i = 1$ et (b) $abs_i = N_i$.

boucle et F_s précède F_d, pour que le calcul soit correct il suffit que la directive *order* soit définie ascendante : $F_s(1)$ est calculé d'abord, puis sa valeur est propagée à tous les points de grille de F_d. Si la directive *order* était descendante, il n'y aurait pas moyen de calculer correctement F_d. La situation est la même, mais inversée, dans la figure 4.7b. Dans ce cas, il est important que le calcul ait un sens descendant. Cette première considération montre qu'il est facile, pour des *fonctions de base* rattachées à un espace 1D, de tester la cohérence d'une connexion avec une référence absolue lorsque abs_i est sur le bord de la grille. Pour $abs_i = 1$, le sens de l'axe doit être ascendant et pour $abs_i = N_i$ le sens doit être descendant.

Une connexion peut être formée par un mélange de références absolues et relatives. Cette situation est montrée en figure 4.8. La connexion de la figure 4.8a est cohérente par rapport aux traversées de la figure 4.9. Les directives *order* de 4.9c sont cohérentes puisqu'on peut appliquer directement la règle 1. Le lecteur peut vérifier que pour 4.9a et 4.9b le calcul se déroule correctement. En effet, la référence relative de l'axe j permet plusieurs traversées possibles. Dans l'exemple 4.8a, tous les points de grille $(1, j-1)$ sont utilisés comme initialisation pour le sous-espace affine donné par la droite j. Le même genre d'initialisation est présent dans la figure 4.8b, où tous les points de grille $(N_i, j+1)$ nourrissent le sous-espace affine donné par la droite j. La figure 4.8c montre un exemple d'initialisation où l'ensemble des points $(i-1,1)$ nourrissent le sous-espace affine donné par la droite i. Ces trois exemples nous permettent d'étendre les considérations précédentes.

Règle 4. *Étant donnée une boucle extérieure $l \in \{i, j, k\}$ associée à une directive* order. *Soit $F_s(I') \rightarrow F_d(I)$ une connexion, où I' contient la référence absolue $abs_l = 1$ ou $abs_l = N_l$, et les* fonctions de base *$F_s(I')$ et $F_d(I)$ sont contenues dans la boucle l. La connexion est incohérente si $abs_l = 1$ et si l est descendant ou si $abs_l = N_l$ et si l est ascendant.*

Pour comprendre ce résultat, on considère la connexion de la figure 4.8a avec les directives *order* :

```
order YB1
```

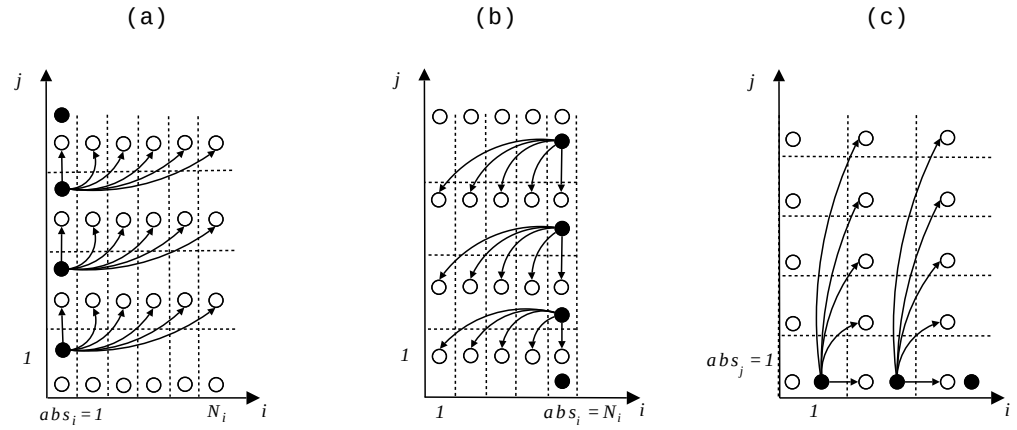

FIG. 4.8 – Mélange de références absolues et relatives dans le cadre de *fonctions de base* rattachées aux espaces 2D S' et S. Les cercles en noir et blanc représentent respectivement les modules source et destination. Avec une référence absolue, nous faisons une projection de l'espace S associé à la *fonction de base* destination. Cette projection identifie un sous-espace affine S'. Dans chacune des trois figures (a), (b) et (c), on représente le transfert de données nécessaires au calcul de F_d : (a) $F_s(abs_i, -1) \rightarrow F_d(i, j)$ avec $abs_i = 1$, (b) $F_s(abs_i, +1) \rightarrow F_d(i, j)$ avec $abs_i = N_i$, (c) $F_s(-1, abs_j) \rightarrow F_d(i, j)$ avec $abs_j = 1$. Dans ces figures, on montre seulement les connexions relatives à ce transfert, les modules isolés peuvent être alimentés par d'autres connexions. Sinon les modules isolés pourraient représenter des cas particuliers que l'utilisateur gérerait directement au moment de la mise en œuvre des *fonctions de base* dans les fichiers module (section 3.5.3).

```
order YX2
    Fs  Fd
```

La directive *order* extérieure représente l'axe i, il est descendant ($YB1$). La règle 4 dit que la connexion est incohérente. En effet, si l'on commence à calculer à partir du haut de la grille pour l'axe i (N_i), le module $F_s(1, j)$ nécessaire pour F_d ne sera pas encore calculé (j est soit 1 soit $N_j - 1$ respectivement si $YX2$ est $YA2$ ou $YB2$).

Pour la gestion des références absolues, nous pouvons améliorer l'algorithme 4 de la section 4.3. Les références absolues sont traitées comme un cas particulier. Lorsque l'algorithme reconnaît une référence absolue, il doit appliquer la règle 4 pour la détection des incohérences. S'il n'y a pas d'incohérence au niveau m, des chemins C_s et C_d, ce niveau est incrémenté et l'analyse du nœud suivant démarre. Donc, la référence absolue arrête l'exécution de l'algorithme dans le cas d'incohérence et sinon l'algorithme continue à l'axe suivant, s'il existe. Nous reprenons l'algorithme 4 de cohérence sous forme d'organigramme de programmation de la figure 4.10, en ajoutant les considérations faites sur les références absolues.

L'exemple 4.8a présente une référence absolue ($abs = 1$) sur son axe i. Si l'on considère les directives *order* de 4.9a et 4.9b, l'axe i est ascendant. Le fait d'avoir la boucle extérieure ascendante et l'$abs_i = 1$, n'est pas suffisant pour dire que la connexion est cohérente, en fait, l'axe j ne peut

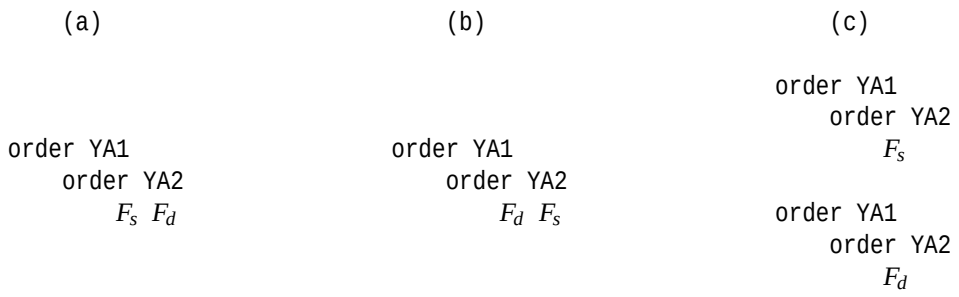

FIG. 4.9 – Directives *order* cohérentes par rapport à la connexion de la figure 4.8a. Le fait d'avoir la référence relative $j-1$ rend correctes les deux solutions (a) et (b). (c) est correct grâce à l'application de la règle 1.

pas être descendant. Le test doit se poursuivre sur le second axe.

Le cas général de référence absolue pour *fonctions de base* rattachées à des espaces 1D est illustré à la figure 4.11. Nous avons traité les cas spécifiques $abs_i \in \{1, N_i\}$, le cas $1 < abs_i < N_i$ introduit des difficultés supplémentaires, car il n'est jamais possible de calculer correctement tous les modules. Toutefois, le processus de vérification ne devrait pas générer une erreur comme pour la référence relative. En fait, l'utilisateur peut gérer lui même certains cas spécifiques à l'intérieur des *fonctions de base* dans les fichiers module. Ainsi, si l'on considère la référence absolue $F_s(2) \rightarrow F_d(i)$, définie sur une boucle i ascendante, à l'itération $i = 1$ le module $F_s(2)$ n'est pas encore calculé, ce qui ne permet pas, a priori, d'alimenter correctement le module $F_d(1)$. Mais l'utilisateur peut, lors de la définition de la *fonction de base* dans le fichier module, définir la valeur de $F_s(2)$, ce qui rend le calcul faisable. Il peut gérer, à l'intérieur de la *fonction de base* F_d, l'entrée à l'itération $i = 1$ comme un cas à part. Une instruction conditionnelle pourrait tester la variable d'indice i et ne considérer les entrées de la *fonction de base* que dans le cas où elles ont déjà été calculées. Ces informations, qui sont exprimées lors de la définition des méthodes *forward* et *backward* d'un modèle, ne sont pas accessibles directement à YAO. Ce type de connexions engendre une incertitude vis-à-vis de l'algorithme de vérification de la cohérence, c'est pourquoi cette situation ne peut pas être considérée, a priori, comme incohérente. Il s'agit d'une connexion particulière et l'algorithme de vérification doit afficher un avertissement qui alerte l'utilisateur, mais n'arrête pas le processus de vérification global. L'algorithme de vérification pourrait tester toutes les connexions et donner, en dernier lieu, un rapport sur l'ensemble. Ceci est illustré dans l'organigramme avec le symbole *warning* et la condition $abs \notin \{1, N\}$.

4.5 Résultats et conclusion

Dans ce chapitre, nous avons traité le problème de la cohérence des directives *ctin* avec des directives *order* prédéfinies par l'utilisateur. Il s'agit de vérifier que chaque directive *ctin* peut être correctement calculée en utilisant la traversée du graphe définie par l'utilisateur. L'algorithme de cohérence analyse un *ctin* et retourne *vrai* si les directives *order* sont cohérentes vis-à-vis de ce

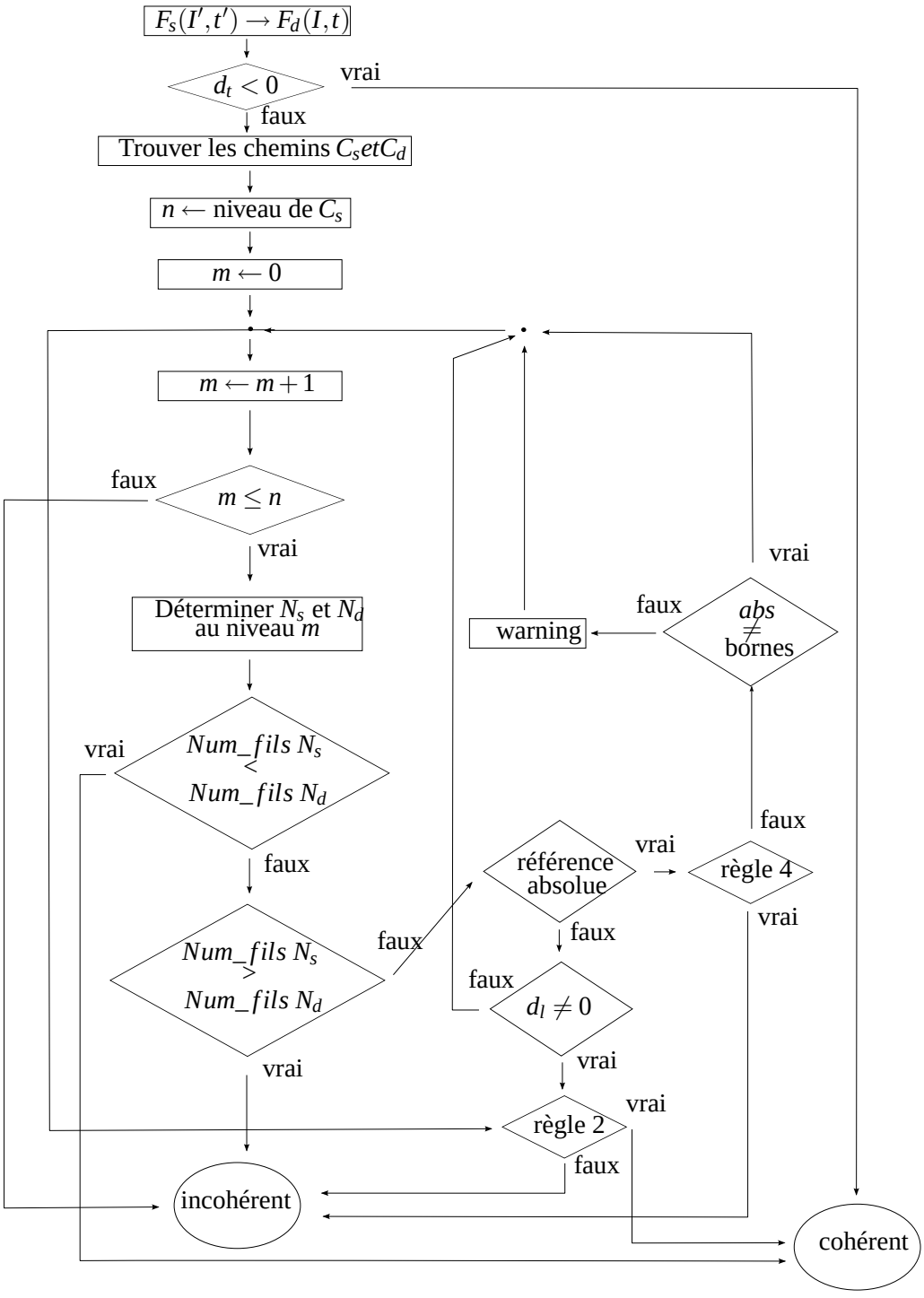

FIG. 4.10 – Organigramme de programmation de l'algorithme de vérification de la cohérence des directives *order*. Ceci est le cas le plus général applicable à toutes sortes de connexions *ctin* (avec références relatives, absolues ou un mélange des deux).

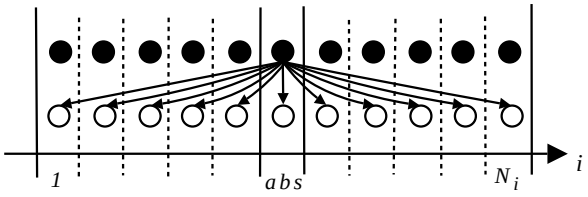

FIG. 4.11 – Connexion $F_s(abs_i) \rightarrow F_d(i)$ entre deux *fonctions de base* rattachées à deux espaces 1D avec une référence absolue abs_i. Les cercles en noir et blanc représentent respectivement les modules source et destination. La valeur de la référence absolue est $abs_i = abs$ avec $1 < abs < N_i$.

ctin, *faux* sinon. Cet algorithme est appliqué à chaque *ctin* defini par l'utilisateur. Pour un graphe modulaire donné, si toutes les valeurs retournées sont *vrai*, on peut alors supposer que les directives *order* sont cohérentes. Nous avons testé l'algorithme sur des applications réelles, ainsi que sur des applications simulées. Le test sur l'une des applications réelles a conduit à la détection d'une paire d'incohérences qui n'a jamais été détectée à l'œil nu.

Le temps de calcul de l'algorithme, qui tourne sur des centaines de *ctin*, est très petit (de l'ordre de quelques dixièmes de seconde), ce qui montre qu'il peut être appliqué à toutes les applications réelles.

Dans la conception d'une nouvelle application YAO, l'écriture des directives *order* est une phase coûteuse en temps et susceptible de générer des erreurs. En outre, c'est une tâche délicate, car les directives *order* indiquent comment le graphe modulaire est traversé et ceci a un impact direct sur le temps de calcul de l'application. En général, l'aspect performance n'est pas négligeable dans les applications d'assimilation variationnelle de données. Ces problématiques sont à la base de la suite des travaux de cette thèse. L'algorithme de cohérence permet d'éviter les erreurs lors de l'écriture des directives *order* par un utilisateur. Le chapitre qui suit traite le problème de la génération automatique de directives *order* cohérentes. Ce point est important, car la génération automatique peut générer toutes les directives *order* possibles, ce qui permet par la suite de faire un choix particulier qui optimise un critère donné.

Chapitre 5

Génération automatique des directives *order*

Les directives *order* définies par l'utilisateur permettent le calcul des modules du graphe modulaire dans un ordre topologique. En général, plusieurs ordres topologiques existent dans un graphe orienté acyclique (DAG[1]). Dans le formalisme YAO, l'utilisateur peut spécifier une traversée particulière d'un espace de calcul. Cette traversée est exprimée à l'aide de boucles imbriquées relativement aux axes de l'espace de calcul, par le biais des directives *order*. Ce chapitre propose une méthode pour la génération automatique de ces directives *order*. Une profonde compréhension de la structure du graphe, donnée par les directives *ctin*, permet d'extraire des informations utiles afin de proposer automatiquement des directives *order* cohérentes et donc de permettre le calcul des modules selon un ordre topologique.

Dans le chapitre précédent, l'algorithme de vérification de la cohérence valide ou invalide les directives *order* définies par l'utilisateur. L'invalidation est suivie par la notification de la connexion, qui n'est pas calculable par ces directives *order*. L'utilisateur peut modifier les directives *order* et exécuter à nouveau l'algorithme de vérification. Parfois, des incohérences sont dues aux directives *ctin* ; dans ce cas, soit le graphe modulaire contient des circuits, soit il est acyclique, mais ne peut pas être traversé par un ordre topologique sous la forme de boucles imbriquées. L'utilisateur peut itérer plusieurs fois le processus de modification des directives *order*, mais ne reçoit qu'une validation ou une invalidation de l'algorithme de vérification de la cohérence. En effet, l'algorithme présenté au chapitre précédent retourne *faux* sans détecter l'origine du problème, qui peut découler des incohérences dans la définition des directives *ctin*. Dans la section 4.3, nous avons déjà parlé de cette situation pour la connexion $F_p(i,j,k) \rightarrow F_p(i,j,k)$. Cette connexion signifie que pour calculer $F_p(i,j,k)$, le module $F_p(i,j,k)$ a déjà été calculé, ce qui n'a aucun sens. Puisque l'objectif de ce chapitre est de générer automatiquement les directives *order*, nous devons détecter les problèmes qui résultent d'une mauvaise écriture des *ctin* qui empêchent la génération d'une traversée valide par des directives *order*.

[1]DAG : *Direct Acyclic Graph.*

Etant donné qu'il existe plusieurs ensembles de directives *order* possibles permettant d'assurer une traversée du graphe modulaire, il faut donc en choisir une particulière et la proposer à l'utilisateur. Cette détermination ne peut se faire que si l'on définit un critère permettant ce choix. Ceci peut par exemple se faire en définissant une heuristique de choix particulière qui réduirait au mieux le temps de calcul. Nous n'abordons pas, dans ce chapitre, la question du choix d'une traversée "optimale". Par contre, nous présentons la méthodologie et un algorithme générique de base qui permet la génération de l'ensamble des directives *order*.

En général, si l'on traite d'une application réelle, il n'est pas possible d'écrire en un seul bloc[2] l'ensemble des directives *order* contenant toutes les *fonctions de base* qui définissent la traversée. Certaines connexions peuvent avoir besoin d'un sens pour un axe et d'autres connexions le sens opposé pour le même axe, de qui rend impossible l'utilisation d'une même boucle. En outre, une connexion, pour être satisfaite, pourrait avoir besoin de traverser d'abord l'axe i, tandis qu'une autre connexion pourrait avoir besoin de traverser au préalable l'axe j. Par conséquent, généralement, le processus génère plusieurs blocs de directives *order*, chacun contenant un certain nombre de *fonctions de base* F_p. La génération de chaque bloc se caractérise par :

1. le choix d'un ensemble de *fonctions de base* et l'ordre dans lequel ces fonctions sont exécutées ;
2. le choix d'une traversée de l'espace. Concrètement, cela conduit au choix des axes (par exemple i est le premier axe, j est le deuxième et k est le troisième, ou une combinaison quelconque des trois axes) et le choix des sens suivant chaque axe (ascendant ou descendant). Chaque bloc sera identifié par l'axe relatif à sa boucle extérieure.

Une directive *ctin* génère une *connexion de base* du graphe modulaire, définie par le point de grille générique (I,t) de la *fonction de base* destination F_d et le point de grille (I',t') de la *fonction de base* source F_s. Dans le chapitre précédent, nous avons dit que la boucle extérieure du temps agit comme une barrière de calcul, qui garantit que les modules à $t+d_t$ avec $d_t < 0$ ont déjà été calculés. Cette considération vaut également pour la génération automatique des directives *order*. Une connexion avec $d_t < 0$ ne contient aucune information sur la façon dont le parcours doit être fait, puisqu'elle n'ajoute aucune contrainte. Ainsi, dans la suite du chapitre, nous ne considérerons pas ce genre de connexions, mais uniquement les connexions avec $d_t = 0$. Lorsque nous notons $F_s(I') \rightarrow F_d(I)$, sans mentionner le temps, cela signifie que nous considérons que d_t est implicitement nul. Dans cette première partie du chapitre, nous supposons que les *ctin* sont composées uniquement de références relatives, nous traiterons les références absolues à la section 5.6.

5.1 Graphe de Dépendance Réduit (*GDR*)

Les directives *ctin* représentent les dépendances entre les modules et définissent un ensemble de contraintes de précédence. Une *fonction de base* F_p est un ensemble d'instructions de programmation définies par l'utilisateur dans le fichier module, qui sont exécutées quand un module

[2]Un bloc de directives *order* est une directive *order* extérieure (une boucle extérieure), qui peut éventuellement contenir d'autres directives *order* imbriquées. Plusieurs blocs sont, en général, nécessaires pour définir un parcours.

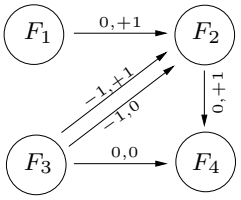

FIG. 5.1 – Exemple de *GDR* avec quatre *fonctions de base* (F_1, F_2, F_3 et F_4) rattachées à un espace à 2 dimensions et 4 *ctin*.

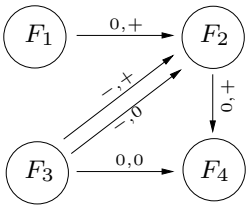

FIG. 5.2 – *GDR* simplifié déduit du *GDR* de la figure 5.1 en supprimant toutes les distances et en ne laissant que leurs signes.

qui lui est associé est appelé par la directive *order*. L'ensemble d'instructions d'une *fonction de base* est indivisible, ce qui signifie que l'application YAO exécute toutes les instructions dans une même étape. Le graphe modulaire est très similaire à un graphe appelé dans la littérature *Graphe de Dépendance Etendu* (*GDE*) [Legrand et Robert, 2003, Darte *et al.*, 2000]. Un module $F_p(I)$ est l'équivalent d'une opération[3] du *GDE*. En général le *GDE*, comme pour le graphe modulaire, ne peut pas être généré au moment de la compilation, car la génération de l'ensemble du graphe est trop coûteuse (proportionnelle au nombre de $F_p(I)$, donc proportionnelle à l'espace de calcul). En plus, ce graphe n'est pas nécessaire pour la génération automatique que nous voulons réaliser. L'analyse du *GDE* peut être faite par le biais d'un plus petit multigraphe orienté[4], en général cyclique, qui a comme sommets les *fonctions de base* F_p. Ce multigraphe correspond au *Graphe de Dépendance Réduit* (*GDR*) [Legrand et Robert, 2003, Darte *et al.*, 2000] de la théorie de la compilation et la génération automatique de code parallèle dans les boucles imbriquées. Classiquement les sommets du *GDR* sont des instructions, et non leurs instances. Du point de vue du *GDR* de YAO, les instructions du *GDR* classique sont les *fonctions de base* F_p, c'est-à-dire des macro-instructions. Dans ce graphe YAO, il y a un arc de F_s à F_d si un *ctin* est défini entre ces deux *fonctions de base* et l'arc est valué par le vecteur de distance du *ctin*. La figure 5.1 présente un exemple de *GDR*. Puisque, dans cet exemple, la dimension de l'espace est 2, les arcs sont valués par (d_i, d_j), ce qui indique que la *fonction de base* destination, au point (i, j), prend ses entrées à partir du module source au point $(i + d_i, j + d_j)$, avec $d_i, d_j \in \mathbb{Z}$. Dans le reste du manuscrit, nous avons uniquement besoin des signes des distances d_l, $l \in \{i, j, k\}$, donc on ne retient que le signe $(-, +)$ si $d_l \neq 0$ et 0 si $d_l = 0$ [5]. Cette simplification est montrée en figure 5.2.

5.2 Analyse structurelle du *GDR*

Le graphe modulaire, comme son homologue le *GDE*, est sans circuit (DAG). En utilisant le *GDR* au lieu du graphe modulaire, on perd la propriété acyclique, comme le montre l'exemple 5.3. Le *GDR* contient, en général, des circuits et des boucles ($\{F_1, F_2, F_3\}$, $\{F_6, F_7\}$, $\{F_4\}$, $\{F_5\}$ de

[3] Une opération dans la littérature est l'instance d'une instruction.

[4] Un multigraphe est un graphe avec des arcs parallèles multiples.

[5] L'abus de langage d_l au lieu de "signe de d_l" est parfois utilisé dans ce texte.

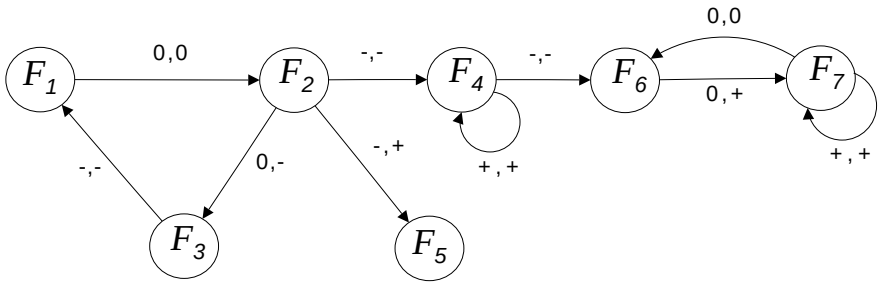

FIG. 5.3 – *GDR* avec des circuits et des boucles.

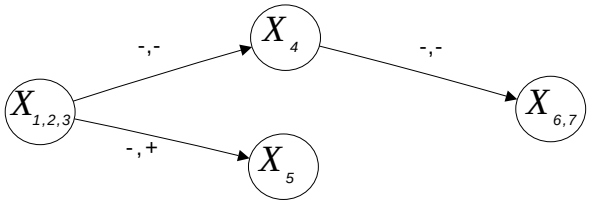

FIG. 5.4 – GDR_r : réduction du *GDR* de la figure 5.3. $X_{1,2,3}$ signifie que la réduction a impliqué les *fonctions de base* F_1, F_2 et F_3.

l'exemple). La décomposition du graphe en *Composantes Fortement Connexes* (CFC) permet de localiser l'ensemble des circuits du graphe *GDR*. La caractérisation de ces CFC se fait par le biais de la relation binaire $\mathcal{R}$ qui est définie par :

Définition 2. *Un couple de* fonctions de base F_p *et* F_q *sont en relation* ($F_p \mathcal{R} F_q$) *si et seulement si il existe un circuit dans le GDR qui contient les deux sommets* F_p *et* F_q.

Il est facile de voir que cette relation est une relation d'équivalence (en supposant la réflexivité), elle partitionne l'ensemble des sommets du *GDR* (les *fonctions de base*) en classes d'équivalence qui seront appelées les CFC du graphe. Le *GDR* peut contenir plusieurs CFC ; s'il n'en contient qu'une, il sera dit fortement connexe. Par définition, les CFC captent tous les circuits du *GDR* ; en effet, chaque circuit est inclus dans une et une seule CFC.

Comme nous l'avons signalé, les directives *order* possibles se décomposent, en général, en plusieurs blocs qui réalisent ainsi une partition de l'ensemble des *fonctions de base* en N sous-ensembles disjoints $X_1, X_2, \ldots, X_N$. On pourra définir le multigraphe réduit (du *GDR*) qui correspond à cette partition. Il suffit pour cela de considérer le multigraphe ayant N sommets, chaque sommet correspondant à un sous-ensemble de la partition, et de considérer uniquement les arcs du *GDR* ayant les deux extrémités dans deux sous-ensembles distincts. On notera ce graphe réduit GDR_r. Le graphe de la figure 5.4 est la réduction du *GDR* de la figure 5.3 lorsque l'on considère la partition définie par les CFC. Le résultat est un DAG avec 4 nœuds, chaque nœud représente une CFC. De manière plus générale, il est facile de démontrer le résultat suivant :

Résultat 1. *Le GDR_r, associé à un partition $X_1, X_2, \ldots, X_N$, est sans circuit si et seulement si chaque CFC du GDR est incluse entièrement dans l'un des sous-ensembles de la partition.*

Lorsque le GDR_r est sans circuit, il est possible de choisir un ordre topologique et de définir ainsi un ordre partiel entre les sous-ensembles X_i de la partition.

Nous présentons dans la suite un résultat qui montre comment la décomposition en CFC porte une information structurelle intrinsèque utile pour la génération des *order*.
Nous supposons que l'on dispose de directives *order* valables et permettant de parcourir le graphe modulaire défini par les directives *ctin*. On suppose que ces directives *order* se présentent sous la forme d'une suite de blocs, chacun étant caractérisé par une boucle extérieure relative à un axe de l'espace. Ces blocs partitionnent l'ensemble des *fonctions de base* en sous-ensembles disjoints.

Résultat 2. *Soit le GDR composé par l'ensemble des m* fonctions de base $\{F_p, p = 1, \ldots, m\}$. *Si $X \subset \{F_p, p = 1, \ldots, q\}$ avec $q \leq m$ est une CFC du GDR, alors toutes les* fonctions de base *de X sont liées à un même espace S_x et doivent appartenir à un même bloc de directives* order *que l'on caractérise par l'axe de sa boucle extérieure, que l'on note $l_X \in \{i, j, k\}$.*

Démonstration. Soit F_{p_i} et F_{p_j} deux *fonctions de base* du CFC X, alors, par définition, F_{p_i} et F_{p_j} sont situées dans un même circuit $F_{p_1}, \ldots, F_{p_q}$ du *GDR*. On note l_i et l_j les deux blocs *order*, qui contiennent respectivement F_{p_i} et F_{p_j}. Si nous parcourons le circuit en sens inverse, le prédécesseur $F_{p_{|i-1|}}$ [6] de F_{p_i}, doit appartenir à un bloc qui est, soit le même, soit celui qui précède l_i, car sinon le calcul de la fonction F_{p_i} ne peut se faire[7]. D'autre part, la définition de la directive *ctin* nous permet aussi d'affirmer que l'espace S_{p_i}, associé à la *fonction de base* F_{p_i}, est inclus dans l'espace $S_{p_{|i-1|}}$ associé à $F_{p_{|i-1|}}$ ($S_{p_{|i-1|}} \subset S_{p_i}$). En partant de $F_{p_{|i-1|}}$ on tire aussi que le bloc qui contient $F_{p_{|i-2|}}$ doit être égal à ou précéder celui qui contient $F_{p_{|i-1|}}$ et que $S_{p_{|i-2|}} \subset S_{p_{|i-1|}}$. Ainsi, en répétant cette descente dans le circuit, on arrive à F_{p_j} avec la condition que le bloc l_j est égal à ou précède le bloc l_i et que $S_{p_j} \subset S_{p_i}$. Si nous reprenons le raisonnement précédent en circulant en sens opposé, à partir de F_{p_j} le long du circuit, on arrive à la conclusion que le bloc l_i est égal à ou précède le bloc l_j et que $S_{p_i} \subset S_{p_j}$. De ces deux résultats on tire alors que nécessairement les deux blocs l_i et l_j sont confondus et que $S_{p_i} = S_{p_j}$. En conclusion, pour une *fonction de base* F_{p_i} de la CFC X et pour toute *fonction de base* F_{p_j} de X, F_{p_j} est située dans le bloc l_i et $S_{p_i} = S_{p_j}$; ce qui démontre le résultat. □

L'exemple 5.3 est composé de deux circuits $\{F_1, F_2, F_3\}$ et $\{F_6, F_7\}$. Le résultat 2 affirme que les *fonctions de base* $\{F_1, F_2, F_3\}$ doivent appartenir à une même boucle extérieure, comme aussi les *fonctions de base* $\{F_6, F_7\}$. Les *fonctions de base* faisant partie des deux circuits sont rattachées à deux espaces qui sont éventuellement égaux. Cette information est utile pour le processus de génération des directives *order*.

[6] $|\ \ |$ est l'opérateur module, $|i-1| = i-1 \bmod q$.

[7] Autrement dit, la boucle extérieure qui contient $F_{p_{|i-1|}}$ doit être soit la même, soit elle doit précéder la boucle extérieure qui contient F_{p_i}. Ceci découle des règles de test de la cohérence que nous avons présentées au chapitre précédent.

5.3 Anomalies dans la définition des *ctin*

Définition 3. *On dit que des directives* ctin *particulières présentent une anomalie relativement aux directives* order *s'il n'est pas possible de parcourir le graphe modulaire par un système de boucles imbriquées relativement aux axes des espaces de calcul. Autrement dit, il n'est pas possible de définir un ordre topologique sur le graphe modulaire uniquement par des directives* order.

Résultat 3. *Soit X une CFC pour laquelle le sous-graphe correspondant G_X, formé uniquement des arcs entre les sommets de X, vérifie la condition suivante : pour tout axe l de son espace de calcul, les distances d_l correspondant à tous les arcs de G_X ne sont pas de même signe ($\leq$ ou $\geq$). Alors les directives* ctin *présentent une anomalie relativement aux directives* order.

Démonstration. Nous faisons l'hypothèse que les directives *ctin* ne présentent pas d'anomalie relativement aux directives *order*. Il est alors possible de trouver des directives *order* permettant de parcourir le graphe modulaire. Alors, d'après le résultat 2, les *fonctions de base* de X se trouvent dans un même bloc *order* ; notons l_x l'axe correspondant à sa boucle extérieure. En fonction du sens de cette boucle (ascendante ou descendante), les distances d_{l_x} sur tous les arcs de G_X sont de même signe : $d_{l_x} \leq 0$ pour le sens croissant et $d_{l_x} \geq 0$ pour le sens décroissant (règle 2 du chapitre 4). Ce résultat contredit la condition retenue dans l'énoncé. Ainsi, l'hypothèse que les directives *ctin* ne présentent pas d'anomalies ne peut pas être retenue. □

Résultat 4. *Soit X une CFC pour laquelle le sous-graphe correspondant G_X, formé uniquement des arcs entre les sommets X, vérifie la condition suivante : il existe un circuit de G_X formé d'arcs ayant des vecteurs de valuation nulle. Alors les directives* ctin *présentent une anomalie relativement aux directives* order.

Démonstration. La démonstration est simple, puisque cette propriété génère un graphe modulaire avec des circuits. □

Ces résultats vont nous permettre de proposer un algorithme capable de détecter toutes les anomalies ; pour cela, nous commençons par en présenter une interprétation. Etant donnée une CFC X définie sur un espace de calcul S_x, dont la dimension est n ($n > 0$), supposons que pour un axe l donné de G_x tous les d_l sont de même signe ($\leq$ ou $\geq$)[8]. Soit F_p une *fonction de base* de X et I un point de l'espace de calcul commun S_x. Le sous-espace affine de S_x, défini par les points I ayant une composante constante, égale à l_0, suivant l'axe l, est noté H_{l_0}. Cet espace divise l'espace S_x en deux sous-espaces, $H^+_{l_0}$ et $H^-_{l_0}$, selon que l'on considère les points de S_x ayant des composantes suivant l strictement plus grandes ou strictement plus petites que l_0. Si l'on considère un arc du graphe modulaire ayant comme module source $F_s(I')$, cet arc sera orienté vers $H^+_{l_{l'}}$ si d_l est négatif et vers $H^-_{l_{l'}}$ si d_l est positif. Ainsi, à titre d'exemple, si tous les d_l, relativement à l'axe l des arcs de G_X sont < 0, alors le parcours des modules relatifs à X se réalise en parcourant, suivant les valeurs croissantes de l, les espaces H_1, H_2, H_3, etc.. Autrement dit, le parcours se fait de H_l vers les points de H^+_l. Ceci peut se présenter avec le pseudo-code :

[8] S'il n'y a pas au moins un axe l avec des distances de même signe, en suivant le résultat 3, on a une anomalie.

Soit N_l la taille maximale suivant l'axe l.
pour $l = 1$ **à** N_l **faire**
Parcourir le sous-espace affine H_l correspondant.
fin pour

De même, dans le cas où les d_l sont tous > 0, le parcours se fait de H_l vers les points de H_l^- et décrit par le pseudo-code :

pour $l = N_l$ **à** 1 **faire**
Parcourir le sous-espace affine H_l correspondant.
fin pour

Ce parcours générique permet l'analyse de tous les arcs pour lesquels d_l est strictement positif ou strictement négatif. Il ne reste plus alors qu'à analyser la faisabilité du parcours selon les axes $d_l = 0$; ceci correspond aux règles de cohérence présentées au chapitre 4. Si l'on note G_l le sous-graphe de G qui est restreint aux arcs de G pour lesquels $d_l = 0$, le problème se ramène à étudier la faisabilité de parcours par boucles imbriquées sur un espace de dimension $n-1$, qui est la projection de S_x sur les axes différents de l'axe l. Ainsi, la même méthode s'applique à G_l en le décomposant en CFC et en étudiant le signe des d_l de chacune de ses CFC, afin de déterminer un axe pour une boucle possible ou bien de détecter une anomalie. Ce processus récursif continue en réduisant régulièrement la dimension de l'espace S_x jusqu'à la dimension restante 1.

Arrivé à la dimension 1, au sein d'une CFC, deux cas peuvent se présenter :
- Si les distances restantes ne sont pas de même signe, alors il y a la détection d'une anomalie, qui indique une impossibilité de parcourir l'espace de dimension 1 restant par une boucle (ce qui équivaut aux cas précédents où la dimension est différente de 1).
- Si les distances restantes sont de même signe, alors on considère le sous-graphe qui consiste à retenir uniquement les arcs ayant les distances restantes nulles. Si ce graphe contient une composante CFC non réduite à un élément, alors il faut conclure qu'il s'agit d'une anomalie (existence d'un circuit dans G formé de vecteurs distance nuls).

Si aucun de ces deux cas d'anomalie ne se manifeste lorsqu'on analyse la dernière dimension, on peut conclure que les *ctin* ne comportent pas d'anomalie. On peut alors former la boucle relative au dernier axe (courant) et dont le sens est donné par le signe commun, croissant si les distances sont ≤ 0 et décroissant si les distances sont ≥ 0. On peut former ensuite les différentes boucles imbriquées en remontant les récursions et en intégrant les différentes boucles obtenues.

Remarque : L'anomalie qui concerne l'existence d'une boucle ayant un vecteur distance nul, $F_p(I) \rightarrow F_p(I)$, peut être détectée par YAO lors de la déclaration des directives *ctin*.

Deux types d'anomalies peuvent se présenter, on les appellera par la suite : *anomalie structurelle* et *anomalie non structurelle.*

Anomalie structurelle. Elle correspond à une anomalie des directives *ctin*, c'est le cas où le graphe modulaire généré contient des circuits ; dans ces conditions, il n'est pas possible de défi-

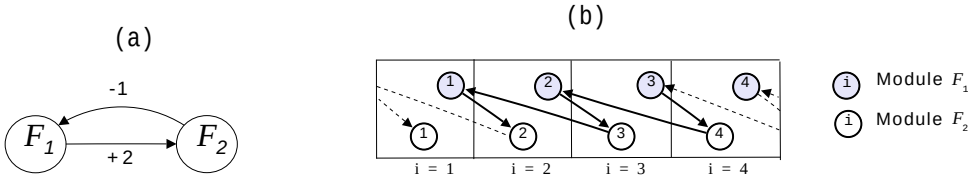

FIG. 5.5 – *Anomalie non structurelle* dans un *GDR* formé par deux *fonctions de base* dans un circuit rattachées à un espace 1D. (a) Présentation du *GDR* et (b) graphe modulaire : déploiement du *GDR* dans l'espace 1D. Ce type d'anomalie ne présente pas un circuit dans le graphe modulaire.

nir un ordre topologique sur le graphe modulaire. Cette anomalie est donc structurelle, puisque la structure du graphe modulaire ne permet pas le calcul des différents modules. L'existence d'un circuit dans le graphe modulaire consiste à partir d'un module $F_{p_0}(I)$ et à parcourir d'autres modules $F_{p_i}(I')$ pour aboutir au module initial $F_{p_0}(I)$. Ceci se traduit par l'existence d'un circuit dans le *GDR* qui contient les *fonctions de base* $F_{p_0},\ldots,F_{p_i},\ldots$ qui sont associées à un même espace de calcul. Or, chaque traversée d'un arc du graphe modulaire consiste à l'application d'un arc du *GDR* qui permet de décaler la position du module destination relativement à la position du module source, en retranchant pour chaque axe l les distances d_l correspondantes. Il est alors facile de vérifier que cette situation correspond à l'existence d'un circuit dans le *GDR* et pour lequel la somme des distances d_l, relativement à tout axe l, est nulle (dans ce cas on considère les valeurs effectives des d_l et non leur signe uniquement). Un cas particulier se présente lorsqu'il existe un circuit dans le *GDR* pour lequel les distances associées aux axes sont nulles pour tous les arcs de ce circuit. C'est le cas aussi des boucles avec des distances nulles.

Anomalie non structurelle. Cette anomalie correspond au fait que le graphe modulaire est sans circuit, mais sa structure ne permet pas de lui associer un ordre topologique sous la forme de boucles imbriquées. Donc, il n'est pas possible d'effectuer le calcul des différents modules en faisant une traversée des espaces de calcul par des directives *order*. Dans ces conditions, les directives *ctin* sont incompatibles avec les directives *order*. L'exemple de la figure 5.5 illustre cette anomalie. L'exemple correspond à une grille de calcul à 1 dimension. Le graphe 5.5b montre bien que le parcours de la grille ne permet pas de calculer (pour le même indice) F_1 et F_2. Si on se place en un point de grille d'indice i et que l'on suppose le sens être ascendant, le calcul de F_1 est possible mais le calcul de F_2 n'est pas possible, car il suppose un sens descendant. Inversement si on suppose un sens descendant.

Un exemple d'*anomalie structurelle* dans un circuit sur deux *fonctions de base* rattachées à un espace 1D est présenté en figure 5.6a.

La figure 5.6b montre un exemple où l'anomalie est confondue entre *structurelle* et *non structurelle*. Seules les valeurs des vecteurs distance peuvent mettre en évidence de quelle anomalie il s'agit. L'anomalie est détectée à cause du manque de signe commun sur un axe quelconque.

FIG. 5.6 – (a) *Anomalie structurelle* dans un circuit sur deux *fonctions de base* rattachées à un espace 1D. (b) Présence d'une anomalie, qui peut être *structurelle* ou non.

Remarque : La méthode de détection des anomalies présentée ci-dessus ne peut pas, en général, distinguer entre les deux types d'anomalies, sauf dans un cas particulier. Il s'agit de l'*anomalie structurelle*, qui concerne l'existence d'un circuit dans le *GDR* dont tous les vecteurs distance associés aux arcs sont nuls.

5.4 Méthode de génération des directives *order*

5.4.1 Présentation de la méthode

Dans la section précédente, nous avons présenté d'une manière descriptive la méthode de détection des anomalies des directives *ctin*. En pratique, la détection des anomalies est très liée à la génération des boucles imbriquées et par conséquent des directives *order*. La méthode que nous présentons permet de générer des boucles en analysant chaque composante fortement connexe. D'autre part, elle fait apparaître la nature récursive de la démarche. Comme nous l'avons vu au résultat 2, les *fonctions de base* d'une même CFC sont associées à un même espace de calcul, celui-ci est caractérisé par sa dimension (1, 2 ou 3) et la liste des axes qui lui sont associés. Ainsi, pour une dimension 2, le sous-espace peut être associé à la liste des axes (i, j), (i,k) ou (j,k). On définira par la suite une procédure de génération de l'*order* qui s'applique à une CFC X donnée. Cette procédure doit tenir compte de la CFC, de la dimension de son espace de calcul et de la liste des axes qui lui sont associés. On notera cette procédure *génération_CFC(*G_X*,*$list$*)*, où G_X représente le sous-graphe restreint à X qui est donc fortement connexe et $list$ représente la liste de ses axes. Ainsi, pour un *GDR* qui se décompose en N CFC, si l'on note $X_1, X_2, \ldots, X_N$ ses CFC ordonnées par ordre topologique relativement à leur graphe réduit GDR_r, la liste des N instructions suivantes permet, en l'absence d'anomalies, de générer des directives *order* qui permettent le parcours du graphe modulaire complet :

génération_CFC(G_{X_1}, $list_1$);
génération_CFC(G_{X_2}, $list_2$);
...
génération_CFC(G_{X_N}, $list_N$);

$list_i$ représente la liste des axes de l'espace de calcul commun aux *fonctions de base* de la CFC X_i. La procédure *génération_CFC()* est une procédure récursive que nous présentons ci-dessous. Nous introduisons, pour cela, les notations qui suivent. Etant donné un sous-ensemble X de *fonctions de base* et une liste d'axes $list$, contenant 0, 1, 2 ou 3 indices d'axes, on note $G_{X,list}$ le sous-graphe du *GDR*, qui a pour ensemble de sommets X et qui est formé par les arcs ayant des distances nulles relativement aux axes qui n'appartiennent pas à la liste $list$. D'une manière générale, l'appel de la procédure *génération_CFC($G_{X,list}$, list)* suppose que le sous-graphe $G_{X,list}$ est fortement connexe. Cette procédure permet de générer des directives *order* qui assurent le parcours des espaces affines dont les axes sont définis par la liste $list$ et permettent le calcul en chaque point des *fonctions de base* de l'ensemble X. La procédure *génération_CFC($G_{X,list}$, list)* procède de manière analogue à la méthode de détection des anomalies, qui a été décrite au paragraphe précédent. La procédure se présente de la manière suivante :

- Si $list$ est vide et X contient plus d'un élément, alors signaler une anomalie et arrêter la génération de l'*order* (il existe dans ce cas un circuit formé de vecteurs distances nulles).
- Si $list$ est vide et X contient un élément, alors retenir l'instruction qui consiste à calculer la *fonction de base* correspondante à la *fonction de base* (unique) de X. Retourner à la récursion précédente.
- Si $list$ est non vide, choisir un axe l pour lequel les distances d_l, relatives aux arcs de $G_{X,list}$, sont toutes de même signe (≥ 0 ou ≤ 0).
- Si un tel axe n'existe pas, alors arrêter la génération de l'*order* et signaler une anomalie.
- Si un tel axe l existe, alors faire les étapes suivantes :
  - Retenir l'axe l comme axe de parcours d'une directive *order*. Choisir le sens croissant (respectivement décroissant) si le signe commun est $0 \leq$ (respectivement ≥ 0). Par contre, le sens sera considéré indéterminé si toutes les distances d_l des arcs de $G_{X,list}$ sont nulles.
  - Supprimer l de la liste ($list$ est alors réduite d'un élément).
  - Considérer les CFC du graphe $G_{X,list}$ (relatif à la nouvelle $list$), soient $X_1, X_2, \ldots, X_p$ ces CFC.
  - Parcourir ces CFC suivant l'*ordre* topologique de leur graphe réduit. Pour toute CFC X_i, appeler la procédure *génération_CFC($G_{X_i,list}$, list)*.

Pour mieux comprendre ces étapes, considérons les trois exemples de la figure 5.7. La figure 5.7a représente une CFC rattachée à un espace 1D sur l'axe i et sa directive *order* générée. La CFC a une boucle négative, ce qui impose à la directive *order* d'être ascendante (YA1) sur l'axe i. Ce sens permet à $F_1(i)$ d'être correctement alimentée par $F_1(i-1)$. On remarque que c'est le seul parcours cohérent. La description de la procédure présentée permet de faire correctement cette génération. On a une seule CFC avec $X = \{F_1\}$ et $list = (i)$. Après avoir lancé la procédure *génération_CFC($G_{\{F_1\},list}$, (i))*, on choisit l'axe i, en retenant le sens de la boucle i, qui sera ascendante, puisque le signe commun est négatif et on vide la liste. On lance récursivement la procédure *génération_CFC($G_{\{F_1\},list}$, $()$)*. Dans ce cas la liste est vide et X contient un seul élément, F_1, ce qui consiste à le retenir.

La figure 5.7b représente une CFC rattachée à un espace 1D sur l'axe i et sa directive *order* générée. La CFC a un signe commun négatif sur i, ce qui impose un sens ascendant. En suppri-

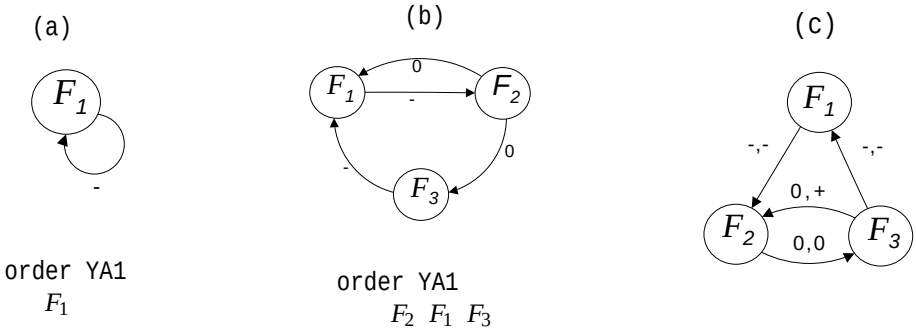

FIG. 5.7 – Exemples de *GDR* rattachés aux espaces 1D et 2D relativement aux axes (i) et (i, j). La génération des directives *order* est également donnée pour (a) et (b). (a) CFC composée d'une *fonction de base* F_1 et une boucle. (b) CFC composée de plusieurs *fonctions de base*. En retirant les axes avec un d_l négatif, on peut visualiser dans quel ordre les *fonctions de base* doivent être exécutées. La génération des directives *order* est également donnée. (c) RDG composé de trois *fonctions de base* rattachées à un espace 2D. Les différentes transformations de ce *GDR*, en appliquant l'algorithme, sont visualisées en figure 5.8.

mant les connexions qui ont des signes négatifs, on obtient le graphe qui permet de comprendre comment planifier l'exécution des *fonctions de base*. F_2 doit être exécutée en premier, puis F_1 et F_3 peuvent être exécutées dans un ordre quelconque. En suivant le pseudo-code, la procédure *génération_CFC()* est appelée avec $X = \{F_1, F_2, F_3\}$ et $list = (i)$. On choisit l'axe i et on vide la liste. Le graphe $G_{X,list}$ consiste à retirer les arcs avec $d_i \neq 0$. On lancera alors la procédure sur trois instances : *génération_CFC(*$G_{\{F_2\},()}$, $()$*)*, *génération_CFC(*$G_{\{F_1\},()}$, $()$*)* et *génération_CFC(*$G_{\{F_3\}}$, $()$*)*, l'ordre F_2, F_1, F_3 correspondant à un tri topologique du $G_{X,list}$. Comme dans ces trois appels la liste est vide, et les X_i contiennent un élément, on retient pour chacun l'instruction qui calcule la *fonction de base* correspondante, ce qui donne la génération de la figure 5.7b.

La figure 5.7c est un *GDR* composé d'une seule CFC, son espace de calcul est un espace 2D sur les axes i et j. On lance donc la procédure *génération_CFC()* avec $X = \{F_1, F_2, F_3\}$ et $list = (i, j)$. La figure 5.8 montre l'évolution de la génération des directives *order* et les différentes étapes de la transformation du graphe. Le signe commun de la composante se trouve sur *i*, il est négatif. L'algorithme identifie la première directive : *order YA1*, l'axe i est retiré de la liste qui sera réduite à (j). Le graphe initial est transformé, comme le montre la figure 5.8a, la procédure *génération_CFC()* est appelée récursivement sur les CFC du graphe $G_{\{F_1,F_2,F_3\},(j)}$. Le nouveau calcul des CFC donne deux CFC, X_1 et X_2 (la première composée de $\{F_1\}$ et la seconde composée de $\{F_2, F_3\}$). En supposant que le tri topologique est X_1, X_2, on lance la procédure d'abord sur $X1$, puis sur X_2. Etant donné que le graphe $G_{X_1,(j)}$ (formé par les seuls nœuds de X_1) n'a pas de connexion et que la liste contient un élément, l'algorithme identifie la directive *order YX2*, où X peut être A ou B (figure 5.8b). Ensuite l'instruction qui calcule la fonction F_1 est retenue (figure 5.8c). A noter que la CFC X_2, composée de F_2 et F_3, est toujours en attente d'être traitée par la récursion précédente, pendant que X_1 (composée de F_1) a déjà terminé son processus de

génération. Une fois X_1 traitée, on lance *génération_CFC($G_{\{F_2,F_3\},(j)}$, (j))*. Le signe commun de X_2 est sur l'axe j, il est positif. Une directive *order YB2* est identifiée et le graphe transformé est présenté à la figure 5.8d. A nouveau la *génération_CFC()* est appelée sur le graphe transformé. Le calcul des CFC donne deux CFC, composées respectivement de F_2 et F_3, le tri topologique étant F_2, F_3. On retient les *fonctions de base* en suivant le tri topologique, comme le montre la figure 5.8e. Ceci est fait par deux appels de *génération_CFC()* et *génération_CFC()* avec $G_{\{F_2\},()}$ et $G_{\{F_3\},()}$.

5.4.2 Procédures de génération

On présentera dans cette section les procédures en pseudo-code de la génération et de la détection des anomalies. Lorsqu'une anomalie est détectée, on doit sortir de la procédure *génération_CFC()* courante et arrêter le processus de génération dans sa totalité. On appelera *exit* le fait d'arrêter la génération en utilisant un terme utilisé dans les langages de programmation. Ceci suppose que l'on arrête les récursions en amont de la procédure.

D'autre part, les différents appels de *génération_CFC()* doivent construire l'arbre qui implémente les différentes directives *order* imbriquées (section 4.3). Chaque appel de la procédure doit développer, à partir d'un sommet, cet arbre. Ainsi, aux deux paramètres précédents, on doit ajouter un troisième paramètre qui correspond à un nœud de l'arbre, on note *arb* la variable qui référencie ce nœud. La procédure *génération_CFC($G_{X,list}$, list)* présentée devient donc *génération_CFC($G_{X,list}$, list, arb)* qui est à trois paramètres, mais qui est identique à la précédente, sauf qu'elle génère l'arbre des directives *order*.

La procédure *génération(G,arb)* permet d'initialiser et de lancer les procédures *génération_CFC()*. Le pseudo-code de cette procédure est donné par l'algorithme 5. La préparation consiste à créer le nœud racine de l'arbre, déterminer les N CFC $(X_1, X_2, \ldots, X_N)$ du *GDR* et les ordonner selon le tri topologique. Le *GDR* peut être composé par *fonctions de base* rattachées à des espaces divers. On note n_i la dimension de l'espace relatif à la CFC X_i (S_{X_i}) et $list_i$ est la liste de ses axes. On appelle ensuite N fois *génération_CFC($G_{X_i,list_i}$, $list_i$, arb)*. Le pseudo-code de la procédure *génération_CFC()* est présenté dans l'algorithme 6.

L'algorithme 6 présente la nouvelle version de la procédure *génération_CFC(G, list, arb)*. Il

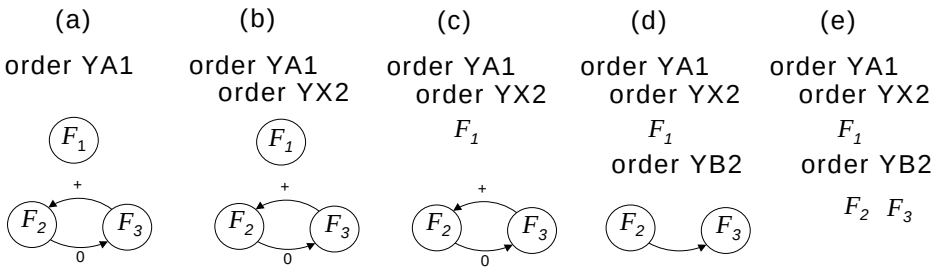

FIG. 5.8 – Evolution de la procédure *génération_CFC()* et manipulation du graphe réalisée à partir du *GDR* de la figure 5.7.

Algorithme 5 Procédure *génération(G,arb)*. Cette procédure prépare les CFC du graphe *G*, initialise la variable *arb* et lance la procédure *génération_CFC($G_{X_i,list_i}$, $list_i$, arb)* pour chaque CFC. S'il n'y a pas d'anomalie, elle retourne l'arbre complet des directives *order* générées.

ENTRÉES: *G* est un *GDR*.
SORTIES: Retourne l'arbre généré ou une anomalie.

1: Créer un nœud pour la racine de l'arbre et référencer ce nœud par la variable *arb*.
2: Déterminer les CFC de *G*, on suppose qu'il existe N composantes X_i.
3: Ranger les CFC en suivant l'ordre topologique, soit $X_1, X_2, \ldots, X_N$.
4: Soit n_i la dimension de l'espace de calcul des *fonctions de base* de X_i et $list_i$ la liste de ses n_i axes.
5: **pour** $i = 1$ **à** N **faire**
6: **appelle** *génération_CFC($G_{X_i,list_i}$, $list_i$, arb)*.
7: **fin pour**
8: **retourner** *arb*.

s'agit d'une procédure récursive qui a trois entrées : un *GDR* fortement connexe *G*, la liste *list* des axes de l'espace auquel les sommets de *G* sont rattachés et la référence *arb* du nœud de l'arbre. A partir d'*arb* la procédure développe la branche de l'arbre relative à un bloc de directives *order*. On aura une branche pour chaque CFC du *GDR* initial.

La ligne 3 correspond au cas de base de la récursion, qui correspond à la liste vide. La variable *list* contient les axes de *G* qui ne sont pas encore traités, elle est réduite d'une unité à la ligne 23 avant de lancer la nouvelle récursion. S'il n'y a plus d'axe à considérer (*list* vide), la récursion s'arrête. Lorsque tous les axes ont été traités, le graphe $G_{X,list}$ est formé d'arcs ayant des vecteurs distance nuls. Le test de la ligne 5 permet de détecter une *anomalie structurelle* relative à l'existence d'un circuit formé d'arcs avec des vecteurs distance nuls.

Les anomalies dues au fait qu'il n'y a pas un signe commun sont traitées à la ligne 17. Nous expliquons plus en détail la procédure *génération_CFC()* en développant deux exemples. Le premier exemple montre la détection d'une anomalie, le second montre une génération standard.

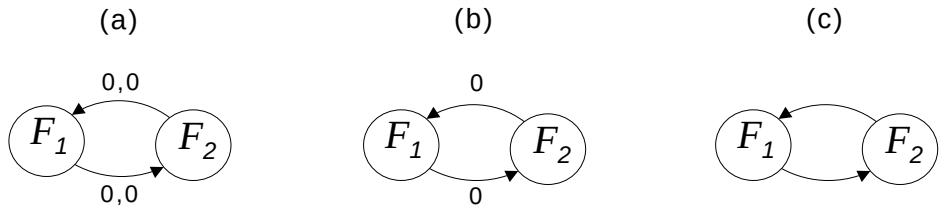

FIG. 5.9 – (a) *GDR* composé de *fonctions de base* rattachées à un espace 2D sur les axes i, j. Ce *GDR* a une anomalie structurelle due à la présence d'un circuit avec des vecteurs distance nuls. (b) et (c) montrent deux étapes de transformation du *GDR* initial au cours des appels de la procédure *génération_CFC()*.

Algorithme 6 Procédure *génération_CFC(G, list, arb)*. Cette procédure génère récursivement les directives *order* à partir du graphe G. G doit être fortement connexe.

ENTRÉES: G est un GDR dont les sommets sont limités à X. *list* est la liste des axes de l'espace S_x ou d'un sous-espace affine de S_x. *arb* est une référence à un nœud de l'arbre des directives *order*.

SORTIES: La construction d'un nœud de l'arbre ou la détection d'une anomalie.

Soit X les nœuds de G.
si *list* est vide **alors**
//Cas de base de la récursion.
  si $|X| > 1$ **alors**
  //*Anomalie structurelle* : circuit avec vecteur distance nul.
    exit erreur.
    //Avec un message d'anomalie au niveau des *fonctions de base* de X.
  sinon
    Soit F_p la seule *fonction de base* présente dans X.
    Créer un nœud fils d'*arb*.
    //Il s'agit d'une feuille.
    Placer F_p dans le nœud créé.
    retourner
  finsi
finsi
Chercher un signe commun dans G, on suppose que l'axe est l.
si Pas de signe commun dans G **alors**
  exit erreur.
  //Avec un message d'anomalie au niveau des *fonctions de base* de X.
finsi
Créer le nœud fils d'*arb*. On appelle la référence à ce nœud arb_x.
Retenir l'axe l comme axe de parcours d'une directive *order*. Choisir le sens croissant YA (respectivement décroissant YB) si le signe de l est ≤ 0 (signe l est ≥ 0). Le sens sera considéré indéterminé (YX) si toutes les distances d_l des arcs de G sont nulles.
Supprimer l de la liste *list* (elle sera diminuée d'un élément).
Considérer toutes les CFC du graphe $G_{X,list}$, soient $X_1, X_2, \ldots, X_N$ les composantes triées topologiquement.
pour $i = 1$ **à** N **faire**
  appelle *génération_CFC(*$G_{X_i,list}$, *list*, arb_x*)*.
fin pour

Exemple 1. Figure 5.9 est un *GDR* composé de deux *fonctions de base* rattachées à un espace à deux dimensions sur les axes i, j, il est fortement connexe. La procédure *génération(*G, arb*)* appelle une fois la procédure *génération_CFC()*, le signe commun est donné indistinctement par les deux axes *i* et *j*, il est indéterminé. La procédure choisit *i* par défaut. La première directive *order YX1*, est identifiée à la ligne 22, la liste est diminuée de *i* et la procédure *génération_CFC()* est appelée une fois pour la CFC $\{F_1, F_2\}$ (figure 5.9b). Ce nouveau appel de *génération_CFC()* qui génère une directive *order YX2* vide la liste et appelle la procédure *génération_CFC* sur le graphe de figure 5.9c qui est toujours fortement connexe. La liste étant vide, la ligne 5 détecte une *anomalie structurelle*, qui correspond à l'existence d'un circuit avec vecteurs de distance nuls.

Exemple 2. Cet exemple montre la génération des directives relatives au *GDR* de la figure 5.3 repris dans la figure 5.10. En appliquant l'algorithme 6, on génère les directives *order* pour chaque

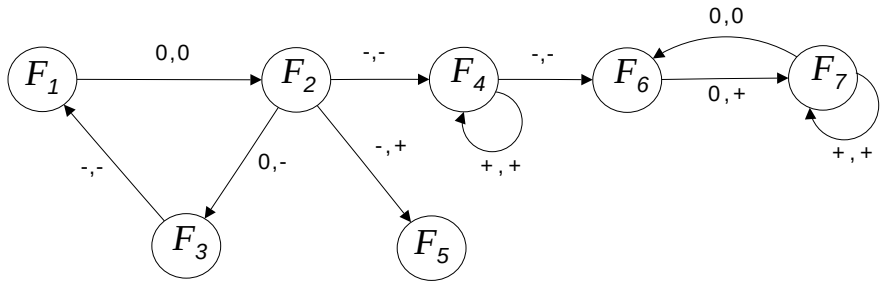

FIG. 5.10 – *GDR* avec des circuits et des boucles.

sommet (CFC) du graphe GDR_r de la figure 5.4, en suivant un ordre topologique, ce qui conduit à la suite des appels ci-dessous :

génération_CFC($G_{\{F_1,F_2,F_3\},(i,j)}, (i,j), arb$)
génération_CFC($G_{\{F_5\},(i,j)}, (i,j), arb$)
génération_CFC($G_{\{F_4\},(i,j)}, (i,j), arb$)
génération_CFC($G_{\{F_6,F_7\},(i,j)}, (i,j), arb$)

A noter que c'est une génération donnée par le choix d'un tri topologique particulier. A titre d'exemple, concernant le graphe $G_{\{F_1,F_2,F_3\},(i,j)}$ l'axe *i* admet un signe commun négatif. Ainsi, la boucle extérieure qui concerne cette CFC est *i* et elle est d'un sens ascendant (YA1). La composante F_5 n'a pas de connexion interne, tous les axes et les sens peuvent être choisis, on choisit par défaut *i* ce qui donne donc la directive *order YX1*. Le même raisonnement vaut pour $\{F_4\}$ et $\{F_6, F_7\}$ qui imposent un sens descendant pour les deux axes *i* et *j*. Le calcul d'un nouvel ordre topologique de l'exemple précédent conduit à la génération des appels de la figure 5.11.

La procédure *génération_CFC(*$G_{\{F_i\},(j)}, (j), arb$*)* calcule un nouveau signe commun pour les axes *j*. Comme dans cet exemple particulier le graphe initial ne contient pas de boucles, un signe quelconque peut être choisi. La procédure de génération se termine lorsqu'il n'y a plus d'axe à

(a)

```
order YA1
   génération_CFC(G_{F1,(j)},(j),arb)
   génération_CFC(G_{F2,(j)},(j),arb)
   génération_CFC(G_{F3,(j)},(j),arb)
order YX1
   génération_CFC(G_{F5,(j)},(j),arb)
order YB1
   génération_CFC(G_{F4,(j)},(j),arb)
order YB2
   génération_CFC(G_{F7,(i)},(i),arb)
   génération_CFC(G_{F6,(i)},(i),arb)
```

(b)

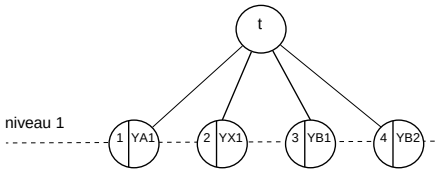

FIG. 5.11 – Génération partielle des directives *order* relatives au *GDR* de figure 5.10. (a) Appels récursifs de la procédure génération_CFC(). (b) Génération de l'arbre *arb* des directives *order*.

(a)

```
order YA1
   order YX2
      F1
   order YX2
      F2
   order YX2
      F3
order YX1
   order YX2
      F5
order YB1
   order YX2
      F4
order YB2
   order YX1
      F7
   order YX1
      F6
```

(b)

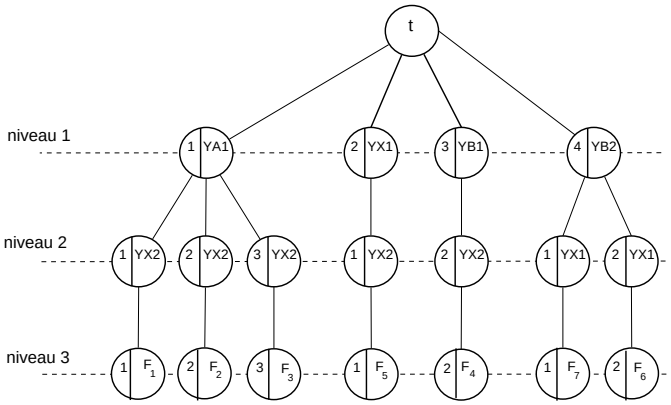

FIG. 5.12 – Génération des directives *order* relatives au *GDR* de figure 5.10. En (a) on a les directives *order* et en (b) l'arbre *arb* des directives générées.

traiter et, dans ce cas, elle retourne les *fonctions de base* qui sont contenues dans la CFC. La dernière génération est montrée en figure 5.12.

5.5 Fusion de boucles

L'exécution de l'algorithme précédent permet de développer l'arbre qui représente les directives *order* générées. Le premier niveau de l'arbre correspond aux racines des sous-arbres qui correspondent aux blocs de directives *order*. La version actuelle de cet algorithme distribue ces blocs à toutes les composantes CFC du *GDR*, ce qui induit une forte dispersion de ces directives en des blocs élémentaires. Il serait intéressant de procéder à des regroupements (fusions) afin de réduire leur nombre. Le problème de la fusion est un problème classique dans le domaine de la compilation et parallélisation automatique de code [Darte, 1999, Kennedy et Mckinley, 1994].

Comme nous l'avons déjà signalé, chaque itération d'une boucle imbriquée correspond au passage par un point d'un espace de calcul et exécute les *fonctions de base* qui sont associées à cette boucle. Il ne s'agit donc pas d'instructions simples, mais l'exécution d'un code qui peut être relativement important. Ainsi, le regroupement doit tenir compte de ces caractéristique techniques dans l'objectif d'optimiser le temps de calcul et la gestion de la mémoire. Si nous considérons à titre d'exemple le *GDR* de la figure 5.10, l'application de l'algorithme amène à générer quatre blocs de directives *order* qui correspondent aux quatre CFC du GDR_r (figure 5.4) : $X_{1,2,3} = \{F_1, F_2, F_3\}$, $X_5 = \{F_5\}$, $X_4 = \{F_4\}$ et $X_{6,7} = \{F_6, F_7\}$. Mais plusieurs choix de parcours peuvent être choisis pour chacune de ces CFC. La boucle extérieure dans le cas de l'exemple peut être :

- *order* YA1 ou YA2 (pour $X_{1,2,3}$),
- *order* YX1 ou YX2 (pour X_5),
- *order* YB1 ou YB2 (pour X_4),
- *order* YB1 ou YB2 (pour $X_{6,7}$).

Ainsi, d'une manière générale, à chaque boucle on peut associer une liste de parcours possibles, chaque parcours étant caractérisé par un couple $(l, sens)$, où $l \in \{1,2,3\}$ représente le numéro de l'axe du parcours et $sens \in \{A, B\}$ (ascendant, descendant) représente le sens de parcours de l'axe. Ainsi, pour l'exemple précédent, on pourra attribuer aux boucles extérieures des CFC du graphe les listes suivantes :

- ((1,A), (2,A)) pour $X_{1,2,3}$,
- ((1,A), (1,B), (2,A), (2,B)) pour X_5,
- ((1,B), (2,B)) pour X_4,
- ((1,B), (2,B)) pour $X_{6,7}$.

Le regroupement de deux blocs *order* revient à fusionner deux sommets du GDR_r. Pour que deux blocs de boucles puissent être candidats à un regroupement direct, il faut qu'ils puissent apparaître à deux places consécutives dans l'un des ordres topologiques possibles du GDR_r. Ainsi, les deux sommets $X_{1,2,3}$ et $X_{6,7}$ ne peuvent pas être candidats à un regroupement direct, car ils ne peuvent pas être placés d'une manière consécutive dans un ordre topologique, le sommet X_4 est toujours placé entre ces deux sommets. Le regroupement de deux sommets s et d, qui occupent deux positions consécutives dans un ordre topologique, n'est possible que lorsque leur liste des axes et

sens de parcours L_s et L_d ont des éléments en commun. On notera $L_{s,d}$ l'intersection de ces deux listes. D'autre part, la fusion de leurs sommets associés du GDR_r amène souvent à intégrer un ou plusieurs arcs du *GDR* dans le nouveau bloc *order* obtenu ; il faut donc que ces arcs soient compatibles avec au moins l'un des éléments de la liste $L_{s,d}$. Pour cela, on attribuera à chaque arc $s \rightarrow d$ du *GDR* une liste de même type, qui énumère tous les parcours qui sont compatibles avec son vecteur distance. Ainsi, à titre d'exemple, on attribue :

- à l'arc $X_{1,2,3} \rightarrow X_4$ la liste ((1, A), (2, A)),
- à l'arc $X_{1,2,3} \rightarrow X_5$ la liste ((1, A), (2, B)),
- à l'arc $X_4 \rightarrow X_{6,7}$ la liste ((1, A), (2, A)).

Le regroupement de deux blocs *order*, correspondant à deux sommets s et d consécutifs dans un ordre topologique, est possible lorsque les deux conditions suivantes sont vérifiées :

1. la liste intersection $L_{s,d}$ est non vide,
2. le *GDR* étant un multigraphe, l'intersection de la liste $L_{s,d}$ avec l'intersection de toutes les listes associées aux arcs $s \rightarrow d$ est non vide.

Lorsque l'une de ces deux conditions n'est pas vérifiée, le regroupement n'est pas possible. Si le regroupement est possible, on associe alors au nouveau nœud la liste L qui correspond à l'intersection de la liste $L_{s,d}$ avec l'intersection de toutes les listes des arcs $s \rightarrow d$ du *GDR*. Ainsi, concernant l'exemple précédent, les regroupements se présentent de la manière suivante :

- le regroupement de $X_{1,2,3}$ et X_4 est impossible car $L_{X_{1,2,3},X_4}$ est vide,
- le regroupement de $X_{1,2,3}$ et X_4 est possible, le nouveau nœud aura la liste ((1, A)),
- le regroupement de X_4 et $X_{6,7}$ est impossible car l'intersection de $L_{X_4,X_{6,7}}$ avec la liste de l'arc $X_4 \rightarrow X_{6,7}$ est vide,
- le regroupement de X_5 et $X_{6,7}$ est possible, le nouveau nœud aura la liste $L_{X_5,X_{6,7}}$ = ((1, B), (2, B)) (pas d'arc entre ces deux nœuds).

Ainsi, le regroupement de blocs de directives *order* peut se faire, en choisissant un ordre topologique donné du *GDR* et en procédant à la fusion des nœuds consécutifs dans cet ordre. La recherche d'un regroupement "optimal" suivant un certain critère peut être combinatoire. Nous proposons par la suite une démarche générale qui pourrait être utilisée lors de la recherche de fusions sub-optimales relativement à un critère donné.

5.5.1 Fusion par mise en niveaux

Le GDR_r est un DAG, on peut réorganiser les CFC par niveaux. Les niveaux sont numérotés de $k = 1$ à $k = MLevel$, où *MLevel* est le nombre maximum de niveaux. Le premier niveau, $k = 1$, contient les nœuds sans prédécesseurs ; les prédécesseurs d'une CFC au niveau k, avec $k > 1$, sont localisés dans les niveaux précédents k', $k' \leq k-1$, avec au moins un prédécesseur localisé au niveau $k-1$. Grâce à la réorganisation par niveaux, il n'y a pas d'arcs entre deux sommets au même niveau. L'organisation des sommets du GDR_r en niveaux permet de proposer une méthode générale pour aborder le problème de la fusion.

Il est possible de proposer un ordre topologique, dit en *largeur d'abord*, en parcourant les niveaux par ordre croissant et en listant les nœuds de chaque niveau dans un ordre quelconque.

Ainsi, relativement à ce type d'ordre topologique, il est possible de placer en positions consécutives deux nœuds d'un même niveau lors de la constitution d'un ordre topologique, ces deux nœuds seront alors candidats pour une fusion possible. La fusion de deux nœuds d'un même niveau k, ne modifie pas la structure en niveaux du graphe. En effet, le nouveau nœud (résultant de la fusion) reste au niveau k et tous les autres nœuds gardent leur niveau initial. Ainsi, il est possible, pour chaque niveau k du graphe, d'itérer ce type de fusion afin d'aboutir à un graphe réduit pour lequel, s'il y a plusieurs nœuds au niveau k, ils sont deux à deux incompatibles (pour la fusion). Cet algorithme de fusion favorise la fusion en *largeur d'abord*.

Il est aussi possible de procéder à des fusions qui favorisent le regroupement de nœuds appartenant à des niveaux différents. Cette méthode consiste à définir des ordres topologiques dits en *profondeur d'abord*. Ce type d'ordre topologique consiste, après avoir choisi un nœud d'un niveau k du graphe réduit, à favoriser le placement immédiat de nœuds qui sont dans des niveaux supérieurs à k. Un tel ordre topologique peut être généré en faisant une traversée qui s'apparente à la traversée en profondeur du GDR_r [9]. La combinaison des deux procédures qui suivent permet de réaliser le parcours en *profondeur d'abord*. A partir d'un nœud s situé sur un niveau k, la procédure *lister_profondeur(s,k)* permet de former une suite de nœuds faisant partie d'un tri topologique et appartenant à des niveaux supérieurs du graphe.

Algorithme 7 Procédure *lister_profondeur(s,k)*. Cette procédure permet de marquer les nœuds d'un graphe réduit de façon à former une suite, une liste qui contient l'ordre topologique. Elle démarre à partir d'un sommet s et favorise l'ordonnancement en *profondeur d'abord*.

ENTRÉES: s est un sommet du graphe réduit et k est le niveau de ce sommet.
  Marquer s (ordonnancer s dans la liste de l'ordre topologique).
  pour tout d au niveau $k+1$ **faire**
    si tous les prédécesseurs de d sont déjà marqués **et** s et d sont compatibles **alors**
      Appeler la procédure *lister_profondeur(d,k+1)*.
    finsi
  fin pour

La procédure suivante, *tritop_profondeur()*, permet de compléter le tri topologique qui favorise l'ordonnancement en profondeur.

[9] En effet, la traversée classique en *profondeur d'abord* consiste à prendre les arcs d'un nœud vers un successeur. Or, parfois, certains arcs rendent la fusion incompatible et par conséquent bloquent le parcours. D'autre part, la méthode choisie permet de regrouper (en profondeur) des nœuds non adjacents.

Algorithme 8 Procédure *tritop_profondeur()*.

```
Traverser le graphe réduit en largeur.
pour tout sommet s au niveau k rencontré faire
  si s est non marqué alors
    Appeler la procédure lister_profondeur(s,k).
  finsi
fin pour
```

Les deux techniques de parcours *largeur d'abord* et *profondeur d'abord* permettent ainsi de former un ordre topologique en suivant une certaine stratégie. Il est alors possible de parcourir la liste de l'ordre topologique obtenue et de fusionner, si c'est possible, les nœuds consécutifs. A partir de ces différentes stratégies de fusion, il est possible de définir des stratégies de fusion dynamiques, qui, en chaque nœud, fusionnent des nœuds en combinant des recherches locales en largeur et en profondeur. L'ordre topologique n'étant pas unique, on peut éventuellement énumérer tous les ordres topologiques et puis en choisir un selon un critère d'intérêt. Au niveau pratique, le choix d'une stratégie de fusion de boucles dépendra des critères qu'il s'agit d'optimiser. Ces critères ne sont pas encore clairs à l'état actuel. Ils sont dépendant de l'architecture matérielle de la machine particulière qui est utilisée pour faire tourner l'application YAO. Dans le chapitre de conclusion, on parlera davantage de ces perspectives.

Nous notons que la fusion analysée dans cette section prend en compte seulement la boucle extérieure, c'est-à-dire les nœuds du premier niveau de l'arbre des directives *order*. Ces nœuds sont les racines des sous-arbres qui représentent les blocs de directives *order*. Ceci est fait en supposant d'avoir choisi un axe parmi ceux qui sont disponibles. Les mêmes techniques peuvent être appliquées au sous-arbres, en écartant l'axe déjà traité. Ceci permet d'avoir moins de dispersion dans les niveaux 2 et 3 de l'arbre. Par ailleurs, le choix de l'axe de la boucle extérieure peut ne pas être évident. D'une part, le choix d'un axe au lieu d'un autre peut permettre de réduire la dispersion des directives *order*, d'autre part, le choix préférentiel de l'axe qui optimise les mémoires caches doit être imposé. La section C analyse brièvement cet aspect, bien connu dans la littérature, pour l'optimisation des accès aux mémoires caches.

Remarque : Concernant le fusion, une façon différente de faire est de fusionner les sommets pendant la traversée. Ceci évite de faire la fusion tout à la fin, mais en même temps oblige de faire une remise en niveaux du graphe. En effet, au moment de traverser les sommets en profondeur, une fusion au niveau k peut impliquer une réorganisation du graphe à cause du changement de niveau d'un ensemble de sommets. Cette réorganisation très fréquente affecte les sommets qui sont localisés aux niveaux $k' > k$, les niveaux précédents ne sont jamais touchés. A cause de cette réorganisation, nous avons préféré donner d'abord un ordre topologique et faire ensuite la fusion.

5.6 Références absolues dans la génération automatique

Comme nous l'avons signalé dans l'introduction, la version actuelle de YAO a été développée au fur et à mesure en fonction des applications réelles qui ont été traitées. Ainsi, ses fonctionnalités ont été introduites afin d'aborder les différents problèmes qui se posaient au niveau de chaque application. En particulier, la référence absolue a été introduite, afin de traiter des cas particuliers d'initialisation. Mais, telle qu'elle est définie dans YAO, cette notion permet de faire des déclarations qui permettent de représenter des situations qui vont bien au delà de l'objectif initial (l'initialisation). Ainsi, la généralité de cette formalisation représente un effet de bord de la façon dont elle a été introduite dans YAO. Cette généralité ne correspond donc pas à une spécification bien formalisée avec les utilisateurs potentiels, et qui correspond à des cas réalistes.

Afin de rester dans l'esprit initial qui a motivé son introduction dans YAO, nous imposons par la suite la contrainte suivante : s'il existe une référence absolue entre deux *fonctions de base* F_s et F_d, alors F_s et F_d ne peuvent pas être dans une même CFC du *GDR*. En effet, cette contrainte est naturelle si l'on restreint le rôle des références absolues à l'initialisation. Si cette contrainte est imposée, la génération automatique des directives *order*, pour chaque CFC du *GDR*, reste inchangée par rapport aux techniques introduites dans ce chapitre. En même temps, si ces références absolues sont utilisées pour l'initialisation, elles doivent être nécessairement très limitées et localisées à des endroits bien précis du calcul. Il est possible de définir des règles de fusion entre deux boucles, lorsque celles-ci sont connectées, dans leur graphe réduit, par au moins une *connexion de base* contenant une référence absolue. On pourrait aussi décider de ne jamais fusionner deux CFC de ce type, cette décision ne devrait pas affecter les performances du programme généré, compte tenu du nombre très limité de ces connexions. Une exploration avec des modélisateurs numériciens est envisagée dans la suite des travaux de cette thèse. Il s'agit de formaliser, avec eux, d'une manière plus précise, cette notion, de bien cadrer sa généralité et de lever certaines de ses limitations. Ce point sera discuté au chapitre de conclusion.

Chapitre 6

Génération automatique de codes parallèles

Traditionnellement, les logiciels informatiques ont été écrits pour le calcul séquentiel. Pour résoudre un problème, un algorithme est conçu et mis en œuvre sous forme de flux sériel d'instructions. Ces instructions sont exécutées séquentiellement sur une unité centrale de traitement d'un ordinateur. D'autre part le calcul parallèle utilise à la fois plusieurs unités de traitement pour résoudre le même problème. Il s'effectue en décomposant l'algorithme en parties indépendantes, afin que chaque unité de traitement puisse exécuter le travail qui lui revient simultanément avec les autres unités. Les unités de traitement peuvent être très variées. Elles peuvent être formées d'un seul ordinateur avec un ou plusieurs processeurs, de cœurs multiples dans un même processeur, de plusieurs ordinateurs dans un réseau, ou de toute autre combinaison.

Dans un contexte de programmation parallèle, un système à *mémoire partagée* désigne un système composé d'un large bloc de mémoire vive qui est accédé par différentes unités de traitement au sein d'un même ordinateur. En général, l'expression *mémoire distribuée* est utilisée en opposition à celle de mémoire partagée et concerne les systèmes avec des architectures distribuées, comme par exemple les grappes de calcul. Le reste du chapitre s'intéressera à la parallélisation automatique pour des architectures à mémoire partagée du code généré par YAO.

Les sous-tâches d'un programme parallèle, dans le cadre des systèmes à mémoire partagée, sont souvent appelées *threads* (processus léger) [Nichols *et al.*, 1996]. Les *threads* ont souvent besoin d'accéder (lecture/écriture) à des variables qu'ils partagent entre eux, ce qui peut causer un phénomène de corruption de données, appelé dans la littérature *data race condition* ou *race condition*. Dans le cadre des applications générées par YAO, deux types de *race conditions* apparaissent[1] :

– Si l'un des *threads* réalise une écriture sur une variable partagée et qu'un autre a besoin de lire cette variable, comme les deux *threads* s'exécutent en parallèle, alors l'ordonnancement de ces deux opérations est imprévisible. On ne peut pas garantir ce que lit le deuxième

[1] Nous discuterons de ces deux *race conditions* lors de la présentation de la parallélisation des procédures *forward* et *backward*.

thread.

– Si les deux *threads* écrivent sur une même variable partagée, il se peut que les deux écritures adviennent simultanément, ce qui peut causer un résultat imprévisible.

Les *race conditions* posent donc des problèmes de synchronisation entre *threads* en lecture/écriture sur ces variables partagées. Une approche classique dans la conception de la parallélisation en mémoire partagée d'un programme consiste à analyser le code de programmation et à trouver les sous-tâches qui sont "parallélisables". L'analyse de tous les types de *race conditions* permet de définir la stratégie de parallélisation à retenir.

Les programmes parallèles sont plus difficiles à écrire que les programmes séquentiels, car la parallélisation introduit plusieurs nouveaux types de bugs logiciels potentiels, dont les *race conditions* sont les plus courants. Communication et synchronisation entre les différentes sous-tâches sont généralement l'un des plus grands obstacles à l'obtention d'une bonne efficacité dans les programmes parallèles. Idéalement, l'accélération (*speedup*) due à la parallélisation est linéaire, doubler le nombre d'unités de traitement devrait réduire de moitié le temps d'exécution, et les doubler une seconde fois devrait à nouveau réduire de moitié le temps d'exécution. Cependant, très peu d'algorithmes parallèles atteignent une telle accélération optimale. La plupart d'entre eux ont une accélération dont la courbe est presque linéaire pour un petit nombre d'unités de traitement et s'aplatit tendant vers une valeur constante pour un grand nombre d'unités de traitement. Une mesure de la capacité d'un programme à réduire le temps d'exécution avec un nombre croissant de processeurs est dénommée *extensibilité parallèle* (*parallel scalability*) [Pacheco, 1996].

L'accélération potentielle d'un algorithme sur une plateforme de calcul parallèle est limitée par les parties du programme qui ne peuvent être parallélisées. Tous les problèmes de calcul scientifique et d'ingénierie sont typiquement constitués de plusieurs parties parallélisables et plusieurs autres non parallélisables (séquentielles). Le temps passé à exécuter un programme est appelé temps CPU (*CPU time*). Le temps réel attendu pour obtenir le résultat est l'*elapsed time*. La différence entre l'*elapsed time* et le *CPU time* correspond à un temps perdu par l'application ; par exemple elle n'a pas obtenu l'utilisation complète d'un processeur, à cause d'une forte charge sur le système. Le speedup est le rapport entre les *elapsed time* des programmes séquentiel et parallèle. Cette mesure est gouvernée au niveau théorique par la loi d'*Amdahl* [Chapman *et al.*, 2007] qui peut être formulée de la façon suivante :

$$S = \frac{1}{(f_{par}/P + (1 - f_{par}))}, \tag{6.1}$$

où f_{par} est la fraction parallèle du code et P est le nombre de processeurs. Dans le cas idéal où tout le code s'exécute en parallèle ($f_{par} = 1$), l'accélération attendue est égale au nombre de processeurs. Si seulement 80 pour cent du code tourne en parallèle ($f_{par} = 0.8$), on peut s'attendre à ce que l'accélération maximale sur 16 processeurs soit égale à 4 et sur 32 processeurs à 4.4. Il est donc important de paralléliser le plus de code possible, afin de rendre l'application plus extensible (*scalable*).

D'autres obstacles à l'obtention d'une accélération linéaire parfaite sont les coûts introduits par le fait de lancer, fusionner et synchroniser les *threads*, ainsi que les accès mémoire. Compte tenu de tous ces facteurs, la parallélisation du code n'aboutit pas nécessairement à une accéléra-

tion du temps de calcul. En règle générale, quand une tâche est divisée pour être exécutée par un nombre important de *threads*, ces *threads* pourraient dépenser voire plus de temps que l'exécution séquentielle. Parfois, le temps dû à la communication dépasse le temps passé à résoudre le problème, et ainsi la parallélisation augmente l'*elapsed time* de l'application. Ce phénomène est connu comme *ralentissement parallèle* (*parallel slowdown*).

Dans le cadre de l'assimilation variationnelle de données concernant des gros modèles numériques, le temps d'exécution d'un programme peut être très important, d'où l'intérêt de la parallélisation du code généré par YAO. En outre, les techniques connues sous le nom de *checkpointing* permettent d'optimiser l'allocation mémoire en opérant parfois un recalcul. Le compromis entre recalcul et allocation mémoire est un aspect très important dans les applications d'assimilation de données. Un code optimisé en temps de calcul permet l'utilisation des techniques de *checkpointing*. Ces techniques d'optimisation mémoire sont introduites à l'annexe A.

Avec YAO, l'utilisateur définit, à l'aide de directives spécifiques, le type de discrétisation et la spécification du modèle numérique direct. YAO génère alors automatiquement les modèles numérique et adjoint, à travers la programmation objet C++. Le formalisme YAO est basé sur le graphe de dépendance et sur le graphe modulaire, similaires à ceux utilisés dans la parallélisation automatique des boucles imbriquées. Ce domaine de recherche est bien développé, plusieurs concepts et algorithmes ont été introduits, permettant l'analyse des boucles imbriquées, leur décomposition et leur fusion [Allen et Kennedy, 1982, Allen *et al.*, 1987, Darte *et al.*, 2000, Darte, 1999, Kennedy et Mckinley, 1994]. Nous montrons dans ce chapitre comment ces algorithmes peuvent être adaptés à YAO, afin d'identifier le parallélisme disponible et de permettre la génération automatique de code parallèle. Grâce aux directives *OpenMP*, il est alors facile de générer un code parallèle pour les architectures à mémoire partagée.

Le développement d'algorithmes spécifiques à YAO, pour la génération automatique de code parallèle avec des directives OpenMP, est souhaitable. En effet, les logiciels existants pour la parallélisation automatique avec OpenMP ont des contraintes spécifiques liées à leur conception. Par exemple, l'outil *CAPO* [Jin *et al.*, 2000] est limité au langage Fortran et repose sur l'interaction avec l'utilisateur pour l'amélioration du processus de parallélisation. Le framework *Gaspard2* [Taillard *et al.*, 2008] permet la génération automatique de code OpenMP, mais les sous-tâches de parallélisme doivent être précisées par l'utilisateur dans un modèle UML. Enfin, le logiciel *PLuTo* [Bondhugula *et al.*, 2008] peut paralléliser efficacement les boucles imbriquées en tenant compte, via la technique du pavage (*tiling* en anglais [Wolfe, 1989]), de la localité des données sur des architectures multicœurs. Cependant *PLuTo* ne supporte pas la programmation orientée objet comme code source en entrée. Ces outils ne peuvent donc pas actuellement être directement intégrés dans l'architecture C++ YAO afin de générer automatiquement du code OpenMP. En effet, ces contraintes d'architecture et de langage sont des contraintes fortes, puisqu'il faudrait reconcevoir intégralement YAO pour pourvoir exploiter ces outils. Au chapitre 7 de conclusion, nous discuterons d'une possible intégration d'un de ces outils dans YAO. Elle consiste à casser le système de transformation *source-to-source* de *PLuTo* pour exploiter seulement des parties bien déterminées de cette chaîne (ces parties étant très modulaires, il est envisageable de les interfacer avec YAO).

Dans ce chapitre, nous montrons que les informations fournies par les directives *ctin*

contiennent les informations nécessaires pour la génération du graphe de dépendance qui permet la parallélisation automatique du code. Cela permet d'éviter l'étape de l'analyse de dépendance présente dans la plupart des logiciels de parallélisation automatique [Jin *et al.*, 2000, Bondhugula *et al.*, 2008].

Dans la section 6.1 nous présentons succinctement le pseudo-code généré par YAO en se focalisant sur la procédure *forward*. A la section 6.2, nous illustrons une technique de parallélisation automatique de la procédure *forward* lorsque les directives *order* sont déjà fixées. La section 6.3 présente une méthode pour l'intégration de la parallélisation lors de la génération automatique des directives *order*. Elle discutera aussi de la problématique de fusion de ces directives. La section 6.4 illustre la génération de code et la parallélisation automatique de la procédure *backward* qui introduit quelques particularités de synchronisation par rapport à la technique décrite à la section 6.2.

6.1 La génération de code

6.1.1 Quelques aspects génériques du générateur YAO

Le graphe modulaire défini dans le fichier de description est transformé en code C++ par le générateur YAO. Chaque instance du fichier de description correspond à une instance du code généré. Le code change pour chaque modification du fichier de description. Un code généré a obligatoirement une portion de code pour la procédure *forward*, une portion de code pour la procédure *backward* et éventuellement une portion de code pour la procédure *linward*. Ces procédures sont aussi appelées procédures *ward*. Il y a une troisième portion de code pour l'initialisation, les entrées/sorties, le calcul de la fonction de coût, etc.. Dans certains cas particuliers nous pourrions avoir une portion pour le test du gradient et une portion pour le calcul du linéaire tangent (*linward*). Le test du gradient est exécuté une fois dans la phase de test de l'application, il correspond à un code qui n'est pas présent dans la phase de production des résultats de l'assimilation. Le linéaire tangent est utilisé pour le test du gradient, pour la méthode incrémentale et pour la méthode duale.

Du point de vue de la parallélisation, un ensemble d'instructions est intéressant si le temps CPU est supérieur à ceux d'autres ensembles. Ainsi, le premier objectif dans la programmation parallèle est de comprendre quelles sont les parties de code qui sont coûteuses en temps de calcul. Certains logiciels de profilage de code (*code profiling*) [Gprof, 1993, VTune, 2009, Sun, 2005, Oprofile, 2000, PAPI, 1999] s'avèrent très utiles pour identifier les portions de code où les programmes dépensent le plus de temps CPU. L'effort de parallélisation peut alors être concentré sur les routines les plus coûteuses. En faisant tourner *Gprof* sur des applications YAO, il est facile de voir que les portions de code les plus coûteuses en temps sont les procédures *forward*, *linward* et *backward*. Le reste du code est négligeable et représente un faible pourcentage de l'ensemble total du programme. Ainsi, il est intéressant de travailler sur la parallélisation de ces procédures.

Nous considérons dans la suite un exemple de fichier de description et nous illustrons le procédure *forward* que YAO génère. Nous parlerons brièvement de la procédure *linward* qui ne présente pas de caractéristiques particulières, du point de vue du parallélisme, par rapport à la procédure

(a)

```
ctin F1 from F1  i   j-1 t-1
ctin F2 from F1  i   j+1 t
ctin F2 from F3  i-1 j+1 t
ctin F2 from F3  i   j   t-1
ctin F2 from F4  i+1 j   t
ctin F3 from F1  i-1 j   t-1
ctin F4 from F3  i   j   t
ctin F4 from F2  i   j+1 t
```

(b)

```
order YA1
   order YA2
      F1 F3


order YB1
   order YB2
      F2 F4
```

FIG. 6.1 – Portion de langage de description défini par l'utilisateur avec *fonctions de base* rattachées à un espace 2D. (a) Les directives *ctin*. On ne montre pas dans ces *ctin* les numéros d'entrée/sortie. On suppose que toutes les *fonctions de base* n'ont qu'une seule sortie. (b) Les directives *order*.

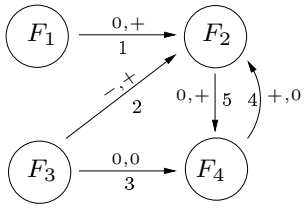

FIG. 6.2 – *GDR* issu des directives de figure 6.1 en ne dessinant que les connexions avec $d_t = 0$. Les arcs sont numérotés de 1 à 5.

forward. Les directives *ctin* et *order* de cet exemple sont données en figure 6.1 et le Graphe de Dépendance Réduit (*GDR*) en figure 6.2. Etant donné que l'espace est de dimension 2 et que les arcs retenus correspondent à des délais temporels nuls, les arcs sont valués par les signes du vecteur distance (d_i, d_j). Le graphe de la figure 6.2 sera utilisé dans ce texte comme référence pour l'analyse de la parallélisation automatique du code[2].

Les méthodes *forward* et *backward* de chaque *fonction de base* sont écrites par l'utilisateur et intégrées par le générateur YAO dans le code généré. Ces fonctions locales sont appelées par les procédures *forward* et *backward* (de YAO) suivant un ordre défini par les directives *order*. Cette section donne plus de détails sur la génération effectuée par le générateur YAO. L'analyse des codes générés met en évidence les similitudes entre le *GDR* de YAO et la théorie de la parallélisation automatique des boucles imbriquées. Ainsi, ces techniques, qui sont le résultat de plusieurs décennies de recherche, peuvent être adaptées au formalisme de YAO. Les résultats de ces recherches sont très étendus puisqu'ils traitent de codes très généraux peuvant être représentés par

[2]Nous avons omis les numéros d'entrée/sortie des directives *ctin* dans la figure 6.1a. Nous supposons que chaque *fonction de base* a une seule sortie.

```
loop i ascendant
   loop j ascendant
     // Initialisation du point de référence input.
     // Il s'agit de l'entrée de la fonction forward du module F1(i,j,t).
     input[0]=F1(i,j-1,t-1)
     // Calcul de la sortie de F1(i,j,t).
     F1(i,j,t).forward(input)

      input[0]=F1(i-1,j,t-1)
      F3(i,j,t).forward(input)

loop i descendant
   loop j ascendant
      input[0]=F1(i,j+1,t)
      input[1]=F3(i-1,j+1,t)
      input[2]=F3(i,j,t-1)
      input[3]=F4(i+1,j,t)
      F2(i,j,t).forward(input)

      input[0]=F3(i,j,t)
      input[1]=F2(i,j+1,t)
      F4(i,j,t).forward(input)
```

FIG. 6.3 – Génération de la procédure *forward* de la part du générateur YAO des directives 6.1a et 6.1b.

le *modèle polyédrique* [Bastoul, 2004, Legrand et Robert, 2003, Darte *et al.*, 2000]. Les boucles générées par YAO sont plus simples, puisque le nombre maximal de boucles imbriquées est égal à 3 et le nombre d'itérations de chaque boucle est constant (respectivement N_i, N_j et N_k). D'autre part, dans le cadre de la procédure *forward*, les dépendances entre modules sont toutes de type dépendance de *flot*[3].

6.1.2 Génération de la procédure *forward*

Dans la figure 6.3 nous donnons, en pseudo-langage, la génération faite, à partir des directives de la figure 6.1, de la procédure *forward*. A chaque directive *order* correspond une boucle, on aura une boucle pour chaque dimension des espaces de calcul. Le sens des axes, ascendant ou descendant, et l'ordonnancement des *fonctions de base* sont ceux spécifiés dans les directives *order*. Une *fonction de base* F_p représente ainsi une classe qui a deux méthodes locales, le *forward* et le *backward*, et certains attributs comme l'état de la fonction F_p. Il existe un objet instance de cette classe pour chaque point I de l'espace de calcul. L'état représente la valeur du vecteur de sortie d'un module. Ainsi, si une *fonction de base* a v sorties, l'état du module correspondant à

[3]La dépendance de *flot* fait partie des trois types de dépendances : *flot*, *anti* et *sortie*. Ces dépendances ont été introduites par [Bernstein, 1966]. Une dépendance de *flot* est décrite, intuitivement, par une opération d'écriture suivie d'une opération de lecture dans la même case mémoire.

cette *fonction de base* calculée sur un point de grille est un vecteur de dimension v. Pour chaque module $F_p(I,t)$, YAO définit son vecteur d'entrée à partir des sorties des modules prédécesseurs et appelle la fonction *forward* locale en passant ce vecteur d'entrée. Ceci permet le calcul de son vecteur de sortie. Les vecteurs de sortie ne sont pas montrés dans le pseudo-code, parce qu'ils sont directement affectés par les méthodes *forward* locales 3.5.3.

Les méthodes *forward* locales sont définies pour chaque *fonction de base* par l'utilisateur. Du point de vue de YAO, ce sont des boîtes noires qui assurent le calcul de ces fonctions ou bien des codes précédemment compilés par l'utilisateur. Les boucles imbriquées permettent de calculer les sorties des *fonctions de base* pour tous les points de grille et pour un pas de temps. Une boucle globale, non représentée dans la figure, permet de traverser les pas de temps dans un ordre ascendant t, $t+1$, $t+2$, etc., et constitue la boucle qui englobe toutes les autres. Ainsi, pour une itération de la boucle globale en temps, nous calculons les fonctions F_p sur tous les points de grille pour un pas de temps. La boucle de temps peut être considérée comme une barrière de calcul où, au temps t, tous les calculs déjà réalisés dans le passé sont connus.

Les différents blocs de directives *order* sont exécutés en séquentiel dans l'ordre spécifié. Dans les applications réelles, nous pouvons avoir des dizaines de blocs de directives. La boucle extérieure (ascendante ou descendante) qui représente un bloc de directives *order* correspond à un axe ; dans le reste du chapitre, on note l cet axe et n_l le nombre de *fonctions de base* contenues dans cette boucle extérieure. Si cette boucle extérieure est ascendante, alors toutes les distances d_l des arcs entre les n_l *fonctions de base* sont nécessairement négatives ou nulles, en raison de l'hypothèse de cohérence du chapitre 4. De même, si l est descendante, alors toutes les distances d_l sont positives ou nulles.

La section suivante présente la parallélisation automatique de la procédure *forward* (la procédure *backward* est traitée à la section 6.4).

6.2 Une technique de parallélisation de directives *order* existantes

6.2.1 Décomposition de domaine

L'espace de calcul peut être partitionné logiquement de façon à ce que chaque sous-espace soit associé à un *thread*. Cela pourrait constituer la base de la parallélisation. Dans le programme partitionné, chaque *thread* sera responsable du calcul de son sous-espace (sa sous-grille de calcul). Chaque *thread* porte essentiellement sur la même tâche, mais sur sa propre portion de données. Cette approche est souvent appelée *décomposition de domaine* (*domain decomposition*). Dans la suite on considère la décomposition suivant un axe de la grille de calcul. Elle consiste à partitionner l'intervalle de variation d'un axe l choisi en q sous-intervalles consécutifs et de considérer les q sous-espaces (bandes) obtenus. La figure 6.4 présente un exemple de décomposition de domaine suivant l'axe i dont l'intervalle de variation a été partitionné en trois sous-intervalles ([0,1], [2,3], [4,6]). Ainsi, dans le cadre de la parallélisation d'un bloc de directives *order* ayant comme boucle extérieure l'axe l, la décomposition de domaine à une dimension suivant l permet de décomposer

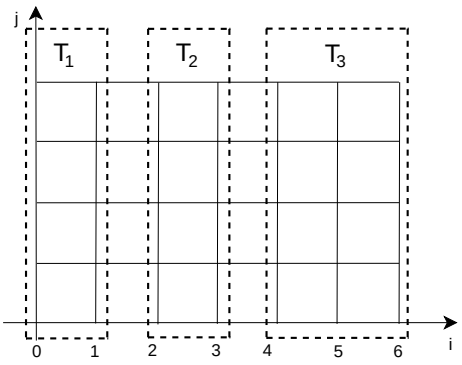

FIG. 6.4 – Décomposition de domaine d'un espace de calcul de dimensions 5x7 avec 3 *threads* T_i. La décomposition est faite en partitionnant l'axe *i* en trois sous-intervalles ([0,1], [2,3], [4,6]). Les *threads* T_1, T_2, T_3 sont associés aux trois sous-domaines obtenus.

le bloc initial en *q* sous-blocs. Chaque sous-bloc sera traité par un *thread*[4]. Que peut-on dire concernant les problèmes de synchronisation entre deux *threads* ?

Les données mises à jour par un *thread* et utilisées par un autre doivent être manipulées avec précaution, la synchronisation doit veiller à ce que le bon ordre d'accès soit respecté. La synchronisation des *threads* est le principal problème à résoudre chaque fois que nous avons une *race condition*. Si deux *threads* sont exécutés en parallèle, les calculs qu'ils effectuent sont indépendants. Ainsi, la synchronisation entre les opérations relevant de ces deux *threads* doit être assurée par des instructions spécifiques. La complexité et la fréquence de ces opérations peut induire un surcoût important. Dans la section 6.2.2, on présente une technique de parallélisation automatique par décomposition de domaine à une dimension. On se place dans le cas le plus classique où on suppose qu'on a un seul bloc de directives *order* dont l'axe extérieur est *l*. Dans un premier temps, on suppose aussi que c'est l'utilisateur qui fournit les directives *order* imbriquées. L'objectif de cette démarche est d'avoir une présentation plus naturelle et compréhensible de la problématique. Celle-ci est, par ailleurs, une démarche classique puisque, historiquement, dans la littérature de la parallélisation automatique des boucles imbriquées, le problème était posé de cette façon. Il s'agit donc de réaliser une décomposition de domaine suivant l'axe *l*, d'extraire d'une manière automatique, relativement à cette décomposition, les parties parallélisables. La même procédure est appliquée à tous les autres blocs. L'étude de ce cas particulier va nous fournir les éléments permettant, à la section 6.3, de traiter le cas le plus général, qui consiste à intégrer la parallélisation lors de la génération automatique des directives *order*.

[4]Dans la figure 6.4, nous avons décomposé l'espace suivant l'axe *i* mais, la décomposition peut être faite similairement en suivant l'axe *j*.

6.2.2 Un algorithme de parallélisation automatique de la procédure *forward*

En suivant les travaux d'Allen-Kennedy [Allen et Kennedy, 1982, Allen *et al.*, 1987], on propose un algorithme pour la parallélisation automatique de la procédure *forward* pour des architectures à mémoire partagée. La présence des fortes dépendances sur l'axe du temps dans les modèles numériques rend la parallélisation suivant cet axe peu intéressante. On concentre donc les efforts sur la parallélisation par décomposition de domaine au sein d'un pas de temps. Comme déjà signalé, on utilise une décomposition de domaine 1D, l'espace est coupé selon un seul axe parmi les axes qui le composent. La décomposition de domaine repose sur une répartition de charge (*load balancing*) statique puisque dans les applications YAO actuelles, la charge de calcul de chaque *fonction de base* est constante sur chaque point de grille.

On suppose que les fonctions *ward* locales sont *thread-safe*. Dans la littérature, on dit qu'une fonction, ou une portion de code, est *thread-safe* si elle fonctionne correctement durant une exécution simultanée par plusieurs *threads*. En effet, la fonction *forward* locale utilise un espace mémoire partagée, mais en chaque point I, le module correspondant accède à une partie de cet espace mémoire qui lui est dédiée ; cette partie de l'espace est donc associée à un et un seul *thread*. Le fait de supposer que les fonctions *ward* locales sont *thread-safe* s'avère primordial pour le processus de parallélisation. Cette hypothèse doit soigneusement être respectée par les utilisateurs. YAO ne sait pas a priori ce qu'il y a dans les fonctions *ward* locales. Ces fonctions sont mises en œuvre par les utilisateurs dans les fichiers module, comme expliqué à la section 3.5.3. Aucune analyse automatique de ces codes n'est envisagée dans ce travail. Par conséquent, l'hypothèse *thread-safe* est fondamentale pour ne pas avoir de *race conditions* non résolues.

L'algorithme que nous présentons ici décompose en premier lieu le bloc initial traité en une suite de sous-blocs élémentaires, puis analyse le caractère parallélisable ou pas de chaque sous-bloc. Il s'agit de détecter le maximum de parallélisme disponible et de réduire les points de synchronisation. Ce travail est possible grâce à l'analyse du *GDR*, afin de mettre en évidence les connexions qui seront dites *critiques*, car elles pourraient conduire à un problème de synchronisation. La figure 6.2 représente le *GDR* des directives *ctin* de la figure 6.1a restreint aux arcs ayant pour distance $d_t = 0$. Par contre, la figure 6.1b donne les deux blocs de directives *order* associées à cet exemple. Les deux blocs correspondent à une boucle extérieure relative à l'axe *i*. La figure 6.5 illustre le deploiement en un point de grille du *GDR* restreint dans l'espace à deux dimensions. Cette figure contient aussi une décomposition de domaine relativement à l'axe *i*. Si l'on considère le premier bloc de directives *order* de la figure 6.1b, alors le sous-graphe du *GDR* de la figure 6.2 limité aux deux *fonctions de base* F_1 et F_3 est sans arcs. Ainsi, au sein de cette boucle, il n'existe pas de dépendance entre F_1 et F_3. Par conséquent, cette boucle ne pose pas de problèmes de synchronisation lors de sa parallélisation. Par contre, si l'on considère le second bloc de la figure 6.1b, alors le sous-graphe du *GDR* de la figure 6.2 limité aux deux *fonctions de base* F_2 et F_4 contient les deux arcs n°4 et n°5. La figure 6.5 montre la position relative de ces deux arcs dans l'espace de calcul. Ces positions montrent bien que l'arc n°5, pour lequel $d_i = 0$, correspond à une dépendance qui s'effectue toujours dans le cadre d'un même *thread*. Par contre, l'arc n°4, qui correspond à l'arc (F_4, F_2) du *GDR* et pour lequel $d_i \neq 0$, force une synchronisation entre les couples de *threads* (T_1, T_2) et (T_2, T_3). Cette synchronisation se pose notamment sur les points de

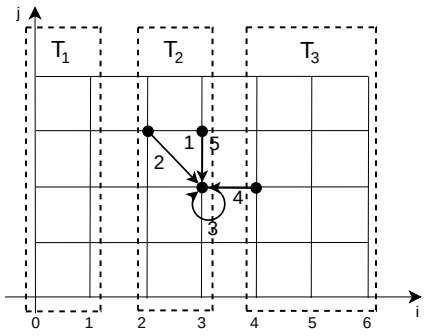

FIG. 6.5 – Décomposition de domaine selon l'axe i en trois sous-espaces, chacun affecté à un *thread*. On visualise la synchronisation pour le point i, j. Les numéros sur les flèches sont les mêmes que ceux des arcs de la figure 6.2.

frontière des sous-domaines obtenus.

Définition 4. *Etant donné un bloc de directives* order *ayant comme axe extérieur l et une connexion $F_s \rightarrow F_d$ du GDR, cette connexion sera dite* critique *relativement à ce bloc si les deux conditions suivantes sont réalisées :*

- *F_s et F_d sont deux* fonctions de base *internes au bloc,*
- *$d_t = 0$ et $d_l \neq 0$.*

Il est facile de voir que seules les connexions *critiques* imposent une synchronisation pour tous les *threads* associés à une décomposition de domaine suivant l'axe l. Faire disparaître toutes les connexions *critiques* au sein d'un bloc (par la technique de décomposition abordée par la suite, par exemple) correspond à faire disparaître toutes les dépendances de *flot* entre *threads*. La méthodologie utilisée pour la parallélisation de la procédure *forward* est caractérisée par la minimisation de ces dépendances de *flot*. Intuitivement, si au sein d'un bloc il n'y a pas de dépendances de *flot* entre *threads*, le bloc sera considéré parallélisable.

De ce qui précède, l'analyse d'un bloc de directives *order* admettant une boucle extérieure, relative à l'axe l, et composée de l'ensemble des fonctions $F = \{F_1, F_2, \ldots, F_n\}$, revient à considérer le sous-graphe G_F du *GDR* limité à l'ensemble de ces *n fonctions de base* et aux arcs pour lequels $d_t = 0$. D'autre part, étant donné que les *n fonctions de base* se trouvent dans la même boucle extérieure l, les distances d_l associées aux arcs de G_F sont de même signe (≤ 0 ou ≥ 0). On attribue à chaque arc de G_F la valuation 0 si $d_l = 0$ et la valuation $+$ ou $-$ si $d_l \neq 0$.

En s'inspirant de l'algorithme d'Allen-Kennedy [Allen et Kennedy, 1982, Allen *et al.*, 1987], l'analyse du G_F permet de décomposer la boucle en plusieurs boucles. Cet algorithme permet de déterminer une décomposition de la boucle extérieure, tout en préservant l'hypothèse de cohérence du calcul. L'algorithme se présente de la manière suivante :

- Déterminer les CFC de G_F, on suppose qu'il existe N composantes X_i qui définissent la partition $X = \{X_1, X_2, \ldots, X_N\}$ de F.
- Considérer le graphe réduit, noté $G_{F/X}$, en réduisant chaque CFC X_i en un nœud et en

traçant un, et seulement un, arc entre deux CFC s'il existe au moins un arc de la première à la seconde dans le graphe G_F. Si au moins un des arcs en G_F qui relient ces deux CFC est valué par $+$ ou $-$, alors on value l'arc correspondant dans $G_{F/X}$ par ce signe. Autrement, si toutes les valuations sont 0, alors on value l'arc correspondant dans $G_{F/X}$ avec 0 [5].

– Ranger les CFC en suivant un ordre topologique. En suivant cet ordre, générer, pour chaque CFC, une boucle extérieure d'axe l.

Tenant compte de la règle 2, cette décomposition est une distribution maximale du bloc initial, autrement dit on ne pourrait pas décomposer plus sans rompre l'hypothèse de cohérence. Elle reprend la même idée de génération automatique des directives *order* présentée dans le chapitre précédent mais elle s'applique à un sous-graphe G_F. Ce graphe, une fois décomposé, permet une détection facile de parallélisme possible dans les boucles extérieures et relativement à l'axe l. On peut analyser chaque boucle élémentaire liée à une composante X_i, afin d'effectuer une décomposition de domaine sur l'axe l. La boucle élémentaire est considérée comme parallélisable si elle n'impose pas de synchronisation entre les *threads*.

Règle 5. *Soit G_F le graphe décrit précédemment. Pour chaque CFC X_i du graphe G_F on considère tous les arcs de G_F qui ont pour extrémités deux* fonctions de base *faisant partie de X_i. Si au moins un de ces arcs est* critique *($d_t = 0$ et $d_l \neq 0$), la boucle élémentaire relative aux* fonctions de base *de X_i n'est pas parallélisable. Sinon, elle sera considérée comme parallélisable*[6].

Ainsi cette distribution maximale, en boucles élémentaires du bloc initial, permet d'isoler tous les groupes d'instructions qui peuvent être considérés comme parallélisables. Nous étiquetons par la suite avec p et $\overline{p}$ les boucles élémentaires respectivement parallélisables et non parallélisables. On appellera cette procédure *algorithme de décomposition*.

Une distribution maximale de boucles n'est pas la meilleure solution en terme d'efficacité parce qu'elle incrémente le nombre de points de synchronisation entre boucles, c'est-à-dire le nombre de barrières implicites de chaque bloc de boucles imbriquées sur lesquelles les *threads* se synchronisent avant de passer au bloc suivant. En outre, le fait d'avoir un nombre maximal de boucles ajoute un temps supplémentaire dû au fonctionnement de ces boucles. Il est possible de proposer des méthodes pour la fusion de ces boucles.

Généralement les fusions des boucles sont guidées par des critères à optimiser. Ces critères sont dépendant des architectures matérielles utilisées : nombre de registres, nombre de mémoires caches (L1, L2, L3) et leur taille, taille de la mémoire RAM, type d'architecture (32 ou 64 bits), etc.. En effet, une fusion très importante a l'effet de réduire le surcoût dû au fonctionnement des boucles, mais en même temps pourrait empêcher un fonctionnement optimal des mémoires caches et, encore à plus bas niveau, des registres de l'unité de traitement. Une mauvaise utilisation de la hiérarchie mémoire se répercute directement sur les performances de l'application. D'autre part, les critères pour obtenir un code performant sont aussi liés aux problèmes de synchronisation entre unités de traitement (dans le cadre du parallélisme), à la répartition de la charge de calcul

[5] Le graphe $G_{F/X}$ correspond à une portion du graphe GDR_r défini dans le chapitre précédent.

[6] En effet des transformations de boucle plus complexes, comme par exemple celle unimodulaire [Darte *et al.*, 2000], peuvent être appliquées, en rendant ainsi dans certains cas la boucle parallèle. Nous ne considérons pas ces transformations dans ce travail.

(nous avons choisi une répartition de charge statique, mais ceci pourrait ne pas être une solution générale), etc.. Les critères d'optimisation ne sont pas clairs à l'état actuel, une exploration est envisagée dans des travaux futurs en collaboration avec des spécialistes de ce domaine. La liberté donnée par cette thèse de choisir la façon de fusionner les sommets du graphe permettra d'en choisir une qui optimise le temps de calcul pour une architecture matérielle particulière. L'intégration d'informations sur l'architecture matérielle s'avère être une prospective intéressante, puisqu'elle permettrait de gérer le compromis entre le regroupement maximal (pour gagner en nombre de boucles) et un regroupement qui optimise la hiérarchie des mémoires (pour optimiser la localité spatiale et temporelle des données). Dans la suite, nous faisons une présentation générale des méthodes de fusion qui peuvent être appliquées dans le cadre de l'optimisation d'un critère donné. Nous donnons un exemple d'algorithme de fusion. Les méthodes traitées correspondent à une adaptation des méthodes présentées au chapitre 5 aux boucles étiquetées (p et $\overline{p}$). Etant donné que les critères d'optimisation ne sont pas maîtrisés à l'état actuel, cette présentation permettra par la suite de s'inspirer de ces méthodes générales de fusion pour les adapter aux critères d'optimisation.

Le $G_{F/X}$ est un DAG, on peut réorganiser les CFC par niveaux, comme expliqué dans la section 5.5.1. Grâce à la réorganisation par niveaux, il n'y a pas d'arcs entre deux sommets au même niveau. Nous pouvons procéder à des fusions *largeur par niveau*, ce qui correspond à fusionner tous les sommets étiquetés par p et tous les sommets étiquetés par $\overline{p}$. En effet, on peut toujours fusionner les sommets de même type grâce à la fusion *largeur par niveau*, puisque ceci donne en résultat un DAG, ce qui permet d'organiser les sommets en suivant un ordre topologique. Nous obtenons un graphe réduit qui a le même nombre de niveaux, mais avec un ou deux sommets par niveau. On appelle ce graphe $G_{F/X}$ réduit. Si un niveau contient deux sommets, ils sont obligatoirement marqués comme p et $\overline{p}$. Le processus de fusion peut être étendu à des sommets situés sur des niveaux différents (fusions en profondeur).

Etant donné que nous considérons une suite de boucles élémentaires issues de la décomposition d'un bloc de directives *order*, défini par un axe extérieur l, le test de la cohérence du point de vue de la fusion des boucles ne se pose pas dans ce cas. Le seul test à considérer est la nature parallélisable ou pas des boucles élémentaires à fusionner. D'une manière générale, nous sommes intéressés à faire des fusions qui respectent deux points. La fusion doit :

1. garder globalement le plus haut degré de parallélisme et
2. l'ordre de fusion doit respecter un ordre topologique.

Le premier point nous oblige à imposer des règles de fusion par rapport à l'étiquette des sommets. La fusion est faite de la façon suivante :

- la fusion de deux sommets valués par $\overline{p}$, donne un nouveau sommet $\overline{p}$,
- la fusion de deux sommets valués par p qui ne sont pas connectés par un arc *critique* ($d_l \neq 0$) donne un nouveau sommet p.

En effet, l'existence d'une connexion *critique*, entre deux sommets de type p, casse la propriété de parallélisation de ces deux sommets. Pour la même raison on ne fusionne jamais deux sommets de type différent et on peut toujours fusionner deux sommets $\overline{p}$. Cette façon de faire maintient le plus haut degré de parallélisation. D'autre part, le point 2 nous oblige à fusionner les sommets en respectant un certain ordre par rapport aux niveaux. Fusionner deux sommets qui sont dans deux

niveaux très éloignés n'est pas une bonne démarche, car ceci pourrait ne pas correspondre à un ordre topologique et générer un circuit dans le nouveau graphe.

Pour respecter les deux points précédents, nous proposons une méthode de fusion locale. A partir du graphe obtenu par la *largeur par niveaux*, cette méthode permet, en partant d'un sommet s quelconque, d'explorer des sommets appartenant à des niveaux de plus en plus éloignés. Le résultat est une sous-liste de sommets qui peuvent être fusionnés et qui respectent un ordre topologique. Cette façon de procder est une méthode locale, car elle cherche à fusionner des sommets qui sont proche de s. Si s est étiqueté par $\bar{p}$, la procédure *lister_profondeur_non_p(s,k)* suivante décrit cette méthode :

ENTRÉES: s est un sommet de type $\bar{p}$ du $G_{F/X}$ réduit et k est le niveau de ce sommet.
SORTIES: Formation d'une sous-liste, à partir de s.

1: Ordonnancer s dans la liste de l'ordre topologique.
2: **pour** tout d au niveau $k+1$ **faire**
3: **si** tous les prédécesseurs de d sont déjà ordonnancés **et** d admet l'étiquette $\bar{p}$ **alors**
4: Appeler la procédure *lister_profondeur_non_p(d,k+1)*.
5: **finsi**
6: **fin pour**

Cette méthode est légèrment différente si on veut l'appliquer aux sommets de type p. Nous notons qu'un arc entre deux sommets qui sont analysés pour être fusionnés n'apporte aucune restriction sur la fusion des sommets de type $\bar{p}$. La fusion de deux sommets p, en revanche, doit tenir compte de l'arc qui les lie, il faut interdire toute fusion dans le cas où cet arc est *critique*. La procédure *lister_profondeur_p(s,k)* suivante propose la variante à appliquer aux sommets p.

ENTRÉES: s est un sommet de type p du $G_{F/X}$ réduit et k est le niveau de ce sommet.
SORTIES: Formation d'une sous-liste, à partir de s.

1: Ordonnancer s dans la liste de l'ordre topologique.
2: **pour** tout d au niveau $k+1$ **faire**
3: **si** tous les prédécesseurs de d sont déjà ordonnancés **et** d admet l'étiquette p **et** l'arc (s,d) est non *critique* **alors**
4: Appeler la procédure *lister_profondeur_p(d,k+1)*.
5: **finsi**
6: **fin pour**

L'application des deux procédures *lister_profondeur_non_p(s,k)* et *lister_profondeur_p(s,k)* permet de former des sous-listes qui peuvent être intégrées dans la recherche d'un ordre topologique qui tient compte du type des sommets (p et $\bar{p}$). Les directives *order* fournies par l'utilisateur peuvent alors être écrites comme une combinaison de boucles parallèles et séquentielles. Les deux procédures peuvent, à titre d'exemple, être appelées par l'algorithme 9. Cet algorithme réalise une approche mixte, puisqu'on suppose des fusions qui favorisent d'abord la largeur et puis la profondeur par niveaux.

Algorithme 9 Algorithme générique de fusion par niveaux des sommets du graphe $G_{F/X}$ en gardand le plus haut degré de parallélisme.

ENTRÉES: Le graphe $G_{F/X}$.

1: Organiser le graphe $G_{F/X}$ par niveaux.
2: Pour chaque niveau, fusionner les sommets qui ont la même étiquette (p ou $\bar{p}$). On appelle le graphe obtenu $G_{F/X}$ réduit (un niveau contient un ou deux sommets)
3: // La boucle qui suit réalise un ordre topologique des sommets du $G_{F/X}$ réduit obtenu en favorisant la profondeur des niveaux.
4: **pour** tout niveau k pris par ordre croissant **faire**
5: **pour** tout sommet s de niveau k rencontré **faire**
6: **si** s est non ordonnancé **alors**
7: **si** s est étiqueté par $\bar{p}$ **alors**
8: Appeler la procédure *lister_profondeur_non_p(s,k)*.
9: **sinon**
10: Appeler la procédure *lister_profondeur_p(s,k)*.
11: **finsi**
12: **finsi**
13: **fin pour**
14: **fin pour**
15: Parcourir la liste topologique obtenue et fusionner, si possible, deux à deux les sommets consécutifs.

Remarque : Cet algorithme montre un exemple d'algorithme de fusion, il n'est pas toujours optimal en nombre de nœuds fusionnés. Les deux procédures *lister_profondeur_non_p* et *lister_profondeur_p* présentent un inconvénient, puisqu'elles pourraient se bloquer à un niveau donné, alors qu'il est possible de fusionner des nœuds dans des niveaux supérieurs. Cet inconvénient peut être levé si l'on ajoute une boucle à ces deux procédures qui prend en compte cet aspect.

La section qui suit présente un exemple d'application de la méthode de parallélisation. Cette méthode peut également être appliquée aux procédures *linward* et *backward*, ce qui donne une parallélisation complète de tous les calculs faits à chaque pas de temps. Cependant le *backward* a certaines spécificités que l'on discutera à la section 6.4.

6.2.3 Exemple de l'acoustique marine

On présente un exemple du déroulement de la méthode de parallélisation sur l'une des applications réelles citées en section 3.5.5. Il s'agit d'une application YAO sur l'acoustique marine, qui admet un nombre limité de *fonctions de base* F_i et qui a un espace de calcul 2D. Nous illustrons l'évolution de la méthode sur cette application. Nous utilisons les mêmes noms des *fonctions*

de base que [Berrada, 2008][7], un lecteur intéressé peut reconnaître les fonctions de l'application. Cette application permet d'assimiler des observations réelles de pression acoustique afin de récupérer certains paramètres géoacoustiques, comme la célérité, la densité, l'atténuation. Dans la référence précédente, les 9 *fonctions de base* utilisées sont notées $n(z)$, *C*, *B*, *bet*, *gam*, *R*, X_t, ψ et ψ_{fd}. Afin de simplifier les notations nous les noterons respectivement $F_1, \ldots, F_9$. La figure 6.6 montre le *GDR* composé par les 9 *fonctions de base* et les arcs marqués par les vecteurs distance issus des directives *ctin*. Dans cette figure, les CFC sont en pointillés et numérotées de 1 à 6. Les directives *order* fournies par l'utilisateur sont données en figure 6.7 ; il s'agit d'un bloc de directives *order* dont la boucle extérieure est relative à l'axe *i* ascendant ($l = i$). Après le calcul du graphe $G_{F/X}$ et sa décomposition selon l'algorithme de décomposition, nous marquons chaque sommet par p et $\bar{p}$. Nous procédons ensuite à la mise par niveaux, comme présenté en figure 6.9 ; le cercle simple représente un sommet parallélisable (p) et le double cercle représente un sommet non parallélisable ($\bar{p}$). Seule la CFC n°5 est non parallélisable, du fait de l'existence de $d_i = -1$ associé aux connexions (F_6, F_8).

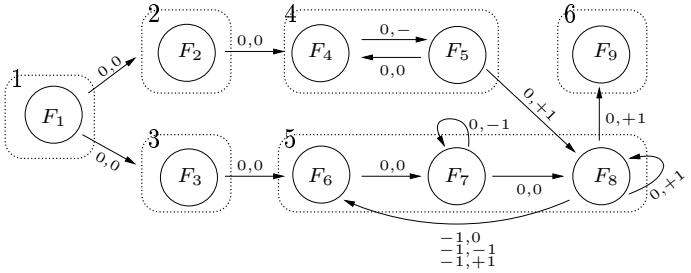

FIG. 6.6 – *GDR* de l'acoustique marine : les lignes en pointillés sont les CFC numérotées de 1 à 6. Les arcs entre F_8 et F_6 sont représentés par un arc avec 3 valuations.

On applique alors la méthode de fusion proposée. La figure 6.10 montre la fusion des sommets 2 et 3 marqués par p dans un nouveau sommet marqué par p appelé (2,3). Ceci correspond à la ligne 2 de l'algorithme 9 et permet donc la fusion des sommets par niveau. On traverse alors le $G_{F/X}$ obtenu (figure 6.10) dans l'ordre croissant de ces niveaux ; à ce stade, il y a un ou deux sommets par niveau (dans le cas particulier de cet exemple, il y a un sommet par niveau). On cherche à ordonnancer les nœuds pour les fusionner dans un deuxième temps. On part du sommet 1 et on lance la procédure *lister_profondeur_p(s,k)* avec $s = 1$ et $k = 1$. Cette procédure marque le sommet *s* et lance la procédure *lister_profondeur_p(d,k+1)* avec *d* qui correspond au sommet (2,3) qui est ensuite marqué. On lance la même procédure au niveau 3 pour $d = 4$. La traversée en profondeur s'arrête après avoir marqué 4 car les sommets 4 et 5 ne sont pas de même type (respectivement p et $\bar{p}$). Etant donné que les sommets des 3 premiers niveaux sont tous marqués, la traversée par ordre croissant des niveaux (ligne 4 de l'algorithme 9) reprend au niveau 4. La procédure *lister_profondeur_non_p(*5,4*)* marque le sommet 5 et s'arrête ensuite car le successeur (le sommet 6) est de type p. Le sommet 6 est ensuite marqué, ce qui fait terminer la boucle *for* de la ligne 4 de l'algorithme de fusion. A ce stade on a obtenu une suite de sommets qui correspond

[7] Une version plus détaillée de cette application est présentée dans [Badran *et al.*, 2008, Hermand *et al.*, 2006].

order YA1
  order YA2
    F_1 F_3 F_2 F_5 F_4 F_6 F_7
  order YB2
    F_8 F_9

FIG. 6.7 – Directives *order* définies par l'utilisateur pour l'application de l'acoustique marine.

order YA1 (**parallèle**)
  order YA2
    F_1 F_3 F_2 F_5 F_4

order YA1 (**non parallèle**)
  order YA2
    F_6 F_7
  order YB2
    F_8

order YA1 (**parallèle**)
  order YA2
    F_9

FIG. 6.8 – Directives *order* issues de la méthode de parallélisation pour l'application de l'acoustique marine.

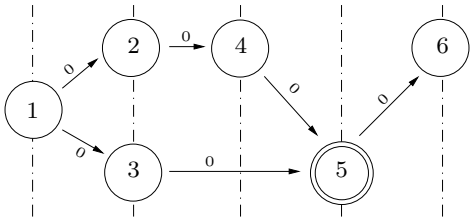

FIG. 6.9 – Le graphe $G_{F/X}$, après avoir fait la mise en niveaux et marqué tous les sommets. Le double cercle représente un sommet non parallélisable.

à un ordre topologique : 1, 2, 3, 4, 5, 6.

La dernière étape de l'algorithme 9 est d'effectuer la fusion des sommets de cette liste. Ainsi, les deux sommets p, 1 et (2,3), peuvent être fusionnés dans un nouveau sommet appelé (1,2,3), qui est lui aussi parallèle. La même opération est faite sur les groupes 1, 2, 3 et 4 comme illustré en figure 6.11. L'algorithme se termine, car il n'est pas en mesure de fusionner d'avantage. Le tri topologique trouvé est traduit en un tri des *fonctions de base*. Le tri résultat respecte, dans chacun des trois sommets obtenus, l'ordonnancement donné par l'utilisateur et correspond à : [F_1 F_3 F_2 F_5 F_4], [F_6 F_7 F_8], [F_9] ou bien [n(z) B C gam bet], [R X_t ψ], [ψ_{fd}]. La décomposition finale des directives *order* est donnée en figure 6.8. Cet algorithme donne une décomposition de domaine sur l'axe relatif à la boucle extérieure $l = i$, qui peut ensuite être automatiquement parallélisée dans le code généré, grâce à des directives OpenMP par exemple.

L'écriture des directives *order* parallèles de la figure 6.8 est faite dans cet exemple de façon très simpliste. Afin de ne pas violer l'hypothèse de cohérence, les *fonctions de base* au sein d'un

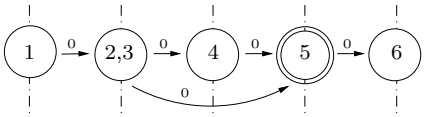

FIG. 6.10 – Fusion des sommets 2 et 3 dans un nouveau sommet *p* appelé (2,3).

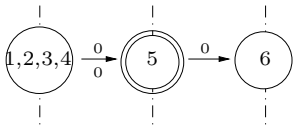

FIG. 6.11 – Fusion des trois sommets 1, (2,3) et 4 dans un nouveau sommet marqué par *p* appelé (1,2,3,4).

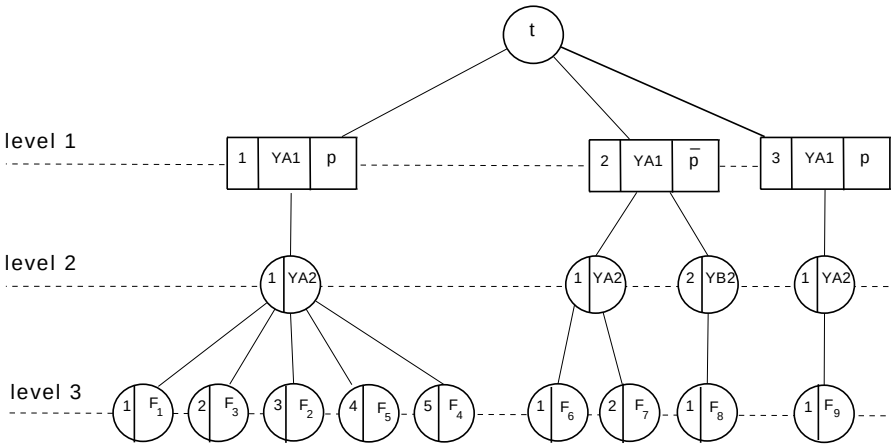

FIG. 6.12 – Arbre de directives *order* issu du processus de parallélisation de l'application de l'acoustique marine.

même groupe sont écrites en gardant le même ordre fourni par l'utilisateur. De même, l'ordre des axes et leur sens, qui sont fournis par l'utilisateur, sont conservés. Il est facile de s'inspirer des procédures du chapitre 5 pour générer l'arbre de directives *order*. De plus, les noeuds du premier niveau de cet arbre doivent être marqués par p ou $\overline{p}$ à l'aide d'une variable booléenne spécifique. En figure 6.12, nous donnons l'arbre de l'acoustique marine, qui correspond aux directives de la figure 6.8.

6.3 Parallélisation lors de la génération automatique des directives *order*

Dans la section 6.2, nous avons abordé le problème de la parallélisation automatique de boucles, en supposant que l'on dispose de directives *order* valables fournies par un utilisateur. Il s'agissait alors d'analyser chaque bloc principal, relativement à l'axe de sa boucle la plus extérieure, afin de détecter les parties qui sont parallélisables au sein de ce bloc. D'autre part, au chapitre précédent (chapitre 5), nous avons présenté une méthode pour la génération automatique des directives *order*. Cette méthode, si elle ne détecte pas d'anomalies et si elle n'est pas suivie

de fusions de boucles, permet de générer autant de blocs de directives *order* que de CFC du *GDR*. Elle associe donc, à chaque CFC X_i, une liste de couples, chaque couple étant formé par $(l, sens)$, où $l \in \{1,2,3\}$ et $sens \in \{A,B,X\}$ et définit une boucle extérieure possible qui permet de calculer les modules relatifs aux *fonctions de base* de la CFC X_i. Nous rappelons que A (respectivement B) représente le sens ascendant (respectivement descendant) et correspond au cas où toutes les distances d_l, relatives aux arcs du sous-graphe du *GDR* restreint à X_i, sont négatives ou nulles, avec au moins une strictement négative (respectivement ≥ 0 avec au moins une > 0). Par contre X représente le cas où les deux sens sont possibles et correspond au cas où tous les d_l sont nuls. Ainsi, en tenant compte de la règle 5 présentée à la section précédente, l'existence d'un couple de type (l,X), dans la liste associée à la CFC X_i, permet d'en déduire que la boucle extérieure relative à l'axe l est parallélisable. Dans ces conditions, afin de rendre le maximum de blocs élémentaires parallélisables, il serait intéressant de choisir, pour chaque CFC X_i, la boucle selon l'axe l comme boucle extérieure. Toute CFC X_i admettant au moins un couple du type (l,X) sera dite parallélisable et étiquetée par p. On lui associe alors la sous-liste, issue de sa liste initiale, formée uniquement par les couples de type (l,X), qui par définition est non vide. Par contre, une CFC qui ne contient pas dans sa liste un couple de type (l,X), sera considérée non parallélisable. On lui attribue alors l'étiquette $\bar{p}$ et on lui associe sa liste initiale. Il est alors possible de procéder à des fusions. La fusion de deux blocs élémentaires ne doit pas, dans ce cas, détruire le caractère parallélisable d'un bloc donné. Pour cela, on ne doit pas autoriser la fusion de deux blocs étiquetés respectivement par p et $\bar{p}$. Ainsi la fusion doit se faire entre blocs ayant les mêmes étiquettes.

La fusion de blocs étiquetés par p se réalise sur le graphe GDR_r ; à chaque sommet est associé sa liste restreinte aux couples de type (l,X). Il est alors possible d'appliquer, selon les cas, une procédure de fusion : par largeur d'abord, par profondeur d'abord ou bien une procédure mixte. Le résultat de la fusion de deux sommets de type p est un sommet de type p, sa liste sera formée par l'intersection des différentes listes des sommets fusionnées avec les listes des arcs du GDR_r restreint aux sommets fusionnés.

La fusion de blocs étiquetés par $\bar{p}$ se réalise en appliquant une procédure de fusion, sur le GDR_r en leur attribuant leur liste initiale. Le résultat de la fusion de deux sommets de type $\bar{p}$ est un sommet de type $\bar{p}$.

Ainsi, avec ce processus de fusion, on détermine le premier niveau de l'arbre de directives *order*, où l'on associe, à chaque nœud de ce niveau, la liste de toutes les boucles extérieures possibles, qui sont caractérisées par les couples (l,X).

Remarque : Dans ce qui précède, nous avons proposé une méthode de génération automatique de boucles parallèles. Elle consiste à associer, s'il n'y a pas d'anomalies, des boucles possibles pour chaque CFC du graphe réduit. Nous avons aussi vu que certaines CFC ne sont pas parallélisables et d'autres le sont. A chaque CFC X_i parallélisable, on lui associe tous les axes qui permettent sa parallélisation, il s'agit de tous les axes l pour lesquels les distances d_l, correspondant au sous-graphe G_{X_i} (le graphe réduit restreint à X_i), sont nulles. Nous avons noté (l,X) cette situation. D'autre part, pour une telle boucle extérieure d'axe l, la génération d'une boucle qui lui est imbriquée revient à considérer le graphe partiel de G_{X_i} qui consiste à garder uniquement

les arcs pour lesquels $d_l = 0$. Or, dans le cas d'un axe l (qui permet une parallélisation de X_i), ce graphe partiel correspond au graphe G_{X_i} qui est fortement connexe. Ainsi, il est possible de générer une boucle imbriquée dans celle d'axe l. Cette seconde boucle, pour qu'elle soit parallélisable par décomposition de domaine relativement à son axe m, doit vérifier la propriété suivante : toutes les distances d_m, relatives au sous graphe G_{X_i}, sont nulles. Ainsi, le couple (m,X) appartient à la liste associée X_i. En conclusion, étant donnée une CFC X_i parallélisable et la liste des couples (l,X) qui lui est associée, le nombre d'éléments de cette liste permet de générer des décompositions de domaines de plus en plus fines. Ainsi, si cette liste contient un seul élément (l,X), il est alors possible de prendre comme boucle extérieure celle qui est relative à l et de procéder à une décomposition de domaine relativement à l. Si la liste contient deux éléments (l,X) et (m,X), il est alors possible de prendre les deux boucles imbriquées extérieures relativement aux axes l et m et de procéder à des décompositions de domaines relativement à ces deux axes. De même, si la liste contient trois éléments, il est possible de considérer des boucles imbriquées relativement à ces trois axes et de procéder à des décompositions de domaines selon ces trois axes.

6.4 Génération et parallélisation de la procédure *backward*

Les procédures *forward* et *linward* traversent le même graphe modulaire, ells sont donc similaires et la même méthode de parallélisation peut être appliquée aux deux procédures. La procédure *backward* traverse le graphe inverse du graphe modulaire, elle parcourt les points des espaces de calcul en suivant l'inverse d'un ordre topologique. Autrement dit, pour un ordre topologique défini par des directives *order* (par un utilisateur ou générées automatiquement), la traversée en sens opposé se fait en appliquant les étapes de transformation suivantes :

- inverser l'ordre des blocs de directives *order*,
- pour chaque bloc, conserver l'ordre d'imbrication des boucles,
- pour chaque boucle, de la plus extérieure à la plus intérieure, inverser le sens de parcours suivant son axe et inverser l'ordre de ses instructions (boucles ou *fonctions de bases*). Substituer le calcul des *fonctions de base* par le calcul de leurs adjoints.

L'exécution de ces étapes permet de faire une traversée inverse du graphe, ce qui correspond aussi à une traversée inverse de la grille de calcul. L'inversion des directives *order* de l'exemple 6.1b est :

```
order YA1
   order YB2
      adjoint(F4) adjoint(F2)

order YB1
   order YB2
      adjoint(F3) adjoint(F1)
```

Un deuxième exemple d'inversion de directives *order* sur deux *fonctions de base* rattachées à des espaces de dimensions différentes est donné en figure 6.13. Dans cet exemple, on montre un parcours en avant des directives *order* et son parcours en arrière équivalent. Ce dernier illustre aussi le même parcours, en termes de traversée et de traversée inverse de la grille de calcul.

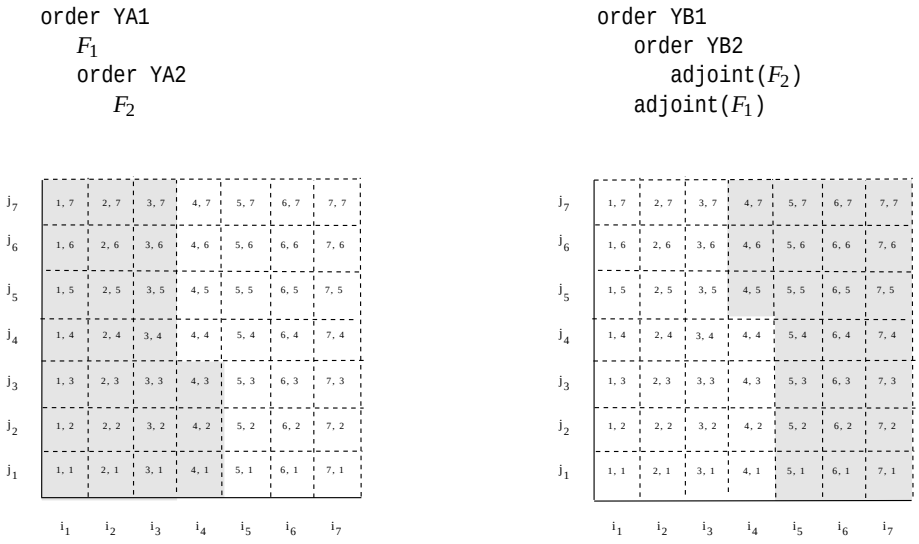

FIG. 6.13 – Exemple de directives *order* et son inversion. La *fonction de base* F_1 est rattachée à un espace 1D et la fonction F_2 à un espace 2D. Les deux parcours (en foncé) sur la grille de calcul montrent l'inversion par rapport à l'espace 2D propre à F_2 mais également, en prenant une ligne quelconque de la grille, l'inversion par rapport à l'espace 1D.

La rétropropagation sur le graphe modulaire peut être vue comme une propagation sur son graphe inverse. Celui-ci peut être considéré comme étant généré par un *GDR* qui correspond au graphe inverse du *GDR* initial. Etant donné que les boucles extérieures restent les mêmes et que les CFC des deux graphes inverses coïncident, l'application de la règle 5 permet de déduire que la même méthode permet de paralléliser les procédures *forward* et *backward*. D'autre part, il est facile de voir que les règles de fusion entre blocs élémentaires de la procédure *forward*, restent valables pour la procédure *backward*. Ainsi, les directives *order* parallèles obtenues par la décomposition/fusion de la procédure *forward* peuvent être intégralement conservées pour la procédure *backward*. Il suffit alors de leur appliquer les étapes de transformation définies ci-dessus. Le générateur YAO permet de générer automatiquement la rétropropagation des calculs dans la procédure *backward*. YAO ne génère pas un deuxième graphe, le graphe inverse, il se limite à utiliser de façon différente le même graphe. Par contre, la parallélisation de la procédure *backward* par la méthode vue à la section 6.2.2 présente une difficulté supplémentaire de synchronisation. Cette méthode élimine toutes les dépendances de *flot* (écriture et puis lecture) entre *threads* dans un bloc de directives *order*. Outre les dépendances de type *flot*, la procédure *backward* fait apparaître un nouveau type de dépendance, connu sous le nom de dépendance de *sortie*[8]. Celle-ci impose d'ajouter d'autres synchronisations au processus de parallélisation des procédures de propagation en avant. Dans la suite, on présente le pseudo-code de la procédure *backward*. Ce dernier permet de

[8]La dépendance de sortie fait partie des trois types de dépendances : *flot*, *anti* et *sortie*. Ces dépendances ont été introduites par [Bernstein, 1966]. Une dépendance de sortie est décrite, intuitivement, par deux opérations d'écriture de la même case mémoire.

visualiser la synchronisation qu'il faut ajouter, ce afin de garantir une exécution parallèle correcte. Les caractéristiques de cette synchronisation sont aussi illustrées.

La figure 6.14 montre le pseudo-code de la procédure *backward* issu des directives de la figure 6.1. Une boucle globale, non représentée dans la figure, permet de traverser les pas de temps dans un ordre descendant $t+2$, $t+1$, t, etc., et constitue la boucle qui englobe toutes les autres. On note F_s la *fonction de base* qui est calculée et F_d une *fonction de base* qui succède à F_s. Ce pseudo-code correspond à deux blocs de boucles, il permet de réaliser quatre sous-blocs de calcul, un pour chaque *fonction de base* de la figure 6.1. Chaque sous-bloc permet de faire le calcul du modèle adjoint pour une *fonction de base* particulière et le rétropropager dans le graphe modulaire. Ceci a été décrit dans l'algorithme 3, qui présente la procédure *backward* (chapitre 3). On suppose avoir exécuté la procédure *forward* au préalable.

Pour chacun des quatre sous-blocs, le pseudo-code de la figure 6.14 montre quatre phases : l'initialisation du point de référence, le calcul de la matrice jacobienne, le calcul du modèle adjoint (le produit) et la rétropropagation (la mise à jour). La matrice jacobienne de la *fonction de base* F_s est calculée pour le vecteur $\mathbf{x}_s$ qui est obtenu lors de la procédure *forward* et représente le point de référence. Ce point est le vecteur *input* du pseudo-code, qui est la sortie des modules prédécesseurs. Une fois que le vecteur *input* est initialisé, on appelle la fonction *backward* locale, qui calcule la matrice jacobienne de F_s au point *input*. Ce calcul est stocké dans la matrice *jacobian* et correspond à la matrice $\mathbf{F}_s$ de l'algorithme 3 à la section 3.2.2. Avant la rétropropagation, nous calculons le produit entre la matrice jacobienne transposée et le gradient en provenance des modules successeurs (les modules issus des *fonctions de base* F_d) : $output = transpose\_product(F_s(i,j,t).gradient, jacobian)$. Ce calcul correspond au calcul du modèle adjoint de F_s et assure donc le calcul des coefficients $\boldsymbol{\alpha}_s = \mathbf{F}_s^T \boldsymbol{\beta}_s$ de l'algorithme 3 à la section 3.2.2. Le gradient est stocké dans l'attribut *gradient*, de l'objet concerné, dans la phase de rétropropagation (ceci correspond au fait d'avoir rétropropagé les $\boldsymbol{\alpha}_s$ en les affectant aux différentes composantes des $\boldsymbol{\beta}_j$). Cette rétropropagation est la dernière phase que l'on exécute avant de passer au sous-bloc de calcul suivant (le traitement de la *fonction de base* suivante). Ceci est fait en faisant une mise à jour des modules prédécesseurs de F_s. Ainsi, à titre d'exemple, l'instruction de mise à jour $F_3(i,j,t).gradient += output[0]$ du pseudo-code permet de rétropropager le calcul contenu par le vecteur *output* à l'attribut *gradient* de l'objet F_3. Elle correspond à la somme de l'étape 2.b de l'algorithme 3 ($\boldsymbol{\alpha}_s$ est représenté par le vecteur *output* dans le pseudo-code).

Dans la phase de mise à jour de l'attribut *gradient*, plus d'un *thread* peut mettre à jour une même donnée en même temps, conduisant à une éventuelle corruption de données (*race condition*). En fonction des d_l, où $l \in \{i,j,k\}$ est l'axe de la décomposition de domaine, et de d_t le délai temporel, la mise à jour d'une même donnée peut être faite par deux *threads* différents. Le *GDR* de la figure 6.2, tel qu'il a été défini à la section précédente, n'est pas suffisant pour faire une analyse complète de la détection de ce type de *race conditions* puisqu'il ne tient compte que des *ctin* avec des délais $d_t = 0$. En effet les *ctin* qui ont délai temporel $d_t < 0$, exclues du *GDR*, jouent un rôle important dans la détection de ces *race conditions* de l'instruction de mise à jour. Cette situation est illustrée dans la figure 6.15. Cette figure représente la rétropropagation des modules $F_1(i-1,j,t-1)$ et $F_1(i,j-1,t-1)$ de l'exemple 6.14.

```
loop i ascendant
  loop j descendant
    // Initialisation du point de référence input.
    // Il s'agit des entrées de la méthode backward du module F4(i,j,t).
    input[0]=F3(i,j,t)
    input[1]=F2(i,j+1,t)
    // Calcul de la matrice jacobienne de F4(i,j,t).
    jacobian=F4(i,j,t).backward(input)
    // Produit entre la matrice jacobienne de F4(i,j,t) transposée et le gradient.
    output=transpose_product(F4(i,j,t).gradient, jacobian)
    // Rétropropagation du gradient de F4(i,j,t).
    F3(i,j,t).gradient+=output[0]
    F2(i,j+1,t).gradient+=output[1]

    input[0]=F1(i,j+1,t)
    input[1]=F3(i-1,j+1,t)
    input[2]=F3(i,j,t-1)
    input[3]=F4(i+1,j,t)
    jacobian=F2(i,j,t).backward(input)
    output=transpose_product(F2(i,j,t).gradient, jacobian)
    F1(i,j+1,t).gradient+=output[0]
    F3(i-1,j+1,t).gradient+=output[1]
    F3(i,j,t-1).gradient+=output[2]
    F4(i+1,j,t).gradient+=output[3]

loop i descendant
  loop j descendant
    input[0]=F1(i-1,j,t-1)
    jacobian=F3(i,j,t).backward(input)
    output=transpose_product(F3(i,j,t).gradient, jacobian)
    F1(i-1,j,t-1).gradient+=output[0]

    input[0]=F1(i,j-1,t-1)
    jacobian=F1(i,j,t).backward(input)
    output=transpose_product(F1(i,j,t).gradient, jacobian)
    F1(i,j-1,t-1).gradient+=output[0]
```

FIG. 6.14 – Génération de la procédure *backward* par le générateur YAO à partir des directives 6.1a et 6.1b. L'ordonnancement des *fonctions de base* et les sens des axes sont inversés par rapport à la procédure *forward*. Les méthodes *backward* appelées dans ce pseudo-code sont les fonctions *backward* locales à chaque *fonction de base*. *input* et *output* sont des vecteurs, *jacobian* est la matrice jacobienne. *transpose_product* est une fonction qui permet de faire le produit de la matrice jacobienne transposée et du vecteur *gradient*. Ce dernier est un attribut de l'objet représentant la *fonction de base*.

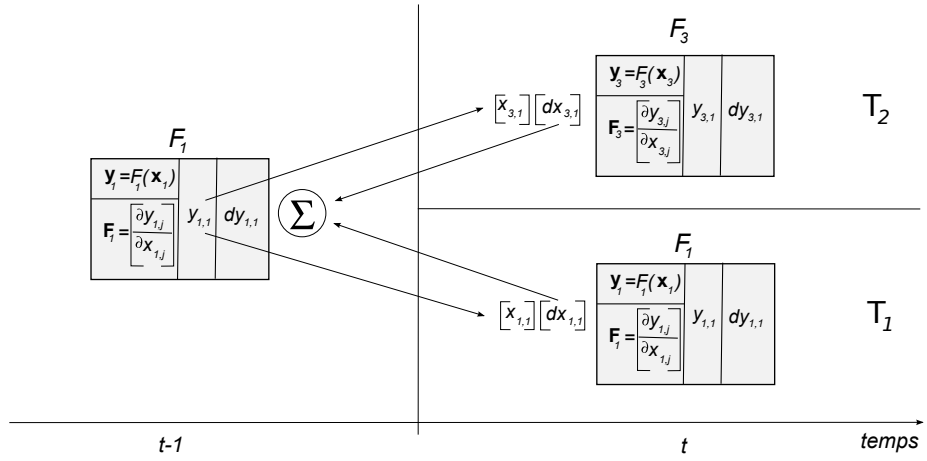

FIG. 6.15 – *Race condition* sur deux mises à jour dans la procédure *backward* de l'exemple 6.14. Les deux connexions représentent un transfert de données entre deux pas de temps t et $t-1$, par les *threads* T_1 et T_2, du module F_1 vers F_1 et F_3 dans la phase de propagation. C'est dans la phase de rétropropagation que la *race condition* fait son apparition.

Les deux mises à jour relatives à ces modules sont contenues dans le deuxième bloc de directives *order*, elles sont :

```
(1)   F1(i-1,j,t-1).gradient+=output[0]       (mise à jour à partir de F3)
(2)   F1(i,j-1,t-1).gradient+=output[0]       (mise à jour à partir de F1)
```

Dans ce qui précède, nous n'avons pas opté pour une parallélisation relative à l'axe du temps (t). La méthode présentée permet, à chaque pas de temps, une décomposition de domaine relativement à un axe l choisi. Ainsi, au temps t, tous les modules du temps $t-1$ sont déjà calculés. Si nous supposons que les deux *threads* T_1 et T_2 correspondent à une décomposition de domaine relativement à l'axe i, et que T_1 et T_2 sont associés aux variations de i respectivement dans $[1,M_1]$ et $[M_1+1,M_2]$ ($M_1,M_2 \in \mathbb{N}$ et $1 < M_1 < M_2 \leq N_l$, où N_l est la taille de l'axe l). Le calcul du β du module $F_1(M_1,j,t-1)$ se réalise par la somme des α des deux modules $F_3(M_1+1,j,t)$ et $F_1(M_1,j+1,t)$. Or, ces deux modules au temps t sont calculés, séparément, par les deux *threads* T_1 et T_2. Cet exemple montre que des calculs concernant une même case mémoire peuvent être réalisés par deux *threads* différents, ce qui nécessite, dans ce cas, une synchronisation afin d'éviter que ces calculs se réalisent simultanément. La figure 6.15 illustre ce phénomène d'écriture d'une même case mémoire par deux *threads* différents. En outre, cette figure met en évidence que les numéros de sortie, présents dans la spécification des *ctin*, sont un autre élément à prendre en considération pour la détection de cette écriture simultanée (dans les *ctin* de la figure 6.1, on avait supposé que toutes les *fonctions de base* ont une seule sortie). Mais, en général, les *fonctions de base* ont plusieurs sorties. Dans la rétropropagation, l'expression $\beta_{sj} = \sum_{(d,i)\in SUCCM(s,j)} \alpha_{di}$ (étape 2.b de l'algorithme 3 de la section 3.2.2) réalise la somme à partir des indices des successeurs de

la $j^{ième}$ sortie de la *fonction de base* F_s. La corruption des données dépend du fait que l'écriture se réalise à la même $j^{ième}$ sortie de F_s.

En conclusion, les mises à jour qui nécessitent des synchronisations sont caractérisées par deux *ctin* liant une même sortie d'une même *fonction de base* source F_s à deux *fonctions de base* destination F_d et F'_d, situées dans un même bloc de directives *order*, qui admet l comme axe de sa boucle extérieure. Si nous notons d_l, d'_l et d_t, d'_t respectivement les deux distances de l'axe l et les deux délais temporels relatifs aux deux *ctin*, nous supposons que :

- $d_l \neq d'_l$,
- $d_t = d'_t$ mais différents de 0.

Il est alors possible de vérifier, sous ces deux conditions, que pour certaines valeurs de l, les modules correspondants à F_d et F'_d sont dans deux *threads* différents et réalisent séparément des mises à jour sur une même case mémoire relative à un module asocié à F_s.

Ces instructions de mise à jour (dépendance de sortie) obligent à synchroniser les *threads* dans certaines parties de la région parallèle, ce qui pourrait avoir de graves répercussions sur les performances. Ces synchronisations ne peuvent pas être supprimées, comme pour la parallélisation des procédures de propagation en avant de la section 6.2. La méthodologie dans cette section était de manipuler les directives *order* (les boucles imbriquées) pour minimiser les dépendances de *flot* et rendre ainsi parallélisable un bloc de directives (ou une partie de celui-ci) si possible. En revanche, dans la procédure *backward*, on n'essaie pas de réduire les dépendances de *sortie*. Il s'agit de les répérer et d'utiliser un moyen technologique pour les éviter. La panoplie d'instructions OpenMP fournit des moyens pour limiter les ralentissements dus aux synchronisations des opérations de mise à jour. Etant donné qu'il s'agit de simples instructions, la directive *atomic* [Chapman *et al.*, 2007] semble être une bonne solution d'efficacité. En effet, cette directive permet d'isoler une simple instruction du contexte parallèle et garantit qu'elle sera exécutée sans interférences. Elle évite que deux *threads* écrivent simultanément sur une même case mémoire. Ceci empêche la corruption de données qui pourrait être provoquée par la mise à jour. Le générateur YAO, au moment de l'écriture de la procédure *backward*, analyse les caractéristiques de chaque mise à jour selon les conditions précédemment énoncées. S'il trouve une condition de synchronisation, il ajoute une directive *atomic*. Des tests ont démontré que, avec l'ajout de ces directives OpenMP, le calcul réalisé par la procédure *backward* est correct et les performances ne sont pas sensiblement impactées. En effet, dans le cadre de YAO, la dépendance de *flot* oblige à respecter un certain ordre de calcul, tandis que la dépendance de sortie oblige à ne pas écrire simultanément sur une même variable partagée. Il est intuitif que les dépendances de premier type peuvent être bien plus coûteuses, en termes de performance, que celles du deuxième type.

Remarque : Déboguer un code parallèle n'est pas une tâche simple. En général, il ne suffit pas de comparer les sorties du code séquentiel et parallèle. En effet, si les sorties ne sont pas exactement les mêmes, ceci ne veut pas dire que le code parallèle est nécessairement faux. Il est bien connu que la somme des nombres à virgule flottante sur un ordinateur est, en général, non-commutative. Cela signifie que $s = a+b+c$ peut être différent de $s = c+a+b$ à cause de la précision machine. La différence est, en général, faible mais si ces petites différences sont multipliées plusieurs fois dans le produit de la matrice jacobienne par le vecteur gradient de la procédure *backward*, l'écart

peut être significatif. Dans tous les cas, même si les deux sommes ne sont pas égales, il n'y en a pas une préférable à l'autre. Dans la procédure *backward*, plusieurs produits et mises à jour ont lieu. Dans la vérification de l'exactitude du code parallèle, nous devons garder à l'esprit ces règles numériques de base. L'assimilation variationnelle de données cherche à minimiser une fonction de coût par une méthode de gradient, le gradient indique alors une direction à suivre dans la minimisation. Or, quand elle subit de légères modifications, cette direction reste valable pour la minimisation. Au dernier chapitre (chapitre 7), nous discuterons de la perspective d'intégrer une bibliothèque qui permet de tester la précision numérique des applications.

Chapitre 7

Conclusion et perspectives

Les méthodes d'assimilation de données 4D-Var et leur évolution posent des problèmes algorithmiques de plus en plus complexes ; dans la pratique, les difficultés liées à leur implémentation constituent l'obstacle majeur quant à leur utilisation. Leur apport serait plus important si l'on disposait d'outils logiciels facilitant leur mise œuvre. Plusieurs approches logiciels ont été proposées et répondent, en général, à des aspects spécifiques. Le logiciel YAO, qui est basé sur la notion de graphe modulaire, offre un cadre permettant d'aborder l'ensemble des fonctionnalités nécessaires à la mise en œuvre de sessions d'AD 4D-Var. La version actuelle de YAO est opérationnelle sur des applications de taille moyenne. L'objectif à moyen terme est de faire évoluer cette version vers une plateforme complète et générale, qui doit être capable de traiter des applications réelles de taille importante et nécessitant un formalisme complexe. Comme nous l'avons déjà signalé, l'objectif de cette thèse est de préparer l'évolution de la version actuelle de YAO vers cet objectif.

Le travail de reformalisation, de synthèse et de développement algorithmique d'automatisation de tâches, que nous avons entrepris dans cette thèse, a permis de donner des réponses immédiates et opérationnelles et de faire le lien avec des domaines actuels de recherche en informatique. Un dialogue est maintenant ouvert avec des spécialistes modélisateurs, des chercheurs en informatique (parallélisation automatique à mémoire partagée et distribuée, précision numérique, ...), il a permis de proposer un projet *Emergence* avec le LIP6 qui a été accepté en juin 2010. Un projet ANR, qui se situe dans le cadre de l'appel 2011 sur les *Modèles Numériques*, est en cours de rédaction. Ce projet va permettre de faire collaborer trois laboratoires de l'IPSL, deux de l'INRIA, une équipe du CNAM et une du LIP6. Nous discutons dans la suite de cette section des perspectives de ce travail.

Au cours de cette thèse, nous avons, en premier lieu, repris les spécifications et les directives de la version actuelle de YAO, et nous les avons formalisées d'une manière plus générale. Nous avons ensuite présenté, à partir de l'analyse des dépendances de calcul entre *fonctions de base* (les dépendances étant définies par le *graphe de dépendance réduit GDR*), plusieurs types d'algorithmes destinés à vérifier la cohérence de certaines directives et leur automatisation, notamment celles qui définissent le parcours de l'espace de calcul et la parallélisation automatique en mémoire partagée du code généré.

Le travail de formalisation que nous avons entrepris a permis de pointer certaines lacunes, notamment au niveau des références absolues des directives *ctin*. En effet, celles-ci ont été introduites afin de traiter des cas particuliers d'initialisation. Mais, telles qu'elles sont définies dans YAO, elles autorisent des déclarations qui permettent de représenter des situations qui vont bien au-delà de l'objectif initial (initialisation). Ceci pourrait correspondre à des cas non réalistes dans la pratique. La directive *ctin* permet de transmettre des données d'un sous-espace affine vers un espace. Par contre, elle ne permet pas la transmission de données entre deux sous-espaces affines ou d'un espace à un sous-espace affine. La notion d'espace associé aux *fonctions de base*, qui existe dans cette version de YAO, et que nous avons formalisée dans ce document, doit être reprise et améliorée, afin de généraliser la modélisation dans YAO. Il faut aussi introduire les opérateurs entre espaces différents afin de pouvoir traiter le couplage de modèle et les problèmes de changement d'échelle de discrétisation. Mais ce travail de généralisation nécessaire à l'évolution de YAO doit être repris dans le cadre d'un travail collaboratif, en particulier avec des modélisateurs numériciens provenant de domaines d'application variés. Il s'agit de bien cibler les spécifications et les fonctionnalités à introduire dans YAO, afin de bien définir les concepts et directives qui correspondent à des situations conséquentes et réalistes.

Rendre YAO opérationnel sur des applications importantes nécessite une organisation du calcul d'un grand nombre de modules, ce qui n'est pas une tâche facile pour l'utilisateur. La génération automatique des ordres de parcours des espaces de calcul constitue un outil d'aide important. Mais, comme nous l'avons vu, les algorithmes que nous avons proposés permettent la génération de plusieurs ordres possibles. D'autre part, nous avons proposé et discuté des stratégies de décomposition et de fusion de boucles lors de leur génération et leur parallélisation. Ces choix et ces stratégies donnent la possibilité d'optimiser le code en suivant des critères donnés, qui seront définis en fonction de l'architecture de la machine. Ainsi, le développement futur de YAO devrait lui permettre d'intégrer le concept de *modèle machine*. Ce modèle machine est défini généralement par un fichier (par exemple un fichier XML), qui permet de donner des détails architecturaux sur une machine donnée. Ce fichier contient les informations de type : nombre de registres, nombre de mémoires caches (L1, L2, L3) et leur taille, la taille de la mémoire RAM, le type d'architecture (32 ou 64 bits), etc.. Le modèle machine pourrait être un dispositif important pour optimiser le code généré par YAO. A titre d'exemple, une fusion très importante des boucles a pour effet de réduire immédiatement le surcoût dû au fonctionnement des boucles, mais en même temps pourrait empêcher un fonctionnement optimal des mémoires caches et, encore à plus bas niveau, des registres de l'unité de traitement. Une mauvaise utilisation de ces différents niveaux de mémorisation se répercute directement sur les performances globales de l'application. La liberté donnée par cette thèse de choisir la façon de fusionner les boucles permettra de faire un choix qui gère au mieux ce type de compromis.

Indépendamment du modèle machine, le traitement d'applications de taille importante pose le problème de l'allocation mémoire. En effet, l'application de la procédure *backward* nécessite le stockage des états de tous les modules calculés lors de l'exécution de la procédure *forward*. Nous avons présenté et discuté, dans l'annexe A, la technique dite du *checkpointing*. L'introduction de cette technique, et sa gestion automatique par YAO, nécessite une reformalisation de certaines structures de données de YAO et l'introduction de fonctionnalités supplémentaires.

La parallélisation en mémoire partagée, que nous avons traitée dans cette thèse, correspond au cas où les espaces des boucles sont représentés par des parallélépipèdes rectangles (dans le cas d'espaces 3D) définis par des contraintes du type $1 \leq i \leq N_i$. Les méthodes de parallélisation que nous avons traitées déterminent des parallélisations en gardant cette représentation en parallélépipèdes rectangles. Ce travail doit continuer en se basant sur des *modèles polyédriques* [Bastoul, 2004, Legrand et Robert, 2003, Darte *et al.*, 2000]. Ceux-ci considèrent des espaces de calcul sous la forme de polyèdre convexe (défini par des inéquations linéaires). Les méthodes de parallélisation associées réalisent des transformations linéaires affines sur des polyèdres convexes. L'objectif étant de transformer l'espace de calcul, afin de faire apparaître des parallélisations non triviales. Ce domaine de recherche est assez avancé actuellement et des outils logiciels existent. Ce sont des outils du type *source-to-source* comme, par exemple, le logiciel *PLuTo* [Bondhugula *et al.*, 2008]. *PLuTo* prend en entrée un code d'un langage impératif (C ou Fortran) composé de boucles imbriquées, qui parcourent des polyèdres convexes. Il est alors capable d'analyser ces boucles imbriquées et de créer, par le biais de l'analyse des dépendances, un modèle polyédrique adapté et de générer un code avec des directives OpenMP. L'outil *PLuTo* est formé d'une chaîne de composants, chacun ayant un rôle spécifique et contribuant à la transformation *source-to-source*. Il serait intéressant d'étudier la possibilité d'utiliser *PLuTo* pour la parallélisation automatique des boucles dans YAO. Etant donné que les directives de YAO nous permettent de disposer du graphe de dépendance réduit, il s'agit alors d'interfacer YAO avec certaines composantes de *PluTo*, afin de réaliser une génération de code parallèle, à partir des directives écrites en langage YAO.

Toutefois, les parallélisations en mémoire partagée ont la contrainte de pouvoir tourner sur des architectures matérielles composées de quelques dizaines de cœurs seulement. Or, les grandes applications d'AD 4D-Var doivent tourner sur des milliers de cœurs distribués sur des centaines de nœuds dans les centres de calcul nationaux. Nous souhaitons, dans la suite de ces travaux de thèse, passer à l'échelle des architectures massivement parallèles à mémoire distribuée à l'aide du standard MPI [Pacheco, 1996] (bibliothèque d'échange de messages). Il faut effectivement pour rendre YAO opérationnel et applicable à des problèmes de grandes dimensions rendre les applications générées extensibles à des architectures de parallélisme massif. La problématique de générer automatiquement des codes parallèles à mémoire distribuée est une problématique ouverte en recherche informatique. Le fait d'aborder cette problématique dans le cadre de YAO rendrait ensuite une approche hybride OpenMP/MPI tout à fait naturelle. Cette approche hybride permettrait de faire une décomposition de domaine au niveau de l'espace de calcul global, en distribuant les calculs sur plusieurs nœuds dans une architecture distribuée par le biais de MPI. Chaque pavé issu de la décomposition serait calculé dans un nœud de calcul et ensuite il serait possible de faire une deuxième décomposition de domaine au sein d'un même pavé, en utilisant OpenMP.

Annexes

Annexe A

Checkpointing

Pour de gros modèles numériques l'*elapsed time* d'un programme d'assimilation de données peut être très important. Il y a au moins deux aspects négatifs. Le premier est trivial, la production d'un résultat est très longue et surtout si nous voulons exécuter plus d'un processus d'assimilation, ce qui est souvent le cas. Il faut garder à l'esprit que, lorsque nous faisons tourner un processus d'assimilation, on pourrait faire des erreurs dans l'estimation des paramètres initiaux et il est donc naturel de faire tourner plusieurs fois le programme. Le deuxième aspect est que si l'*elapsed time* est trop grand, la probabilité d'une coupure de courant, d'un débranchement ou d'une panne de courant devient importante au cours de l'exécution du programme. Dans ce cas, le calcul doit être redémarré de zéro (redémarrage à froid) et les jours passés en attente du résultat sont perdus.

La solution historique au redémarrage à froid est l'utilisation d'une technique connue sous le nom de *checkpointing*. Il s'agit d'un procédé de récupération à partir d'une défaillance du système. Un extitcheckpoint est une copie de la mémoire de l'ordinateur qui est périodiquement sauvegardée sur le disque (*snapshot*) avec les valeurs des registres, les dernières instructions exécutées, etc.. En cas de défaillance, le dernier *snapshot* sert de point de récupération. Lorsque la panne est réglée, le programme de redémarrage copie le dernier *snapshot* dans la mémoire et redémarre le calcul à partir du *snapshot* (redémarrage à chaud).

Avec l'augmentation des architectures parallèles l'utilisation de tels programmes parallèles peut être une solution alternative à la technique du *checkpointing* pour les défaillances du système. Un code parallèle réduit l'*elapsed time* et le redémarrage à froid devient moins coûteux.

Le *checkpointing* pour les défaillances des systèmes a été introduit avec l'arrivée des bases de données et dans les systèmes d'exploitation distribués. En général, l'allocation de la mémoire dans les applications d'assimilation de données est un aspect critique. Un emploi différent des techniques de *checkpointing* peut être envisagé dans le domaine de l'assimilation de données, afin de réduire la taille mémoire des applications qui dépasse, parfois, la capacité disponible de la RAM. Cette taille importante de données est due à la nécessité de stocker les valeurs de toutes les *fonctions de base* en tous les points de grille et pour chaque temps t. Ceci, afin d'effectuer l'algorithme de rétropropagation (section 3.2.2). C'est pourquoi, une variante de la technique du *checkpointing* a été introduite, afin de régler ce problème ; elle consiste à stocker moins d'informations en RAM

FIG. A.1 – Sous-graphe G composé par deux *fonctions de base* F_s et F_d connectées par une *connexion de base*.

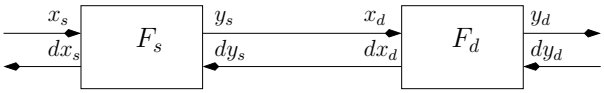

FIG. A.2 – Rétropropagation du calcul de l'adjoint dans le graphe G.

et à refaire les calculs lorsque l'information est nécessaire. L'*elapsed time* est évidemment touché par le recalcul dû à ce type de *checkpointing*. Si l'on a des performances intéressantes sur le temps, nous pouvons par contre dégrader l'*elapsed time* pour obtenir une meilleure allocation mémoire. Le compromis entre ces deux performances est très important dans les applications d'assimilation de données.

YAO s'est avéré être une bonne plateforme de développement pour la réalisation d'applications d'assimilation variationnelle de données. Cependant, toutes les applications développées modélisent des phénomènes qui tournent sur des grilles de calcul (espaces) de moyenne taille. En effet, pour des modèles numériques de dimension très importante, la quantité d'informations à stocker devient rapidement écrasante. Dans cette annexe, on fait une première analyse de l'allocation mémoire dans YAO. Il ne s'agit pas de donner la meilleure solution au problème, ni de le poser sous un formalisme complet. Il s'agit d'une introduction à la problématique, qui laisse la perspective pour un travail futur.

A.1 Stockage des sorties de la procédure *forward*

Le problème se situe au niveau du calcul de la rétropropagation de la procédure *backward*. Ce calcul, pour un module donné, a besoin des sorties des modules prédécesseurs, ces dernières sont calculées durant la procédure *forward*. Ainsi, pour avoir ces informations disponibles lorsque l'on exécute la procédure *backward*, l'application doit les stocker auparavant. Ces procédures sont présentées à la section 3.1. Nous considérons un cas simple de calcul du modèle direct et du modèle adjoint. Dans la figure A.1, nous illustrons un exemple de sous-graphe G d'un *GDR*. G est composé de 2 *fonctions de base* (F_s et F_d) et une *connexion de base* entre elles. Le calcul de F_s dans la procédure *forward* est donné par $y_s = F_s(x_s)$. Le résultat du calcul est propagé en avant et il devient l'entrée de F_d, $y_d = F_d(x_d) = F_d(y_s)$.

Le calcul de l'adjoint de F_d est effectué par la procédure *backward*, en faisant le calcul $dx_d = G_d^t dy_d$, où G_d^t est la transposée de la matrice jacobienne de F_d calculée au point x_d (dans ce cas simple, il s'agit d'un scalaire) et dy_d est la perturbation du vecteur sortie de F_d. La figure A.2 montre la rétropropagation du calcul de l'adjoint dans le graphe G. Afin de calculer la méthode

backward de F_d, x_d doit être disponible (ce qui est aussi la sortie y_s de la *fonction de base* F_s). Cette procédure est calculée en mode inversé, donc en partant de la fin du graphe vers le début de celui-ci. Afin d'avoir x_d au moment de l'exécution de la procédure *backward* de F_d, YAO doit avoir stocké les informations pendant la procédure *forward*. En général, YAO stocke toutes les sorties de toutes les *fonctions de base*, ainsi il dispose des informations dont il a besoin pour le calcul de la procédure *backward*. Ceci revient à stocker une importante quantité d'informations.

Afin de comprendre pourquoi l'allocation mémoire empêche la mise en œuvre de grandes applications, nous allons faire une analyse détaillée en termes d'occupation mémoire du programme généré par YAO. Ceci est l'objectif de la section A.2. La section A.3 présente quelques stratégies d'optimisation mémoire connues dans la littérature sous le nom de *checkpointing* et la section A.4 introduit une possible application au logiciel YAO.

A.2 Analyse de l'allocation mémoire

Cette section fait une analyse qui permet de comprendre quelles sont les variables significatives pour une estimation de l'allocation mémoire dans les applications YAO. On considère un simple fichier de description :

```
...
option O_REAL type
trajectory trajectory_name T
space space_name Nx Ny Nz trajectory_name
module module_name space_name output NB_output input NB_input
...
```

La directive *option* permet de définir des options de génération. *O_REAL* permet de spécifier si le *type* de données traitées est *float* ou *double*. La directive *trajectory* déclare une trajectoire avec T pas de temps. La directive *space* déclare un espace où chaque axe a pour taille N_x, N_y et N_z. La directive *module* déclare une *fonction de base* avec *NB_output* sorties et *NB_input* entrées. On suppose dans cet exemple un nombre de *fonctions de base* égal à *NB_modules*. On considère également le cas simple d'une seule trajectoire et d'un seul espace. Le générateur YAO, à partir de cette description, génère les structures de données C++ statiques suivantes :

```
class module_name {
   type state[T][NB_output];
   type gradient[T][NB_output];
...
};
module_name Ymodule_name[Nx][Ny][Nz];
```

YAO génère une classe appelée *module_name* et une matrice *Ymodule_name* qui contient les objets *module_name* pour chaque point de grille. La classe *module_name* contient 2 matrices, ou plus généralement *NB_matrices* matrices[1]. Le vecteur *state* contient les sorties du module *module_name* pour chaque pas de temps de la trajectoire. Le vecteur *gradient* contient le gradient à propager ou à rétropropager. Les vecteurs *state* et *gradient* sont contenus dans la matrice *Ymodule_name*$[N_x][N_y][N_z]$, ils sont stockés pour tous les points de grille.

Il est ainsi possible de donner une estimation de la mémoire allouée par YAO, en sachant que ces structures de données sont la source principale de l'occupation mémoire. L'allocation mémoire en octets est donnée par l'expression suivante :

$$sizeof(type) * Sx * Sy * Sz * T * NB\_modules * NB\_output * NB\_matrices. \quad \text{(A.1)}$$

Le paramètre *sizeof(type)* dépend du compilateur, mais en général on a 4 octets pour un *float* et 8 octets pour un *double*. On remarque que l'expression ne dépend pas du nombre d'entrées *NB_input*.

L'expression (A.1) inclut tous les paramètres qui permettent d'avoir une estimation précise de l'allocation mémoire de YAO ; dans cette analyse asymptotique, nous ne considérons pas d'autres variables et structures de données qui peuvent être négligées.

A.2.1 L'application de l'acoustique marine

L'application de l'acoustique marine est une application YAO qui a déjà produit des résultats scientifiques intéressants (sections 3.5.5 et 6.2.3). Dans cette application, nous avons un espace de dimension 2 ($N_x = 18000$, $N_y = 603$, $N_z = 1$) et il n'y a pas de temps, l'application tourne sur un pas de temps ($T = 1$) [2]. Comme illustré dans la figure 6.6, le nombre de *fonctions de base* est 9 ($NB\_modules = 9$) et leur nombre de sorties correspond en moyenne au ratio entre les *ctin* et *NB_modules* ($NB\_output = \frac{13}{9} = 1,44$). Le *type* des variables de l'application est *double*, soit 8 octets. Si nous appliquons l'expression (A.1), on obtient : $8 * 18000 * 603 * 1 * 1 * 9 * 1,44 * 2 \simeq 2,25$ Go. Cette estimation est confirmée par la commande UNIX *top* lancée sur une exécution de l'application. On peut remarquer que certains paramètres de l'expression (A.1) sont unitaires ($N_z = 1$ et $T = 1$), ils n'apportent pas une augmentation de l'occupation RAM.

Les architectures à 32 bits supportent au maximum des RAM de dimension 4 Go, avec des processus qui peuvent aller jusqu'à 2 ou 3 Go[3]. D'après l'estimation (A.1) de l'application sur l'acoustique marine, l'allocation mémoire touche déjà aux limites pour ce type d'architectures.

[1] En effet, en général *NB_matrices* n'est pas un nombre constant. Dans des générations particulières, comme par exemple les générations avec inclusion du test du gradient, on a plus de matrices stockées. On se limite dans cette analyse aux deux matrices *state* et *gradient*, puisque ce cas est le plus courant dans la phase de production des résultats pour une application d'assimilation.

[2] La dimension de cet espace est un cas particulier. $N_x = 18000$ correspond à une longueur de 9 km et $N_y = 603$ correspond à une longueur de 156 m, ce qui est un cas réel d'utilisation. Ce modèle numérique permet à ces paramètres de varier d'un minimum de quelques centaines de mètres à un maximum de 40 km pour N_x et d'un minimum de quelques dizaines de mètres à un maximum de plusieurs centaines de mètres pour N_y.

[3] La valeur exacte dépend du réglage machine.

Ceci dit, des architectures à 64 bits permettent de faire tourner sans problème cette application. Un autre exemple d'application réelle, de taille à peu près similaire, est celui de la couleur de l'océan [Brajard *et al.*, 2006].

A.2.2 L'application NEMO

L'application NEMO [Madec, 2008] présente un espace de calcul de plus grande dimension par rapport à l'application précédente. La configuration qui correspond à celle qui a été mise en œuvre avec YAO est connue sous le nom de *GYRE* et a un espace à 3 dimensions formé par $N_x = 32$, $N_y = 22$, $N_z = 31$. Les trois tailles sont propres à la configuration GYRE et dépendent de sa résolution. Des résolutions spatiales plus fines amèneraient à des tailles plus importantes. La dimension de la trajectoire varie en fonction de l'expérience d'assimilation de données à faire. La trajectoire est souvent comprise entre 400 et 9600 pas de temps[4]. Le nombre de *fonctions de base* est de 86 avec un nombre moyen de sorties de 2. Si nous appliquons l'expression (A.1) sur une fenêtre temporelle d'un mois et d'un an, on obtient : $8*32*22*31*400*86*2*2 \simeq 6$ Go et $8*32*22*31*4800*86*2*2 \simeq 288$ Go. Il est évident que, pour des applications de si grande dimension, l'allocation mémoire explose[5]. Le paramètre principal, comme c'est très souvent le cas dans d'autres applications, est la fenêtre temporelle T. Il semble ainsi intéressant d'opérer une optimisation sur cette coordonnée, en vue d'améliorer l'exploitation de la mémoire RAM.

A.3 Recalculer *versus stocker*

Il y a deux approches possibles dans la littérature, dont l'objectif est de résoudre ce coûteux calcul de l'adjoint : le *Store-All* (SA) et le *Recompute-All* (RA) [Ferron et Hascoët, 1996].

A.3.0.1 La technique *Store-All* (SA)

Cette technique est orientée vers une mémorisation totale des informations. Quand les procédures *forward* sont exécutées, toutes les sorties des modules à chaque pas de temps sont stockées. Ainsi, l'information est prête pour l'utilisation dans le calcul de l'adjoint, comme illustré dans la figure A.3. Cette stratégie utilise un stockage de mémoire maximale et est le mode de fonctionnement actuel de YAO, comme illustré par le pseudocode à la section A.2. Durant la procédure *forward*, pour chaque pas de la trajectoire et pour chaque module, on stocke le vecteur *state* (les points en noir de la figure), information qui sera récupérée dans la procédure *backward* (les points en blanc de la figure).

[4] 400 est équivalent à une fenêtre temporelle de 1 mois et 9600 de 2 ans ; dans notre estimation nous prenons des cas souvent utilisés qui correspondent à un mois et un an ($T = 400$ et $T = 4800$).

[5] Actuellement le laboratoire LOCEAN dispose d'un serveur qui a 256 Go de RAM.

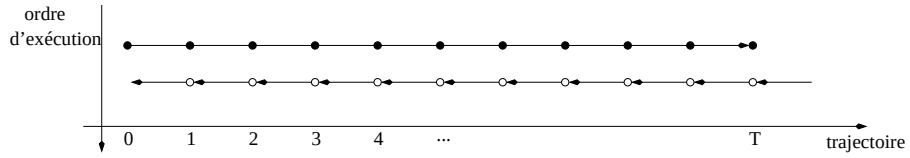

FIG. A.3 – SA : sur l'axe horizontal il y a les pas de temps de la trajectoire et sur l'axe vertical l'ordre d'exécution des prodédures *forward* et *backward*. Avec les points noirs et blancs, nous représentons le stockage et la récupération des informations.

A.3.0.2 La technique *Recompute-All* (RA)

Cette technique est orientée vers une mémorisation minimale des informations : chaque fois qu'une information est demandée, elle est recalculée. Durant l'exécution des procédures *backward*, les informations ne sont pas encore présentes, comme pour la technique SA, car les états des modules ne sont pas stockés. L'information est donc recalculée en exécutant des procédures *forward* à partir du début de la trajectoire, comme montré en figure A.4. Avec cette technique, la demande en allocation mémoire est presque nulle, mais elle impose un recalcul maximal.

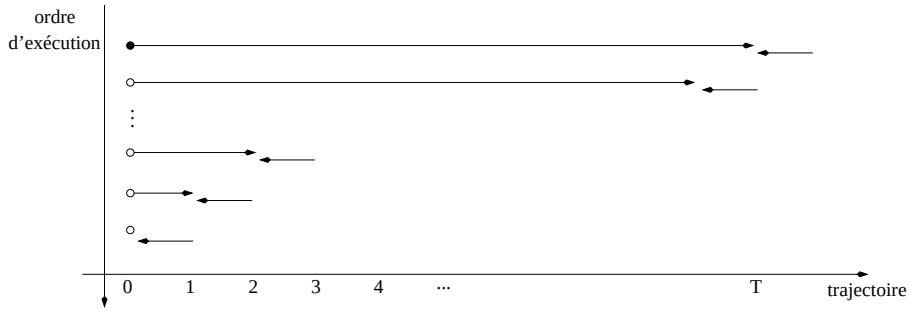

FIG. A.4 – RA : sur l'axe horizontal, il y a les pas de temps de la trajectoire et sur l'axe vertical l'ordre d'exécution des prodédures *forward* et *backward* (les *forward* de gauche à droite et les *backward* de droite à gauche). Chaque procédure *backward* au pas de temps t suit le recalcul préalable de toutes les procédures *forward* des pas de temps précédents.

A.3.0.3 *Checkpointing*

Les deux techniques SA et RA sont inefficaces pour des applications de grandes dimensions, d'une part du fait que l'allocation mémoire importante, et de l'autre du fait d'un recalcul trop coûteux. Nous voulons donc trouver un compromis entre les deux techniques. Le *checkpointing* consiste à réduire en même temps l'allocation mémoire propre à la technique SA et le recalcul propre à la technique RA. Il est basé sur le concept de *snapshot* qui est un instantané de l'état d'exécution de la procédure *forward*. La caractéristique du *snapshot* est que, si nous arrêtons l'exécution de la procédure *forward*, nous pouvons la redémarrer en utilisant seulement l'information

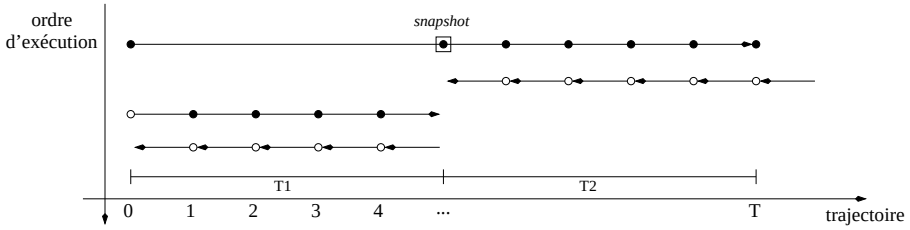

FIG. A.5 – SA avec *checkpointing*. Nous exécutons des procédures *forward* jusqu'à la fin de la trajectoire en stockant les états dans la partition de T_2, puis nous exécutons les procédures *backward* jusqu'au *snapshot*. A partir du début de T on recommence l'exécution des procédures *forward* jusqu'au *snapshot* en stockant les états de la partition T_1. C'est à ce moment que nous pouvons calculer les procédures *backward* de la partition T_1.

contenue dans le *snapshot*. Si, par exemple, nous prenons un *snapshot* au milieu de la trajectoire, la fenêtre temporelle est naturellement partitionnée en deux parties T_1 et T_2, la procédure *Store-All* devient comme en figure A.5. Les points noirs représentent l'écriture de l'état de tous les modules sur un pas de temps. En général, le *snapshot* est formé par les états des modules sur plusieurs pas de temps, puisque, pour redémarrer le calcul, la procédure *forward* pourrait avoir besoin des sorties d'un module à $t + d_t$ avec $d_t < 0$. Le *snapshot* est ainsi un ensemble d'états (représenté par un ensemble de vecteur *state*). La dimension du *snapshot* dérive des directives *ctin*, et plus précisément du délai d_t. C'est en stockant les derniers $|d_t| + 1$ états que l'on sera en condition de redémarrer l'exécution de la procédure *forward* à partir d'un *snapshot*.

Un avantage de la stratégie SA avec *checkpointing* par rapport à la technique SA simple est qu'au prix d'un *snapshot* supplémentaire, nous diminuons ainsi de moitié l'allocation mémoire (en supposant que le *snapshot* est de petite dimension, ce qui est souvent le cas). Une alternative est de faire la même démarche pour la technique *Recompute-All*, comme montré en figure A.6. L'avantage de la technique RA avec *checkpointing*, par rapport à la technique RA simple, est que nous réduisons de moitié le temps de calcul au prix d'un *snapshot*.

A.4 Une application à YAO

Le nombre de *snapshots* pour les deux techniques présentées peut être augmenté. On remarque que, si nous utilisons un nombre important de *snapshots*, les deux techniques sont équivalentes, elles convergent vers le même schéma. Une question très importante est la dimension du *snapshot* qui dépend du plus grand délai $|d_t|$ de tous les *ctin* de l'application. En général, le délai est $|d_t| \leq 2$. Une application à YAO des techniques SA et RA pourrait être l'utilisation de la technique SA avec un certain nombre de *snapshots* S. Un scénario possible est d'avoir 20 pas de temps ($T = 20$) avec 3 *snapshots* ($S = 3$) et k pas pour chaque intervalle T_i. Nous savons que $T = k(S+1)$, d'où on en tire $k = 5$ ($S+1$ car, avec S points, on peut former $S+1$ intervalles). En supposant d'avoir

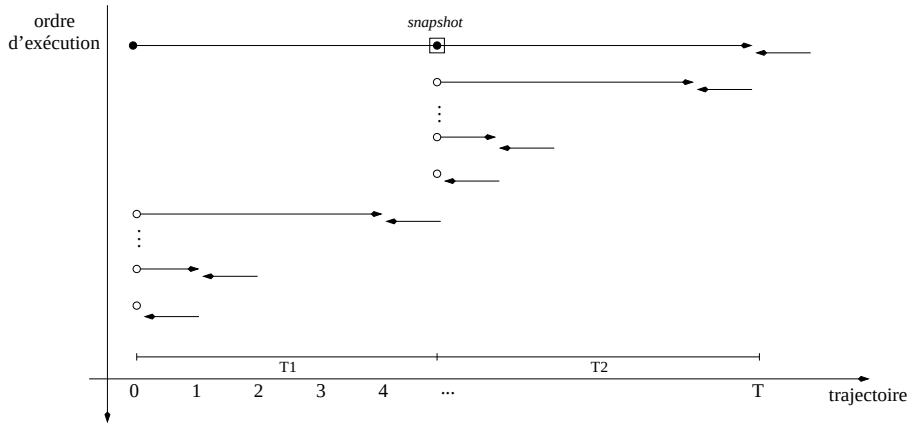

FIG. A.6 – RA avec *checkpointing*. Nous exécutons les procédures *forward* jusqu'à la fin de la trajectoire et on stocke les états dans les *snapshots*. A la fin de la trajectoire, nous calculons une fois la procédure *backward*. La deuxième procédure *backward* est calculée en recalculant les procédures *forward* depuis le début du *snapshot*. On itère jusqu'au *snapshot*, puis nous effectuons les mêmes opérations pour la partition T_1 en commençant à recalculer cette fois-ci depuis l'origine de l'axe de la trajectoire.

$d_t = -1$, le *snapshot* doit pouvoir sauvegarder deux *state* pour chaque module[6]. Ces deux *states* correspondent aux informations sur deux pas de temps, comme illustré à la figure A.7.

Performance de la technique SA avec *checkpointing*. Cette technique a un temps de calcul qui dépend d'une passe avant complète pour initialiser les *snapshots*, une passe arrière complète pour le calcul des procédures *backward*, $S+1$ passes en avant de dimension $k+d_t-1$ chacune ($d_t \leq 0$) et S récupérations : $2*T+(k+d_t-1)*(S+1)$. Le calcul dans le cas de la technique SA simple est $2*T$, nous avons donc perdu $(k+d_t-1)*(S+1)$ pas de recalcul et S récupérations. L'allocation mémoire est donnée par le nombre de *snapshots* S, étant donné que nous devons stocker $S*(|d_t|+1)$ informations. En outre, nous devons stocker une matrice de k états. L'expression de l'allocation mémoire totale est $\bar{T} = k+S*(|d_t|+1)$. La même expression pour la technique SA simple est T, qui est égal à $k*(S+1)$.

Dans la section A.2, l'exemple NEMO montre que la mise en œuvre de l'allocation mémoire dans YAO (la technique SA simple actuellement), donne une allocation de 288 Go pour $T = 4800$. Utilisant la technique SA avec *checkpointing* et $T = 400*12$ avec $k = 400$, $S+1 = 12$ et $d_t = -1$, on obtient $\bar{T} = k+S*(|d_t|+1) = 400+11*2 = 422$. En recalculant l'allocation mémoire totale (A.1) changeant le paramètre $T = 4800$ avec $\bar{T} = 422$, on obtient $2*8*32*22*31*422*86*2*2 = 50,7$ Go. En effet, le délai d_t n'impacte pas beaucoup l'occupation totale de la mémoire.

[6]En effet, on n'est pas obligé de stocker deux *states* pour chaque module, puisque les *states* à $t-2$ dont on a besoin concernent seulement certains modules.

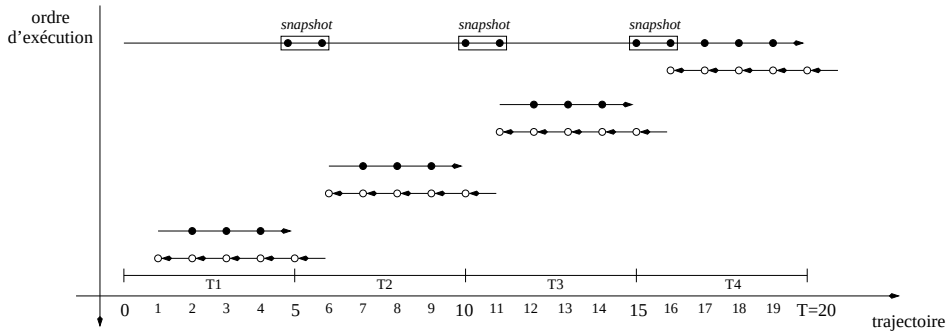

FIG. A.7 – SA avec *checkpointing*. La trajectoire est composée de $T = 20$ pas de temps et $S = 3$ *snapshots*.

La même simulation avec $d_t = -2$ donne $\bar{T} = 400 + 11 * 3 = 433$, ce qui implique une occupation mémoire totale de 52 Go.

A.4.0.4 Parallélisation du recalcul

Le *checkpointing* est, en général, une bonne solution pour résoudre les problèmes liés à une importante allocation mémoire. Pourtant, le temps de calcul est affecté négativement par cette optimisation mémoire. Nous pouvons dans ce cas exploiter une technique de recalcul par tâches parallèles afin de réduire ce retard additionnel. L'idée est de calculer la passe arrière dans la partition T_i de la trajectoire et de recalculer parallèlement la passe avant sur la partition T_{i-1}, comme illustré en figure A.8. Cette technique permet d'éviter une partie des effets secondaires dus au *checkpointing* car quand, en passe arrière, nous arrivons au *snapshot* qui est à la fin de la partition T_i, la passe avant parallèle a déjà recalculé les informations dont on a besoin pour continuer le calcul de la passe arrière dans la partition T_{i-1}. Le coût de cette parallélisation est une augmentation de k sur la mémoire allouée, les structures de données allouées pour une partition doivent être doublées, de manière à contenir en même temps les états que la passe arrière est en train d'utiliser, et les états que la passe avant est en train de recalculer. Ainsi, les passes en arrière ne lisent pas les données à partir de la même structure de données. Un commutateur gère la logique de coordination de la passe avant et de la passe arrière pour écrire et lire dans la bonne structure. L'allocation

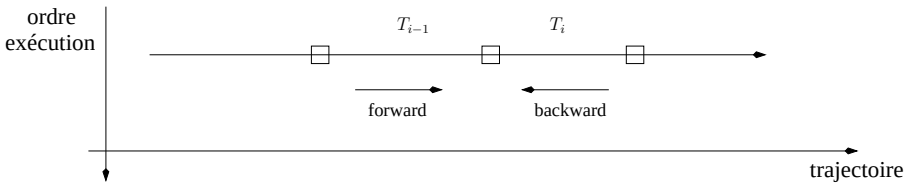

FIG. A.8 – *Checkpointing* parallèle : on calcule parallèlement la passe arrière à T et le recalcul de la passe avant à T_{i-1}.

mémoire totale de la technique SA avec *checkpointing* parallèle est : $\bar{T} = 2 * k + S * (d_t + 1)$. Par exemple, avec l'instance $T = 4800$, $k = 400$, $S = 11$ et $|d_t| = 1$, on a $\bar{T} = 800 + 22 = 822$ (donc une allocation d'environ $98,7$ Go pour l'application NEMO). Nous remarquons qu'une augmentation du nombre de processeurs ne permet pas une extension de cette technique de parallélisme par tâches, car les procédures *backward* sont calculées de manière séquentielle.

A.4.1 Possible extension à l'espace

La technique du *checkpointing* sur l'axe du temps ne donne pas une solution définitive au problème du besoin en mémoire des applications. Une possible extension du *checkpointing* sur l'axe du temps est d'envisager un *checkpointing* sur les axes de l'espace. Dans ce cas, le concept de *snapshot* change, les informations le constituant concernent un intervalle d'un point de l'espace. L'intervalle est déterminé par rapport au vecteur distance **d**. Si l'on considère un vecteur distance à deux dimensions avec composantes $(d_i, d_j) = (-1, -1)$, le *snapshot* spatial peut être représenté comme en figure A.9. A partir de ce *snapshot*, on sera en condition de redémarrer l'exécution de la procédure *forward*. Ceci pourrait ouvrir la possibilité de réduire davantage l'allocation de la mémoire.

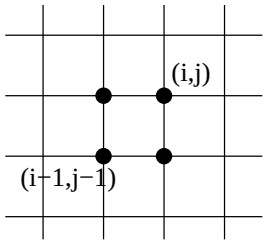

FIG. A.9 – Un *snapshot* spatial pour un vecteur distance $(d_i, d_j) = (-1, -1)$.

Annexe B

Démonstration de la formule de la procédure *backward*

Soit un modèle M représenté par un graphe modulaire G, qui calcule pour un vecteur d'entrée $\mathbf{x}$ un vecteur de sortie $\mathbf{y}$. On suppose définie en fonction de $\mathbf{y}$ une fonction $J(\mathbf{y})$ dérivable, il est alors possible de calculer les dérivées partielles $\frac{\partial J}{\partial y_i}$ par rapport aux composantes du vecteur $\mathbf{y}$. Considérons un module F_s (F_s est une *fonction de base* calculée en un point, elle représente un module dans cette annexe) du graphe modulaire G, qui admet en entrée un vecteur $\mathbf{x}_s$ de dimension n et calcule en sortie un vecteur $\mathbf{y}_s$ de dimension m. Ce module correspond à une fonction vectorielle :

$$\mathbf{y}_s = F_s(\mathbf{x}_s) = (F_{s1}(\mathbf{x}_s), F_{s2}(\mathbf{x}_s), ..., F_{sm}(\mathbf{x}_s)).$$

Il s'agit de calculer $\frac{\partial J}{\partial x_{si}}$, où x_{si} représente la $i^{ème}$ composante du vecteur $\mathbf{x}_s$. Nous remarquons d'abord que le vecteur $\mathbf{x}_s$ permet le calcul du vecteur $\mathbf{y}_s$. Ce dernier permet lui aussi le calcul des vecteurs de sortie $\mathbf{y}_d$ des modules F_d successeurs de F_s dans le graphe G. Ainsi, étant donné y_{sj}, qui représente la $j^{ième}$ composante de $\mathbf{y}_s$, on lui associe $SUCCM(s, j)$, qui est formé par l'ensemble des couples (d,k), où F_d admet comme $k^{ième}$ variable d'entrée la variable y_{sj}. Tenant compte de ces notations et des règles de dérivation de la composition de fonctions, on tire alors :

$$\frac{\partial J}{\partial x_{si}} = \sum_{j=1}^{m} \frac{\partial J}{\partial y_{sj}} \frac{\partial y_{sj}}{\partial x_{si}} = \sum_{j=1}^{m} \frac{\partial J}{\partial y_{sj}} \frac{\partial F_{sj}}{\partial x_{si}}. \tag{B.1}$$

D'autre part :

$$\frac{\partial J}{\partial y_{sj}} = \sum_{(d,k)\in SUCCM(s,j)} \frac{\partial J}{\partial x_{dk}} \frac{\partial x_{dk}}{\partial y_{sj}}. \tag{B.2}$$

Or, par définition $\frac{\partial x_{dk}}{\partial y_{sj}} = 1$ car $x_{dk} = y_{sj}$.

De (B.1) et (B.2), on obtient alors la relation suivante :

$$\frac{\partial J}{\partial x_{si}} = \sum_{j=1}^{m} \left(\sum_{(d,k)\in SUCCM(s,j)} \frac{\partial J}{\partial x_{dk}} \right) \frac{\partial F_{sj}}{\partial x_{si}}. \tag{B.3}$$

La formule (B.3) montre que la connaissance des dérivées partielles de J par rapport aux entrées des modules F_d, successeurs de F_s, permet le calcul des dérivées partielles $\frac{\partial J}{\partial x_{si}}$ (par rapport aux entrées du module F_s). Du point de vue du graphe modulaire, $SUCCM(s,j)$ représente l'ensemble des extrémités terminales des *connexions de base* prenant leur entrée de la $j^{ième}$ sortie de F_s. Si nous notons :

$$\beta_{sj} = \sum_{(d,k)\in SUCCM(s,j)} \frac{\partial J}{\partial x_{dk}} = \sum_{(d,k)\in SUCCM(s,j)} \alpha_{dk},$$

ce qui correspond à l'étape 2.b de l'algorithme 3 à la section 3.2.2, la formule (B.3) devient :

$$\alpha_{si} = \frac{\partial J}{\partial x_{si}} = \sum_{j=1}^{m} \beta_{sj} \frac{\partial F_{sj}}{\partial x_{si}}. \tag{B.4}$$

Ainsi, la connaissance des dérivées partielles par rapport aux variables de sorties : $\frac{\partial J}{\partial \mathbf{y}_i}$ permet d'initialiser l'algorithme *backward*. La formule (B.4) permet d'effectuer le calcul par rétropropagation.

Remarque : Certaines sorties de F_s sont parfois transmises à des *output data points* qui interviennent directement dans le calcul de J. Ceci explique l'étape 2.c de l'algorithme 3.

Annexe C

Stockage de matrices et performances du programme séquentiel

Améliorer les performances d'un programme ne signifie pas seulement le paralléliser, mais aussi trouver les goulots d'étranglement du programme séquentiel qui empêchent une exécution optimale. Les mauvaises performances d'un programme séquentiel sont souvent dues à une utilisation sous-optimale de la mémoire *cache*. Les *cache miss* et *cache hit* [Silberschatz *et al.*, 2008] sont des paramètres importants, puisqu'ils impliquent d'aller chercher les données dans la mémoire principale, avant qu'une unité de traitement ne puisse les utiliser. Ceci est beaucoup plus coûteux en temps de calcul que d'aller chercher les données directement dans la mémoire *cache*. Dans un système à mémoire partagée, un *cache miss* implique un échange de données et une surcharge supplémentaire sur le système d'interconnexion. Un des principaux objectifs, dans l'optimisation de code séquentiels, est de réduire le taux de *cache miss* en organisant les accès aux données de sorte que les valeurs soient utilisées aussi souvent que possible pendant qu'elles sont dans la mémoire *cache*. La stratégie la plus commune est basée sur le principe de localité spatiale et temporelle pour exploiter des comportements prévisibles dans le cadre de l'accès aux données. En C, un tableau à deux dimensions [i][j] est stocké par lignes (*rowise storage*). Par exemple, l'élément [1][1] est suivi par [1][2], qui est suivi par [1][3], et ainsi de suite. Ainsi, quand un élément du tableau est transféré dans la mémoire *cache*, les voisins de cet élément sont aussi transférés, étant donné qu'il font partie du même bloc d'informations. Le programme qui accède au tableau doit respecter ce type de stockage sous peine, autrement, de subir un ralentissement important.

Dans une application YAO, la traversée de l'espace est choisie par l'utilisateur dans les directives *order*. Dans de nombreuses applications YAO, il existe plusieurs traversées possibles. L'utilisateur devrait programmer les directives *order* de façon à trier les axes dans l'ordre i, j, k afin d'exploiter le principe de localité. Des simulations ont montré qu'on pourrait avoir un ralentissement d'un facteur 2 si l'on définit la traversée de la mauvaise façon. En outre, dans le calcul scientifique, la plupart des scientifiques utilisent le langage Fortran. Dans ce langage, le stockage des tableaux est inversé (*columnwise storage*) ; c'est pourquoi, dans les applications YAO, ils ont tendance à programmer ces accès à l'inverse, rendant le programme deux fois plus lent. La géné-

ration automatique des directives *order* prend en compte ces aspects basiques d'optimisation de code séquentiels. L'axe priviligié dans le choix de la boucle extérieure de l'algorithme 6 est toujours *i*. Ceci n'est pas spécifié dans l'algorithme mais, dans la pratique, c'est l'axe qui est choisi de préférence à la ligne 16.

Des travaux d'optimisation du code séquentiel de YAO sont actuellement menés au LOCEAN. On ne donnera pas beaucoup de détails sur ces optimisations, car elles présentent uniquement un intérêt au niveau opérationnel. Elles sont centrées sur deux points :

- redéfinir et reformuler les structures de données de YAO, pour prendre en compte les effets de *cache*. Ceci consiste à changer les structures de données du code généré de la version actuelle de YAO qui ne sont pas optimisées[1].
- utiliser des bibliothèques performantes (comme BLAS par exemple), pour effectuer le produit de la procédure *backward*.

Des essais sur les deux points ont déjà démontré l'accélération importante que l'on peut obtenir avec ces optimisations.

[1] Les structures de données de YAO sont présentées dans l'annexe A.

Bibliographie

[Aho *et al.*, 2006] AHO, A. V., LAM, M. S., SETHI, R. et ULLMAN, J. D. (2006). *Compilers : Principles, Techniques, and Tools*. Addison Wesley, 2de édition.

[Allen *et al.*, 1987] ALLEN, R., CALLAHAN, D. et KENNEDY, K. (1987). Automatic decomposition of scientific programs for parallel execution. *In POPL '87 : Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 63–76, New York, NY, USA. ACM.

[Allen et Kennedy, 1982] ALLEN, R. et KENNEDY, K. (1982). PFC : A program to convert Fortran to parallel form. *In Proceedings of IBM Conference on Parallel Computing and Scientific Computations*.

[Badran *et al.*, 2008] BADRAN, F., BERRADA, M., BRAJARD, J., CRÉPON, M., SORROR, C., THIRIA, S., HERMAND, J.-P., MEYER, M., PERICHON, L. et ASCH, M. (2008). Inversion of Satellite Ocean Colour Imagery and Geoacoustic Characterization of Seabed Properties : Variational Data Inversion Using a Semi-automatic Adjoint Approach. *J. of Marine Systems*, 69:126–136.

[Bastoul, 2004] BASTOUL, C. (2004). Code Generation in the Polyhedral Model Is Easier Than You Think. *In PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France.

[Bernstein, 1966] BERNSTEIN, A. J. (1966). Analysis of Programs for Parallel Processing. *Electronic Computers, IEEE Transactions on*, EC-15(5):757 –763.

[Berrada, 2008] BERRADA, M. (2008). *Une approche variationnelle de l'inversion, de la recherche locale à la recherche globale par carte topologique : application en inversion géoacoustique*. PhD Thesis, Université Pierre et Marie Curie, UPMC, France.

[Berrada *et al.*, 2009] BERRADA, M., BADRAN, F., CRÉPON, M., HERMAND, J. et THIRIA, S. (2009). A robust probabilistic approach for variational inversion in ocean acoustic tomography. *Inverse Problems*, volume 25.

[Bondhugula *et al.*, 2008] BONDHUGULA, U., HARTONO, A., RAMANUJAM, J. et SADAYAPPAN, P. (2008). A practical automatic polyhedral parallelizer and locality optimizer. *In PLDI '08 : Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 101–113, New York, NY, USA. ACM.

[Bouttier et Courtier, 1997] BOUTTIER, F. et COURTIER, P. (1997). Data Assimilation, Concepts and Methods. *In Training Course Notes of ECMWF, European Center for Medium-range Weather Forecasts*, volume 53, UK.

[Brajard *et al.*, 2006] BRAJARD, J., JAMET, C., MOULIN, C. et THIRIA, S. (2006). Use of a Neuro-variational Inversion for Retrieving Oceanic and Atmospheric Constituents from Satellite Ocean Colour Sensor : Application to Absorbing Aerosols. *Neural Networks*, 19(2):178–185.

[Chapman *et al.*, 2007] CHAPMAN, B., JOST, G. et PAS, R. v. d. (2007). *Using OpenMP : Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.

[Courtier *et al.*, 1994] COURTIER, P., THÉPAUT, J. et HOLLINGSWORTH, A. (1994). A Strategy for Operational Implementation of 4D-VAR Using an Incremental Approche. *Q. J. R. Meteorological Society*, 120:1367–1387.

[Darte, 1999] DARTE, A. (1999). On the complexity of loop fusion. *Parallel Computing*, 26:149–157.

[Darte *et al.*, 2000] DARTE, A., ROBERT, Y. et VIVIEN, F. (2000). *Scheduling and automatic parallelization*. Morgan Kaufmann Publishers Inc., USA.

[Ferron et Hascoët, 1996] FERRON, B. et HASCOËT, L. (1996). Capacités actuelles de la Différentiation Automatique : l'adjoint d'OPA par TAPENADE. *In Colloque National sur l'Assimilation de Données*, Toulouse, France.

[Fouilloux et Piacentini, 1999] FOUILLOUX, A. et PIACENTINI, A. (1999). *The PALM Project : MPMD Paradigm for an Oceanic Data Assimilation Software*, volume 1685 de *LNCS*. Springer, Berlin.

[Giering et Kaminski, 1998] GIERING, R. et KAMINSKI, T. (1998). Recipes for Adjoint Code Construction. *ACM Transactions on Mathematical Software*, 24(4):437–474. `http://www.FastOpt.com/papers/racc.pdf`.

[Gilbert et Lemaréchal, 1989] GILBERT, J. et LEMARÉCHAL, C. (1989). Some Numerical Experiments with Variable-storage Quasi-newton Algorithms. *Mathematical Programming*, 45:407–435. `http://www-roc.inria.fr/~gilbert/modulopt/index.html`.

[Gprof, 1993] GPROF (1993). `http ://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`.

[Griewank et Walther, 2008] GRIEWANK, A. et WALTHER, A. (2008). *Evaluating Derivatives : Principles and Techniques of Algorithmic Differentiation*. 2de édition.

[Hascoët et Pascual, 2004] HASCOËT, L. et PASCUAL, V. (2004). TAPENADE 2.1 User's Guide. Technical report, INRIA, France. `http://www-sop.inria.fr/tropics/papers/Hascoet2004T2u.html`.

[Hermand *et al.*, 2006] HERMAND, J.-P., MEYER, M., ASCH, M. et BERRADA, M. (2006). Adjoint-based Acoustic Inversion for the Physical Characterization of a Shallow Water Environment. *J. Acoust. Soc. Am.*, 119(6):3860–3871.

[Ide *et al.*, 1997] IDE, K., COURTIER, P., GHIL, M. et LORENC, A. (1997). Unified Notation for Data Assimilation : Operational, Sequential and Variational. *Special Issue J. Meteorological Society Japan*, 75:181–189. Data Assimilation in Meteorology and Oceanography : Theory and Practice.

[Jin *et al.*, 2000] JIN, H., FRUMKIN, M. A. et YAN, J. (2000). Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes. *In ISHPC '00 : Proceedings of the Third International Symposium on High Performance Computing*, pages 440–456, London, UK. Springer-Verlag.

[Kane *et al.*, 2006] KANE, A., THIRIA, S. et MOULIN, C. (2006). Développement d'une Méthode d'Assimilation de Données in Situ dans une Version 1D du Modèle de Biogéochimie Marine PISCES (in French). Mémoire de D.E.A., LSCE/IPSL, CEA-CNRS-UVSQ laboratories, France.

[Kennedy et Mckinley, 1994] KENNEDY, K. et MCKINLEY, K. S. (1994). Typed fusion with applications to parallel and sequential code generation. Rapport technique CRPC-TR94646, Center for Research on Parallel Computation, Rice University, Houston USA.

[Le Dimet et Talagrand, 1986] LE DIMET, F.-X. et TALAGRAND, O. (1986). Variational Algorithms for Analysis and Assimilation of Meteorological Observations : Theoretical Aspects. *J. Tellus, Series A, Dynamic Meteorology and Oceanography*, 38(2):97–110.

[Legrand et Robert, 2003] LEGRAND, A. et ROBERT, Y. (2003). *Algorithmique parallèle*. Dunod, France.

[Louvel, 1999] LOUVEL, S. (1999). *Etude d'un Algorithme d'Assimilation Variationnelle de Données à Contrainte Faible. Mise en Oeuvre sur le Modèle Océanique aux Equations Primitives MICOM*. PhD, Université de Toulouse III, France.

[Louvel, 2001] LOUVEL, S. (2001). Implementation of a Dual Variational Algorithm for Assimilation of Synthetic Altimeter Data in the Oceanic Primitive Equation Model micom. *J. Geophys. Res.*, 106(C5):9199–9212.

[Madec, 2008] MADEC, G. (2008). *NEMO ocean engine*. Note du Pôle de modélisation de l'Institut Pierre-Simon Laplace No 27, LOCEAN, Paris, France.

[Nardi *et al.*, 2009] NARDI, L., SORROR, C., BADRAN, F. et THIRIA, S. (2009). YAO : A Software for Variational Data Assimilation Using Numerical Models. *In* GERVASI, O., TANIAR, D., MURGANTE, B., LAGANÀ, A., MUN, Y. et GAVRILOVA, M. L., éditeurs : *LNCS 5593, Computational Science and Its Applications - ICCSA 2009*, pages 621–636. Springer-Verlag.

[Naumann *et al.*, 2006] NAUMANN, U., UTKE, J., WUNSCH, C., HILL, C., HEIMBACH, P., FAGAN, M., TALLENT, N. et STROUT, M. (2006). Adjoint Code by Source Transformation with OpenAD/F. *In* WESSELING, P., PÉRIAUX, J. et OÑATE, E., éditeurs : *Proceedings of the European Conference on Computational Fluid Dynamics (ECCOMAS CFD 2006)*. TU Delft. `http://proceedings.fyper.com/eccomascfd2006/documents/35.pdf`.

[Nichols *et al.*, 1996] NICHOLS, B., BUTTLAR, D. et FARRELL, J. P. (1996). *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[Oprofile, 2000] OPROFILE (2000). `http ://oprofile.sourceforge.net/about/`.

[Pacheco, 1996] PACHECO, P. S. (1996). *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[PAPI, 1999] PAPI (1999). Performance Application Programming Interface. http ://icl.cs.utk.edu/papi/.

[Redler *et al.*, 2010] REDLER, R., VALCKE, S. et RITZDORF, H. (2010). Oasis4 - a coupling software for next generation earth system modelling. *Geoscientific Model Development*, 3(1): 87–104.

[Silberschatz *et al.*, 2008] SILBERSCHATZ, A., GAGNE, G. et GALVIN, P. B. (2008). *Operating System Concepts*. Wiley & Sons Australia, Limited, John.

[Sportisse et Quélo, 2004] SPORTISSE, B. et QUÉLO, D. (2004). Assimilation de Données. 1ère Partie : Eléments Théoriques. Rapport technique, CEREA.

[Sun, 2005] SUN (2005). Sun Studio 11 Performance Analyzer. `http ://docs.sun.com/app/docs/doc/819-3687`.

[Taillard *et al.*, 2008] TAILLARD, J., GUYOMARC'H, F. et DEKEYSER, J.-L. (2008). A Graphical Framework for High Performance Computing Using An MDE Approach. *In PDP '08 : Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 165–173, Washington, DC, USA. IEEE Computer Society.

[Talagrand, 1991] TALAGRAND, O. (1991). The Use of Adjoint Equations in Numerical Modelling of the Atmospheric Circulation. *In* GRIEWANK, A. et CORLISS, G., éditeurs : *Workshop of Automatic, Differentiation of Algorithms : Theory, Implementation and Applications*, Breckenridge, Colorado, USA.

[Talagrand, 1997] TALAGRAND, O. (1997). Assimilation of Observations, an Introduction. *J. Meteorological Society Japan*, 75:191–209.

[VTune, 2009] VTUNE (2009). Intel VTune Performance Analyzer. `http ://software.intel.com/en-us/intel-vtune/`.

[Weaver *et al.*, 2005] WEAVER, A., DELTEL, C., MACHU, E., RICCI, S. et DAGET, N. (2005). A multivariate balance operator for variational ocean data assimilation. *Q. J. Royal Meteorological Society*, volume 131(613):3605–3625.

[Wolfe, 1989] WOLFE, M. (1989). Iteration Space Tiling for Memory Hierarchies. *In Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[YAO, 2010] YAO (2010). Home Page. `http://www.locean-ipsl.upmc.fr/~yao/`.

Index