Environnement Interopérable Distribué pour les Simulations Numériques avec Composants CAPE-OPEN

Laurent Pigeon

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Spécialité : « Informatique : Systèmes et Logiciels »

PRÉPARÉE AU LABORATOIRE D'INFORMATIQUE DE GRENOBLE
DANS LE CADRE DE ***l'École Doctorale « Mathématiques, Sciences et Technologies de l'Information, Informatique »***

PRÉSENTÉE ET SOUTENUE PUBLIQUEMENT

PAR

LAURENT PIGEON

LE 21 SEPTEMBRE 2007

ENVIRONNEMENT INTEROPÉRABLE DISTRIBUÉ POUR LES SIMULATIONS NUMÉRIQUES AVEC COMPOSANTS CAPE-OPEN

Directeurs de thèse :
Mme Brigitte Plateau
M. Bertrand Braunschweig

JURY

M. VAN-DAT CUNG	INPG, Grenoble,	Président
M. JEAN-LOUIS PAZAT	IRISA, Rennes,	Rapporteur
M. STÉPHANE LANTÉRI	INRIA, Sophia Antipolis	Rapporteur
MME BRIGITE PLATEAU	INPG, Grenoble,	Directeur de thèse
M. BERTRAND BRAUNSCHWEIG	IFP/ANR, Rueil Malmaison,	Responsable industriel
M. THIERRY GAUTIER	INRIA, Grenoble,	Co-encadrant
M. PASCAL ROUX	IFP-Lyon, Solaize,	Co-encadrant
M. LAURENT TESTARD	HALIAS, Grenoble,	Examinateur

Engagez vous, qu'y disaient . . . Vous verrez du pays, qu'y disaient . . .
Centurion

Remerciements

Enfin ! Ceci représente la dernière page écrite de ce manuscrit qui symbolise alors presque quatre années de travail. Tout au long de la thèse, on s'imagine qu'elle sera simple à mettre en forme, mais c'est peut être la tâche la plus ardue car elle met définitivement un terme à quatre années partagées avec des gens plus formidables les uns que les autres.

Je tiens à remercier l'Institut Français du Pétrole pour avoir soutenu les travaux de recherche de cette thèse. Je voudrais remercier les membres du jury pour avoir accepté d'évaluer mes travaux. Tout naturellement, je remercie Jean-Louis Pazat et Stéphane Lantéri pour leur rapport détaillé ayant contribué à l'amélioration de ce manuscrit, Van-Dat Cung pour m'avoir fait l'honneur de présider le jury de ma soutenance et enfin Laurent Testard pour sa vision métier et son expérience quant aux problématiques soulevées par ces travaux. Ayant été de nombreuses fois en contact avec Laurent Testard en début de thèse, je tiens particulièrement à le remercier pour son enthousiasme, sa grande sympathie et ses conseils toujours des plus avisés. Un grand merci également à mes deux directeurs de thèse : Mme Brigitte Plateau et M. Bertrand Braunschweig. Egalement, je tiens à remercier chaleureusement mes deux co-directeurs, Thierry Gautier et Pascal Roux.

Thierry et moi avons commencé à travailler ensemble cela fait maintenant 5 ans qui me paraissent si peu. Dans mon manuscrit de DEA, déjà je remerciais le chercheur pour son savoir quasi paranormal du C++, mais également pour ses qualités humaines. Après cette thèse, je persiste et je signe et te remercie pour ces années passées sous ton aile. Pour toujours, je me rappellerai de Cannes'05 !

Pascal fut mon mentor à l'IFP Lyon. Malgré son emploi du temps chargé, il a toujours su m'accorder les instants au combien nécessaire pour m'expliquer les concepts liées à la simulation des procédés et au standard CAPE-OPEN si abstrait à mes yeux. Avant d'être mon encadrant, tu fus un compagnon de jeu mémorable (la science n'est-elle pas un amusement avant tout : surtout avec toi). Tu as su également tirer le meilleur de moi et il faut admettre que ce n'est pas chose si aisée.

La thèse n'est pas qu'un travail académique, c'est également une formidable expérience humaine. Elle renforce les amitiés existantes et en apporte de toutes nouvelles au combien incongru si l'on sort de ce contexte. Je tiens à m'excuser auprès de toutes et tous que je pourrais oublier mais n'ayez crainte, je prendrai soin de réparer cela autour du petite bière (pourquoi une et pourquoi petite ?). A toi Maxime (aka Maximo, Maxence, Maxou, Maximilio ou bien encore "Saloperie de mec du sud") pour tant de débats philosophiques (ou brèves de comptoir) sur tant de sujets : tu fus un formidable compagnon de détente et également une d'oreille si attentive pour mes questions existentielles sur les architectures HPC. A toi Xavier. C'est toujours plaisant de travailler avec quelqu'un surtout quant cette personne : c'est toi. Aller, plus qu'un an et demi à tirer et toi aussi, tu pourras écrire tes remerciements. A toi également mon petit ami de la forêt enchantée où vivent farfadets et fées. Gouby, tu es un ami fidèle et sache que ces deux mois de collocations furent les plus mémorables de ma carrière. A tous les membres du laboratoire LIG - antenne de Montbonnot (c'était plus simple ID quand même) passé, présent et à venir qui font que chaque jour, on peut se coucher moins stupide qu'en se levant (ou des fois plus ;-)). A vous Sebo et D'joule. Comment je pourrais oublier l'année passée en votre compagnie dans ce bureau de 3/MAL21 ! Nous formions

une si belle équipe : tout R11 était représenté et, je dois l'avouer, qui mieux que nous pouvions incarner cette direction ;). Merci également à vous, amis d'un jour, amis de toujours : Floflo et Jeannot Lapin, Charlotte, Christoballe, Toine, Sly, Bigbidibidibi...ibi Emilie, Seb, Dams', Seb, Pierre-Alex, Pierrot, Duc, Gé, Monsieur propre, Jean-Mich', Anna, Anne-So, Matt et Gawanée.

Un grand merci à ma famille pour leur soutien et encouragements avant, pendant et après cette thèse. Enfin, je remercie vivement Lucile qui a su me soutenir, me faire rire, me secouer, . . . , enfin, m'aimer pendant cette longue période. Je n'y serais pas arriver sans toi !

Table des matières

Table des figures

Liste des tableaux

Introduction 1

Le génie chimique a été l'une des premières sciences à bénéficier des progrès de l'informatique pour le contrôle de ses procédés, la gestion de ses bases de données, la simulation des procédés, la modélisation moléculaire et même la conception d'installations en 3D.

La simulation des procédés est un outil qui permet à l'ingénieur de résoudre une grande variété de problèmes qui se présente à chaque étape du développement, de la conception, du fonctionnement ou de l'amélioration du procédé. Le génie des procédés s'intègre dans de nombreux secteurs d'activités tels que les industries chimique et para-chimique, pharmaceutique, pétrolière et pétrochimique, agro-alimentaire, *etc*. Par exemple, dans le secteur pétrolier, le procédé de raffinage du pétrole brut en produits finis (butane, supercarburant 95-98, gazole, fuel lourd, ...) suit un schéma de transformation clairement défini et complexe où de nombreuses unités prennent par à l'évolution du brut. Un procédé peut être considéré comme un agencement d'opérations unitaires (mélangeurs, colonnes de distillation et d'extraction, réacteurs, pompes ...) mettant en œuvre des transformations physico-chimiques. Chaque opération unitaire est décrite par un modèle mathématique, c'est à dire un ensemble d'équations permettant d'effectuer des bilans matières et d'énergie. La simulation assistée par ordinateur présente de nombreux avantages pour l'utilisateur ; par exemple, il est possible d'introduire dans l'unité modélisée les modifications des paramètres soit d'entrée soit de fonctionnement et de chercher dans quelle mesure les propriétés du produit recherché s'améliorent ou l'énergie nécessaire à sa production est réduite.

La complexité induite par la simulation de procédés est de deux ordres. La première est une *complexité logicielle*. L'utilisation toujours plus intensive des modèles numériques de plus en plus réalistes et spécialisés dans toutes les phases de développement de procédés est un constat avéré au regard des nombreuses publications du domaine visant à confronter un modèle avec les résultats expérimentaux obtenus sur une unité pilote. Cette accroissement de la demande en simulation a eu comme contrepartie un accroissement équivalent du nombre de logiciels de modélisation, de plus en plus sophistiqués, et de sources nombreuses et variées. Certains de ces logiciels réalisent des fonctions précises et bien définies comme, par exemple, le calcul de propriétés physiques, la simulation d'une opération unitaire particulière, ... Ces composants logiciels sont alors nommés « *Composants de Modélisation de Procédés* » (PMC). D'autres logiciels sont des environnements généraux destinés à faciliter la création de modèles, soit à partir de principes fondamentaux, soit à partir de bibliothèques existantes, soit les deux. Ils permettent ensuite d'effectuer un ensemble de tâches autour du modèle créé, comme la simulation et l'optimisation. Ils incorporent, en général, plusieurs PMC. Cette catégorie, nommée « *Environnement de Modélisation de Procédés* » (PME),

est au sens informatique une application de couplage de codes.

Le seconde source de complexité est une *complexité calculatoire* qui repose sur la puissance de calcul exigée par les modèles numériques sous-jacents à chaque opération unitaire. La nature des simulations numériques mises en jeu dans les applications de procédés est de plus en plus complexe, notamment à cause des différents types de modélisations (hydrodynamique, thermodynamique) et de leurs niveaux de précision de plus en plus fin. Cela correspond à la complexité accrue des sites des opérateurs de procédés et à leur demande croissante de précision de la simulation. Outre ce caractère de précision des modèles qui tend à représenter le plus fidèlement la physique, la taille des problèmes à modéliser grandit continuellement.

On sait, à présent, qu'il est nécessaire d'adopter une démarche intégrée d'ingénierie de procédés, qui puisse prendre en compte toutes les interactions souhaitables. Les PMC isolés sont d'utilisation limitée et peuvent être inutilisables dans certains cas. Le couplage d'une opération unitaire avec un serveur de calcul thermodynamique est nécessaire à la réalisation de cette dernière. Cette intégration doit pouvoir se faire aisément dans un PME général. L'initiative CAPE-OPEN, débutée officiellement en 1995, répond au problème de complexité logicielle. Il recense les principales catégories de PMC et définit des standards d'interfaces pour chaque catégorie. Un composant, conforme au standard, implante les interfaces permettant à tout PME de le commander. Ainsi, avec l'adoption de ce standard, les simulateurs de procédés seront, à terme, un assemblage de composants.

Actuellement, l'accès à de grandes puissances de calcul s'appuie sur les architectures matérielles multi-processeurs ou bien multi-ordinateurs qui correspondent à l'agglomération de nombreuses entités de calcul jouant le rôle d'une seule machine. La programmation de ces architectures nécessite une nouvelle approche de programmation : la *programmation parallèle*. Généralement, la conception d'une application parallèle présuppose une modification importante des méthodes de résolution, de même que des codes historiques disponibles. Quelques travaux [13, 102, 72, 1] ont traités de cet aspect dans le domaine de la simulation des procédés. Toutefois, ils étudient principalement les méthodes numériques de résolution définissant des applications monolithiques. La définition du standard CAPE-OPEN apporte une nouvelle vision du parallélisme engendré par les simulations de procédés. En effet, la parallélisation des méthodes numériques n'est guère envisageable dans le cadre d'environnements fermés où les composants logiciels sont des « *boîtes noires* ». L'obtention de performances nécessite alors une distribution efficace des composants métier de l'application. Jusqu'à présent, peu de travaux apportent une solution à cette problématique.

Notre travail de recherche s'oriente sur *l'étude des algorithmes et des outils permettant de concevoir des simulateurs de calcul distribué pour la simulation des procédés dont l'environnement de simulation est conforme au standard CAPE-OPEN, grâce à des algorithmes d'ordonnancement et de placement des composants sur une architecture de type grappe de calcul.*

1.1 Contexte scientifique et industriel

Cette thèse s'inscrit dans le cadre d'une collaboration CIFRE entre l'Institut Français du Pétrole (IFP) et l'Institut Nationale Polytechnique de Grenoble (INPG). L'INRIA Rhônes-Alpes, à

travers le Projet-Equipe de recherche MOAIS, et le laboratoire LIG (ID-IMAG) apportent également leur contribution à cette thèse.

L'Institut Français du Pétrole est un centre de recherche et de développement industriel, de formation et d'information pour l'industrie des hydrocarbures et de l'automobile. Son champ de recherche s'applique principalement à la conception de nouveaux procédés industriels. Dans ce cadre, l'IFP propose et améliore des solutions informatiques basées sur des techniques de simulation numérique visant à caractériser les propriétés des procédés développés. Il est également un participant actif à l'évolution du standard CAPE-OPEN.

L'Institut National de Recherche en Informatique et en Automatique (INRIA) est un organisme public de recherche français. Son ambition est de mettre en réseau les compétences de la recherche française dans le domaine des sciences et technologies de l'information. Suite au projet APACHE, le Projet-Equipe de recherche MOAIS offre une expertise complète sur les techniques de calcul parallèle.

A travers ce sujet de thèse, l'IFP et le projet MOAIS joignent leurs efforts afin d'expérimenter des solutions adaptées à l'exécution distribuée des simulations de procédés au standard CAPE-OPEN sur des grappes de calcul. Les principaux thèmes abordés par ce projet concerne la conception d'un environnement logiciel de simulations distribuées transparent à l'utilisateur. Cette solution s'inscrit dans la continuité des actions entamées depuis plusieurs années par l'IFP sur les nouveaux environnements de simulations de procédés et l'interopérabilité en se concentrant sur de nouveaux problèmes de plus grandes tailles.

Les recherches sur l'utilisation performante des grappes de calcul sont au cœur des thématiques du laboratoire LIG (ID-IMAG), en particulier sur des aspects comme la conception d'un environnement d'exécution parallèle et distribuée pour multi-processeurs, grappe ou grille à travers l'environnement d'exécution parallèle KAAPI.

1.2 Problématique et méthodologie

Le calcul parallèle haute performance représente un axe de recherche important dans toutes applications de simulations numériques qui demandent de grosses puissances de calcul. Le domaine de l'ingénierie des procédés assistée par ordinateur ne déroge pas à cette règle. L'obtention de performances suit actuellement deux axes de recherche.

Le premier axe consiste à concevoir des algorithmes parallèles performants adaptés à une architecture matérielle cible. Toutefois, ils se prêtent mal aux contraintes économiques industrielles. En effet, les entreprises possèdent des codes historiques robustes et prouvés comme tels pour une approche de résolution particulière. Une simulation de procédés regroupant de nombreux codes métier hautement spécialisés ne peut se prêter à une telle approche sans un effort important de portage des codes et de validation de leur comportement à l'exécution.

Le second axe s'oriente sur l'étude des techniques orientées *middleware*. L'objectif premier de ces approches vise à conserver les codes historiques en l'état et à développer un environnement d'exécution efficace pour les architectures matérielles haute performance actuelles. L'avènement du standard CAPE-OPEN dans la simulation des procédés conforte cette approche. Une application de simulation est alors un assemblage de composants logiciels interopérables. L'environnement de simulation représente une application de couplage de codes. Le domaine de l'informatique

parallèle s'intéresse, depuis une dizaine d'années, à proposer des environnements de couplage parallèle efficace. Leur approche cherche principalement à traiter les communications d'une manière efficace à l'exécution. La recherche *a priori* d'un placement des composants sur les entités de calcul est un facteur essentiel à une utilisation efficace des architectures matérielles. C'est dans ce cadre que s'oriente les travaux proposés dans cette thèse.

1.3 Contributions

La contribution de cette thèse repose, dans un premier temps, sur une étude de faisabilité d'une approche distribuée des simulations de procédés dont les composants sont au standard CAPE-OPEN. Pour cela, un démonstrateur interagissant avec l'environnement de simulation industriel INDISS de la société RSI [99] est implémenté. L'objectif de ce démonstrateur est de mesurer le gain (accélération) d'un tel environnement d'exécution sur des cas tests métier tout en offrant une abstraction complète de l'architecture matérielle cible à l'utilisateur.

Ce démonstrateur repose sur l'analyse du schéma de calcul afin de détecter le parallélisme potentiel de l'exécution. Ainsi, nous proposons des solutions algorithmiques qui permettent de paralléliser automatiquement les calculs effectués sur divers composants. Ces solutions interagissent avec les méthodes de calcul numérique utilisées dans les simulateurs spécifiques au domaine du CAPE. En particulier, il s'agit d'ajouter un module d'ordonnancement en charge de placer les composants logiciels sur les différents sites d'exécution. L'extension apportée à l'environnement INDISS contient environ 4000 lignes de codes. Ce prototype est la première base à nos travaux et nous a permis de quantifier grossièrement le gain atteignable sur une architecture de calcul distribué. La conception de cet environnement métier souffre de quelques limitations qui sont solvables mais difficilement implémentables. En particulier, les « *fausses* » dépendances intrinsèques aux applications itératives , liées à la ré-utilisations des zones mémoires des variables d'une itération à l'autre, sont des problèmes difficiles à résoudre. Nous nous sommes alors orientés sur la conception d'un simulateur du comportement de l'exécution d'une simulation de procédés.

L'environnement d'exécution parallèle et distribué KAAPI sert de base à notre second démonstrateur. Afin de traiter efficacement les simulations, une extension à KAAPI est implémentée offrant une politique d'exécution selon un mode statique. Cette extension représente environ 8000 lignes de codes. Nous avons employé KAAPI afin de tester différentes politiques d'exécution. L'utilisation de KAAPI et de notre extension offre la possibilité d'effectuer de nombreuses expériences optimisant notre première approche.

Cette thèse a fait l'objet de quatre communications en conférences internationales [94, 92, 91, 63], une communication dans une conférence nationale [12] et de deux communications orales [90, 93].

1.4 Organisation du manuscrit

Ce travail de recherche s'intéresse à la parallélisation des simulations physiques du domaine de l'ingénierie des procédés. Ce document est composé de cette introduction et de trois parties

comportant un total de onze chapitres.

La première partie introduit le contexte technique de cette thèse. Le premier chapitre présente le domaine de l'ingénierie des procédés et plus particulièrement sa composante informatique : le CAPE. Cette description met en avant les concepts ainsi que les approches de résolution métier régissant les problèmes de simulation de procédés. Ce bref aperçu du CAPE nous permet d'introduire le standard CAPE-OPEN qui offre un découpage métier des entités participant à une simulation sous la forme de composants logiciels. En particulier, nous nous intéressons à trois types de composants qui prennent par activement à la résolution de la simulation. Le second chapitre fournit une présentation des techniques de calcul parallèle qui permettent l'exploitation efficace des architectures hautes performances. Cinq caractéristiques doivent être traitées afin de proposer une exécution efficace. Deux d'entre elles reposent sur les techniques d'ordonnancement. Au moyen d'une représentation abstraite de l'exécution, elles permettent de proposer des politiques globale et locale d'exécution. Le chapitre 4 présente les prérequis ainsi que les heuristiques d'ordonnancement couramment employés dans le calcul parallèle. Pour conclure cette partie, le chapitre 5 propose une étude des travaux de recherche relatifs à la distribution des simulations CAPE ainsi que ceux ayant pour objectif de proposer des environnements parallèles de couplage de codes. Ce chapitre permet de positionner notre approche par rapport à celles existantes.

La deuxième partie traite, de manière approfondie, l'exécution distribuée d'une simulation CAPE dont l'environnement de simulation est conforme au standard CAPE-OPEN. Le logiciel INDISS, développé par RSI, est retenu comme environnement de base à notre approche. Le chapitre 6 étudie comment INDISS et plus spécifiquement le moteur exécutif INDISS-RT traite l'exécution d'une simulation. D'après l'étude des dépendances de données et des précédences entre les tâches de l'application, nous identifions les sources potentielles de parallélisme et les problèmes soulevés afin de les traiter. Basé sur cette étude, le chapitre 7 présente le démonstrateur développé. Nous exposons les pré-traitements nécessaires à la distribution du travail ainsi que les techniques employées à l'exécution permettant de traiter de manière efficace les communications de données entre les entités de calcul. Le chapitre 8 expose les résultats obtenus sur une grappe de calcul par la simulation d'un ensemble de configuration d'un cas test métier. Les résultats présentés montre l'intérêt d'une exécution distribuée pour les simulations de procédés. Toutefois, un ensemble restreint de tests est entrepris et ceci dû aux contraintes technologiques des codes métier utilisés. C'est pourquoi, nous nous sommes tournés vers la conception d'un démonstrateur visant à simuler le comportement de l'exécution des simulations des procédés en s'affranchissant de ces contraintes.

La troisième et dernière partie s'articule en deux chapitres. Le chapitre 9 débute par une présentation succincte de l'environnement d'exécution KAAPI tel qu'il était au début de cette thèse. Principalement, nous caractérisons son modèle d'exécution basé sur la description de l'application sous la forme d'une graphe de flot de données. La charge de calcul est répartie sur l'ensemble des ressources de manière dynamique à l'exécution *via* l'utilisation d'une heuristique d'ordonnancement par vol de travail. Nous exposons les limitations de cette stratégie d'ordonnancement pour les applications de simulation de procédés. La suite du chapitre expose les extensions apportées à KAAPI afin d'introduire les prérequis à l'intégration d'une technique d'exécution statique. Le chapitre 10 étudie, à travers des expérimentations sur grappe de calcul, les optimisations réalisées sur l'approche précédente. En particulier, elles reposent, d'une part, sur l'influence des barrières

de synchronisation en fin de calcul de chaque pas de temps et, d'autre part, sur l'étude du placement issu de l'extraction d'un second type de parallélisme. Ces optimisations sont possibles grâce à l'environnement d'exécution KAAPI et à l'extension développée. Deux ordonnanceurs sont également comparés en quantifiant à l'exécution celui qui produit le meilleur placement (en terme d'accélération) des composants logiciels de la simulation.

I

Contexte technique

Ingénierie des Procédés Assistés par Ordinateur 2

2.1 Introduction

Le génie des procédés est globalement défini comme étant la mise au point des procédés destinés à produire une substance chimique donnée de façon économique et avec un impact minimal sur l'environnement.

Cette discipline scientifique pris naissance à la fin du $XIX^{ième}$ siècle sous l'action de George Davis. Cet Anglais militait pour la création d'une association des ingénieurs chimistes visant à promouvoir l'enseignement d'une science axée sur les opérations chimiques de transformations. Cette initiative n'eut guère de succès. Cependant, il continua cette science en concevant une série de 12 cours qui traversèrent l'Atlantique pour être introduit au programme d'enseignement du MIT.

En 1908, sous la forte croissance de cette discipline, l'*American Society of Chemical Engineers* (AIChE) fut créée. Cependant, malgré le regroupement de ces ingénieurs en association, il était nécessaire de spécifier ce qu'était un ingénieur chimiste et en quoi il était unique. En 1915, Arthur Little apporta la réponse par le concept latent d'*opération unitaire* initialement proposé par G. Davis. Cette heuristique suppose que tout procédé de fabrication peut être décomposé en une suite d'opérations élémentaires.

2.2 Simulations de procédés

Une simulation de procédés représente un problème physique par un modèle mathématique. Typiquement, une simulation de procédés est requise pour résoudre des problèmes liés à la conception, à l'optimisation, au dimensionnement ou bien à la formation d'opérateurs. Selon le type de problème, différentes stratégies de simulation existent. Par exemple, la vérification des contraintes de conception requiert un calcul stationnaire alors que l'analyse des effets de perturbations requiert un calcul transitoire. Si l'on considère une simulation stationnaire, le modèle mathématique est couramment représenté par un ensemble d'équations algébriques non-linéaires. Pour des problèmes plus complexes tels que les simulations transitoires, des équations aux dérivées partielles

et ordinaires apparaissent. Ainsi, chaque modèle mathématique nécessite un schéma numérique et une méthode de résolution.

L'ingénierie des procédés assistée par ordinateur vise à étudier et à prédire le comportement d'un procédé chimique de transformation au moyen d'un simulateur de procédés. Un simulateur de procédés permet à tout ingénieur de concevoir sa propre simulation à partir de briques élémentaires (notamment, les opérations unitaires) sans avoir nécessairement à développer de nouveaux programmes informatiques.

2.2.1 Définitions

Le CAPE (*Computer Aided Process Engineering*) repose sur la définition de trois concepts clés permettant de représenter une installation chimique :

- flux de matière ;
- opération unitaire ;
- schéma de procédé.

Les deux premiers représentent les éléments physiques atomiques de l'installation alors que le troisième est l'union des deux précédents. Un exemple représentant ces trois notions est présenté en figure 2.1.

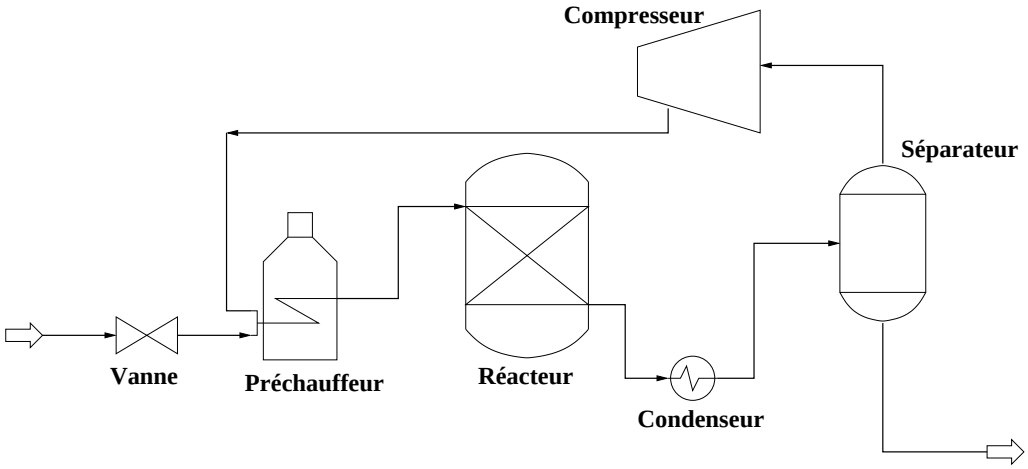


FIG. 2.1 – Schéma de procédé de la synthèse du cumène.

2.2.1.1 Flux de matière

Le fondement même du CAPE repose sur la définition de flux de matière permettant de caractériser les propriétés thermodynamiques et physico-chimiques de la matière au cours du processus de transformation. Les variables qui le caractérisent sont nombreuses (température, pression, fractions molaires de chaque phase, viscosité, ...), cependant, elles ne sont pas toutes nécessaires.

Le théorème de Duhem définit le nombre de variables strictement nécessaire à la détermination complète d'un flux. Il fut énoncé par Prigogine et Defay en 1954.

Théorème 2.2.1 *Quelque soit le nombre de phases, de constituants ou de réactions chimiques, l'état d'équilibre d'un système fermé, pour lequel nous connaissons les masses initiales totales de chaque constituant, est complètement déterminé par deux variables indépendantes.*

L'application de ce théorème permet de calculer, grâce à un serveur de calcul de propriétés thermodynamiques, l'ensemble des variables d'un système à partir de $n_c + 2$ variables indépendantes qui définissent sans aucune ambiguïté ce flux de matière. L'ensemble habituellement choisi est le suivant :

- débits molaires partiels des n_c constituants ;
- température (ou enthalpie dans le cas de corps purs) ;
- pression.

Le but d'un procédé chimique est de transformer un flux de matière initial représentant l'alimentation en un produit fini à plus forte valeur ajoutée. Ce processus de transformation est pris en charge par des unités de traitement spécialisées : les opérations unitaires.

2.2.1.2 Opération Unitaire

Une opération unitaire est un équipement physique prenant part à un processus complexe de transformation de la matière. La figure 2.1 présente l'ensemble des opérations unitaires nécessaire à la transformation d'une alimentation composée de benzène et de propylène en cumène.

L'opération unitaire renferme le savoir métier permettant de modéliser un équipement. Son fonctionnement est défini par un modèle mathématique qui permet de calculer les bilans matière et énergétique entre autres.

2.2.1.3 Schéma de procédé ou *flowsheet*

A priori, un schéma de procédé ou *flowsheet* ne qualifie que l'ensemble {opération unitaire, flux} sans pour autant modéliser un aspect fonctionnel. Cependant, nous verrons en section 2.2.4 qu'il définit une entité d'exécution en particulier en présence de recyclage.

Un recyclage est définit comme étant un effluent de sortie d'une opération unitaire qui est réinjecté en entrée (dans un but de préservation de l'environnement ou bien encore de rendement). Sur l'exemple de la figure 2.1, en sortie de réacteur, le flux de matière est composé de cumène, de propylène et de benzène dans des concentrations dépendant des conditions opératoires (pression, température). Le séparateur a pour objectif de réinjecter l'excédant de matière première en amont du procédé afin d'une part, d'optimiser le rendement et d'autre part, de répondre aux spécifications de synthèse, par exemple, obtenir un produit fini pur à 99% (ce qui est vrai dans le cas où la réaction n'est pas équilibrée où s'il existe des réactions concurrentes).

2.2.2 Problèmes de simulation

Au moyen de la représentation sous la forme d'un schéma de procédé, une simulation permet de répondre à différentes problématiques définissant des classes distinctes. Elles sont présentées par ordre de complexité croissante.

Un problème de « *simulation pure* » permet la résolution du modèle des opérations unitaires à partir des informations d'entrée, des conditions opératoires et des paramètres des équipements. Cela signifie qu'à partir des informations d'entrée, on cherche à déterminer les informations de sortie.

Un problème de conception vise à résoudre le modèle des opérations unitaires à partir de certaines informations d'entrée et de certaines informations de sortie. On cherche à déterminer les informations d'entrée et de sortie manquantes.

Un problème d'optimisation résout les modèles soumis à une fonction objectif et une série de contraintes. On cherche à déterminer les valeurs de quelques variables d'entrée telles que la fonction objectif soit minimisée (ou maximisée), et que quelques variables de sortie soient satisfaites.

2.2.3 Modes de simulation

Les simulateurs commerciaux proposent deux modes de résolution. Le premier, moins exigeant en puissance de calcul, permet de connaître l'état du *flowsheet* en régime permanent. Ce mode de résolution est dit **statique** ou **stationnaire**. Le modèle mathématique permettant de calculer le comportement d'une opération unitaire est basé sur un système d'équations algébriques non-linéaires.

Les simulations dynamiques ou **transitoires** représentent un enjeu majeur dans les simulateurs commerciaux actuels. Ce type de simulateur permet de prédire le comportement transitoire du procédé depuis un état initial à un état final. Il permet de simuler les phases de démarrage, d'arrêt, de redémarrage ou bien encore de perturbation d'un procédé. Le modèle mathématique s'exprime alors sous la forme d'un système d'équations différentielles ordinaires et/ou partielles selon la complexité de l'unité.

Outre le système d'équations modélisant les opérations unitaires, le conditionnement du problème (configuration des paramètres d'entrée du schéma de procédé) est différent selon le mode de simulation.

Pendant très longtemps, les simulateurs commerciaux n'étaient que statiques. La raison en est simple : les machines utilisées ne pouvaient subvenir aux besoins de calcul. De plus, ce mode de simulation permet de répondre à la majorité des problèmes de l'ingénierie des procédés. Malgré cela, l'étude d'un procédé s'oriente de plus en plus sur la prédiction de son comportement dans les phases transitoires telles que celles de démarrage et d'arrêt. De plus, pour un simulateur de formation, le caractère dynamique prend tout son sens. En effet, lors de l'injection d'une anomalie, il est alors possible d'en visualiser les conséquences sur les différents équipements au cours du temps. Un opérateur doit alors prendre les décisions nécessaires en fonction de ces observations. L'accroissement de la puissance de calcul des machines rend possible la conception de modèles

dynamiques qui s'exécutent en un temps acceptable.

Bien que peu utilisé auparavant, on constate que ce type de simulateur est en constante progression. Pour preuve, l'insertion du caractère dynamique dans les spécifications CAPE-OPEN (présentées en section 2.3.4.2) est sur le point d'être approuvée. Le développement et l'utilisation des simulateurs dynamiques joueront un rôle déterminant sur l'avenir du CAPE.

2.2.4 Méthodes algorithmiques de résolution

La résolution d'un schéma de procédé consiste principalement à calculer pour chaque opération unitaire les bilans matières et énergétiques. Pour y parvenir, plusieurs méthodes ont été définies : deux conceptuellement différentes et une troisième hybride.

Quelque soit le mode de résolution choisi (statique ou dynamique) les méthodes de résolution sont sensiblement identiques en pratique : seuls les modèles mathématiques caractérisant les opérations unitaires diffèrent. Le calcul transitoire d'un procédé revient à appliquer la même méthode de résolution que pour un calcul stationnaire à chaque pas de temps. La simulation dynamique tient compte de la dimension temps alors qu'une simulation statique résout un procédé lorsque ce dernier a atteint son régime permanent. Une résolution dynamique peut alors être considérée comme la solution itérative du schéma d'exécution d'une simulation statique. C'est pourquoi, dans cette section, nous ne ferons pas la distinction entre calculs dynamique et statique.

2.2.4.1 Approche modulaire séquentielle

L'approche modulaire séquentielle est l'approche la plus rencontrée dans les simulateurs de procédés. C'est également la méthode de résolution la plus intuitive. En effet, une opération unitaire est considérée comme une *boîte noire* qui offre des méthodes permettant de calculer les flux de sorties en fonction des entrées (flux de matière, conditions opératoires et paramètres d'équipement). Le schéma de procédés représente alors le diagramme de simulation définissant de manière abstraite un procédé chimique industriel. Ce graphe peut être cyclique ou acyclique (présence de recyclages) et les arcs sont orientés selon le sens physique de circulation des flux de matière dans le procédé.

Cette approche se prête particulièrement bien à la résolution de problèmes de simulation où chaque opération unitaire est calculée séquentiellement selon les contraintes d'écoulement.

Dans le cas d'une topologie acyclique, une seule itération est nécessaire afin de fournir la solution convergée. Chaque opération unitaire peut contenir un code itératif de résolution (voir figure 2.2).

L'exécution d'un opération unitaire nécessite que tous ses flux d'entrée soient connus. Or, un schéma de procédé cyclique ne peut vérifier cette propriété. En effet, les flux appartenant au recyclage ne peuvent être caractérisés qu'à l'exécution. Par exemple, sur la figure 2.3, l'opération *Mixer1* ne peut se calculer tant que toutes ses entrées ne sont pas disponibles. Or, pour que le flux *S6* soit défini, il est nécessaire que le module *Flash* soit calculé et par conséquent que *Mixer1* le soit également.

L'exécution d'une simulation avec recyclages requiert une phase de pré-traitement ayant pour but d'ordonner les exécutions des opérations unitaires en coupant les cycles. Pour cela, quatre étapes sont nécessaires :

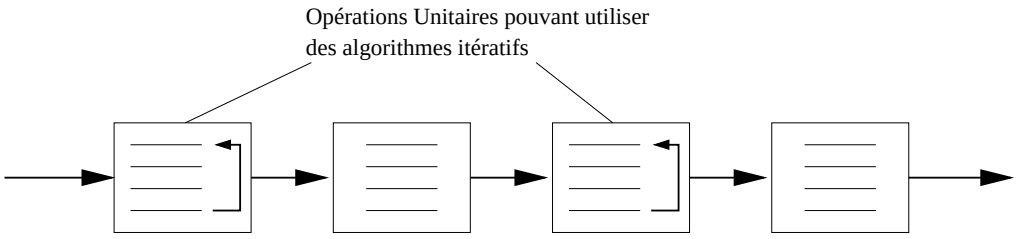


FIG. 2.2 – Résolution selon l'approche modulaire séquentielle pour un procédé séquentiel acyclique

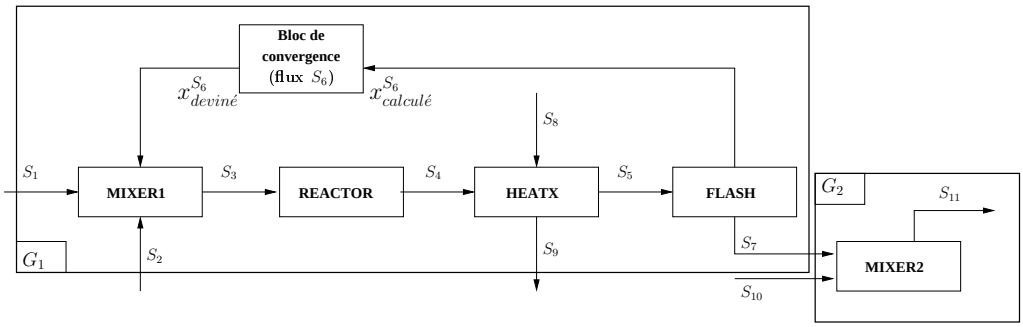


FIG. 2.3 – Schéma de procédé simple contenant un recyclage. Cela équivaut à insérer un *bloc de convergence* aux flux coupés (ici, le flux S_6).

1. Partitionnement (*partitioning*) : ce traitement vise à décomposer un schéma de procédé en plusieurs groupes irréductibles. Un groupe irréductible est formé d'opérations unitaires ayant une certaine affinité d'exécution : il est composé soit d'une seule opération unitaire soit d'un ensemble appartenant à un recyclage. Les opérations unitaires regroupées dans une telle partition sont intrinsèquement liées et doivent être exécutées *ensemble* avant le passage au groupe suivant. Sur la figure 2.3, le schéma de procédé expose deux groupes irréductibles G_1 et G_2 où $G_1 = \{\text{Mixer1, Reactor, HeatX, Flash}\}$ et $G_2 = \{\text{Mixer2}\}$.
2. Agencement (*ordering*) : cette étape calcule un ordre total d'exécution des partitions. Sur notre exemple, l'exécution du groupe G_1 doit précéder l'exécution du groupe G_2. En effet, G_2 nécessite les flux S_{10} (alimentation du procédé) et S_7 calculé par G_1.
3. Coupe (*tearing*) : cette étape vise à analyser chaque partition afin de détecter les flux à couper. Plusieurs algorithmes permettent de calculer cet ensemble selon un objectif particulier. Par exemple, les travaux présentés dans [7] se basent sur la coupe du nombre minimum de flux alors que les auteurs de [111] coupent un ensemble de flux exposant les meilleures propriétés de convergence. Quelque soit l'objectif, plusieurs ensembles peuvent être valides. Pour le cas présenté en figure 2.3, en considérant l'objectif de minimisation de la coupe, chacun des flux S_3, S_4, S_5 et S_6 peut être coupé.
4. Séquençage (*sequencing*) : cette dernière étape calcule l'ordre total d'exécution des opé-

rations unitaires pour chaque partition où il existe des flux coupés. Dans le groupe G_1 de la figure 2.3 où le flux S_6 est coupé, le séquençage des opérations est le suivant : $Mixer1 < Reactor < HeatX < Flash$ où $<$ définit la relation de précédence.

De nombreux travaux se sont intéressés à ces techniques de pré-traitement. Le lecteur intéressé est renvoyé à [119] pour de plus amples informations.

Les étapes d'agencement et de séquençage permettent de spécifier un ordre total d'exécution sur l'ensemble des opérations unitaires du schéma de procédé. L'exécution repose alors sur un calcul de type itératif. Les variables définissant complètement les flux coupés sont dites *devinées* afin de débloquer l'exécution. Le calcul séquentiel est alors accompli et les valeurs *devinées* sont comparées aux valeurs effectivement calculées. Si leur différence est en deçà d'un certain seuil de tolérance alors le résultat est accepté et la prochaine partition dans la liste est exécutée selon la même technique. Comme présenté en figure 2.3, cela revient à placer une bloc logique de convergence en charge de calculer l'erreur et de proposer un nouvel ensemble de valeurs pour le flux coupé.

Il existe quelques méthodes permettant de proposer de nouvelles valeurs aux flux coupés. La plus triviale est appelée méthode par substitution successive où les données calculées à l'itération n sont proposées à l'itération $n + 1$:

$$x_{n+1} = F(x_n)$$

Une autre méthode permettant d'atteindre une convergence plus rapide est la méthode de Wegstein [117] où les nouvelles valeurs proposées sont fonction des deux itérations précédentes :

$$x_{i+1} = F(x_i, x_{i-1})$$

Ce schéma itératif présente deux niveaux d'exécution : un niveau module où chaque opération unitaire est résolue et un niveau schéma de procédé (correspondant au bloc logique de convergence) en charge de faire converger les flux coupés.

Jusqu'à présent, seuls les problèmes de simulation, où tous les paramètres d'entrée (alimentation, paramètres d'équipement et conditions opératoires) sont connus, ont été traités. En l'état, l'approche modulaire séquentielle ne permet pas de résoudre un problème de conception où certaines variables de sortie sont fixées. La résolution d'un tel cas nécessite l'ajout d'une boucle de contrôle au niveau schéma de procédé dont le but est de faire varier un paramètre d'équipement qui influe directement ou indirectement sur la contrainte de spécification fixée (par exemple, faire varier une ouverture de vanne pour augmenter un débit). Dans le cas où le schéma de procédé présente des recyclages, boucles de convergence et de contrôle sont exécutées séparément. En effet, la différence de structure des équations associées aux flux de recyclage et des équations de spécifications conduit naturellement à leur distinction au niveau du traitement numérique. Cette démarche est également utilisée pour les méthodes d'optimisation où une boucle externe est ajoutée.

Afin de répondre à ces contraintes, l'approche orientée équations est proposée comme alternative.

2.2.4.2 Approche orientée équations

Contrairement à l'approche modulaire séquentielle, cette méthode résout simultanément la totalité du schéma de procédé. Il n'existe plus qu'un niveau d'exécution : le niveau schéma de procédé. Pour cela, toutes les équations des modèles décrivant les opérations unitaires sont assemblées. L'ensemble ainsi formé est associé aux équations de connexions qui définissent les interactions entre les opérations unitaires. Si un problème de conception est à résoudre, les équations de spécifications sont également ajoutées au système. L'approche orientée équations résout simultanément le système.

Cette méthode de résolution met en œuvre des techniques numériques adaptées au traitement de systèmes de grande dimension et creux. La décomposition du système n'est plus forcément faite par référence aux groupes d'équations correspondant aux modèles des unités, comme dans l'approche modulaire séquentielle, mais par analyse de la matrice d'incidence du système complet des équations. Rappelons que le système est non-linéaire si bien que le cœur de l'approche orientée équations se base sur un traitement itératif nécessitant un solveur (*e.g.* Newton-Raphson). Ce solveur doit être polyvalent, contrairement à l'approche modulaire séquentielle où un solveur spécifique adapté aux spécificités du modèle peut être employé.

Cette approche fut présentée comme une alternative à l'approche modulaire séquentielle. Cependant, pendant un certain nombre d'années, la puissance de calcul et de stockage (mémoire) des machines n'était pas suffisante pour répondre aux exigences de l'approche orientée équations. Avec la puissance actuelle des machines *personnelles*, cette technique peut être appliquée (elle requiert cependant une forte capacité de mémoire). Un problème persiste cependant au niveau de la robustesse des méthodes de convergence. En effet, si les valeurs initialement prévues sont trop éloignées des valeurs physiques réelles, il est possible que le système ne converge pas. De plus, il est possible de rencontrer un problème de minimum local et d'obtenir une solution erronée.

2.2.4.3 Approche hybride : l'approche modulaire simultanée

L'approche hybride a été mise au point pour pallier aux inconvénients des approches orientée équations et modulaire séquentielle. Le but de ce document n'étant pas de classifier les stratégies de résolution, le lecteur est renvoyé à [120, 8] où l'étude comparative des deux approches est présentée en détail.

L'approche modulaire simultanée [28] conserve le concept de module où chaque opération unitaire utilise les algorithmes les mieux adaptés à sa résolution. Elle s'approprie la rapidité de convergence de l'approche orientée équations particulièrement adaptée à la résolution de problèmes de conception et d'optimisation. Pour cela, les opérations unitaires ne sont plus considérées comme des boîtes noires mais « *grises* ».

Le schéma d'exécution de l'approche modulaire simultanée résout simultanément les équations décrivant les flux coupés et les contraintes de spécification. Pour cela, elle nécessite la matrice de sensibilité (ou matrice jacobienne) de chaque variables caractérisant complètement le flux

coupé. Cette matrice est soit obtenue analytiquement soit approximée selon différentes méthodes (*e.g.*, par différences finies). Deux niveaux sont définis :

– un modèle orienté équations approché
– un modèle procédural rigoureux

Ainsi, à chaque itération, un modèle orienté équations du réseau est résolu dans le but de générer de nouveaux candidats pour le modèle rigoureux. Le modèle rigoureux est alors évalué pour générer les nouvelles valeurs des inconnues dans le modèle approché (voir figure 2.4).

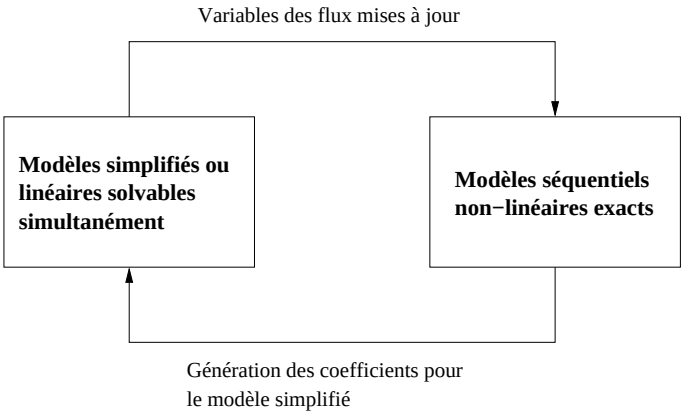


FIG. 2.4 – Approche modulaire simultanée : un schéma d'exécution en deux étapes.

Afin de garantir un certain niveau de performance, une attention particulière est portée sur la construction de la jacobienne ou de son approximation. En effet, si celle-ci est trop *précise*, l'approche modulaire simultanée peut être mois performante que l'approche modulaire séquentielle. *A contrario*, si elle est construite à partir d'approximations grossières, la résolution peut échouer (convergence non atteinte) ou requérir un grand nombre d'itérations pour converger.

Cette approche est implantée dans quelques simulateurs commerciaux où les opérations unitaires fournissent non seulement les sorties mais également les matrices de sensibilité associées (dans ProSimPlus par exemple). Si ce n'est pas le cas, cette approche peut néanmoins améliorer le temps de résolution pour les problèmes de conception et d'optimisation évitant ainsi la multiplicité des boucles de l'approche modulaire séquentielle.

2.2.5 Besoin d'intégration de composants tiers

De nombreuses entreprises ont, à bon escient, adopté un mode de programmation modulaire. Cette approche permet une meilleure répartition des tâches en fonction des compétences de chaque équipe. Elle nécessite la mise en place de spécifications permettant d'intégrer au mieux les différentes composantes de l'application. Etant donnée le concept d'opération unitaire, le domaine de l'ingénierie de procédés assistée par ordinateur se prête naturellement à ce mode de programmation. Les plus grands simulateurs intègrent un ensemble d'opérations unitaires qui peuvent facile-

ment être connectés afin de former un schéma de procédé.

Néanmoins, il s'avère utile pour tout ingénieur ou chercheur de pouvoir intégrer aisément dans un simulateur une opération unitaire spécifique externe. Ceci est plausible étant donné que tout environnement de simulation expose ses interfaces propriétaires. Cependant, il existe autant d'interfaces propriétaires que de simulateurs commerciaux. De même, le savoir-faire de certaines entreprises repose sur la conception de modèles métier hautement spécialisés simulant, par exemple, une opération unitaire spécifique telle qu'un pipeline ou une membrane de diffusion. Ces opérations ne font pas partie intégrante de la librairie d'opérations unitaires fournies en standard avec les environnements de simulation commerciaux. Il devient intéressant pour ces entreprises de ne plus être attachées à un environnement de simulation particulier. Leur savoir-faire peut être intégré dans tout environnement.

La nécessité de définir un standard mettant en avant l'intégration de composants tiers est pressante. Le standard CAPE-OPEN fut instauré pour répondre à ce besoin.

2.3 Standard CAPE-OPEN

Le standard CAPE-OPEN (pour *Computer Aided Process Engineering OPEN*) est largement reconnu dans la communauté de l'ingénierie des procédés, ceci malgré sa jeunesse (projet débuté en 1995). Pour preuve, les principaux éditeurs de simulateur tels que SimSci-Esscor, AspenTech ou bien encore RSI, respectivement avec leur simulateur PRO/II, AspenPlus ou Hysys et INDISS, ont adopté ce standard. Le CO-LaN (pour *CAPE-OPEN Laboratories Network*) supporte et maintient le projet afin de promouvoir la réutilisabilité des codes en adoptant une approche orientée composants. Le standard CAPE-OPEN vise ainsi à définir des spécifications d'interfaces pour les entités métier du CAPE. Le standard fixe une décomposition des techniques de simulation du CAPE en deux entités distinctes : les *Composants de Modélisation de Procédés* (PMC / *Process Modelling Component*) et les *Environnements de Modélisation de Procédés* (PME / *Process Modelling Environment*).

2.3.1 Définitions

Cette section définit les quelques notions informatiques qui ont motivé l'élaboration du standard CAPE-OPEN.

2.3.1.1 Réutilisabilité

Le concept de réutilisabilité est apparu dès les premières améliorations des langages de programmation. Avec l'utilisation des langages procéduraux, la volonté de rendre les codes modulaires est apparue. Un code modulaire permet de définir un ensemble de fonctionnalités que l'on regroupe en module. Avec l'émergence de la programmation orientée objets, ce concept s'est renforcé en définissant explicitement la notion de module sous la forme d'un objet.

Le regroupement des codes par fonctionnalités sous la forme de librairies a fortement contribué à les réutiliser. En pratique, ceci est réalisé au moyen d'une « *interface* » d'accès sous la forme d'un prototype de méthode. Cette notion de réutilisabilité n'est pas totalement transparente au

développeur. En effet, un code est écrit suivant un langage de programmation spécifique et ne constitue pas une unité d'exécution à proprement parler.

2.3.1.2 Couplage de codes

Le couplage de codes permet à tout code de s'intégrer dans un environnement d'exécution et d'être facilement « *commandable* ». Pour cela, ils doivent définir un contrat permettant à chacun de connaître leurs fonctionnalités. Cette notion de contrat est apportée *via* les techniques de **programmation orientée composants**.

L'approche composant spécifie des interfaces normalisées d'accès aux objets exportant certaines méthodes. Ces interfaces sont accessibles au mode extérieur et peuvent être invoquées par d'autres composants. Cette spécification d'interfaces (objet et méthodes exportées) est régie au travers d'un langage de description d'interfaces nommé IDL (*Interface Description Language*) indépendamment de tout langage de programmation.

La définition du contrat IDL permet à tout composant exposant les mêmes spécifications qu'un autre d'être substitué à ce dernier : les codes sont dits interchangeables.

2.3.1.3 Interchangeabilité

Le concept d'interchangeabilité permet à tout binaire d'être remplacé en l'état par un autre répondant aux mêmes spécifications. Ces spécifications sont définies par le contrat IDL.

Considérons, par exemple, un composant permettant de résoudre un système linéaire. Il est possible d'utiliser un composant solveur basé sur une méthode de résolution soit de type itératif, soit direct. Ces deux composants présentent la même fonctionnalité qui est de résoudre le système linéaire. Ainsi, pour peu que les deux composants exposent la même interface et, par conséquent, les mêmes méthodes, il devient possible d'utiliser l'un ou l'autre selon le problème à traiter sans avoir à modifier le code appelant.

2.3.2 Historique

En 1994, Peter Banks (British Petroleum) se rend à la conférence FOCAPD à Snowmass dans le Colorado en ayant en tête les visions futures de BP en ce qui concerne les simulations de procédés. En considérant la complexité croissante des modèles de simulation, l'une des préoccupations de BP est de voir apparaître un système *plug and play* permettant de connecter une entité fonctionnelle de calcul de source quelconque dans un simulateur. Cette dernière doit collaborer à la simulation sans passer par les méthodes courantes de mise à niveau du code. Cette conférence, regroupant de nombreux acteurs de la communauté du CAPE, est le lieu de prédilection pour débattre de ce concept. Cependant, la communauté n'est guère réceptive aux arguments du chercheur. Toute la communauté ? Non ! Hans-Horst Mayer (BASF) se trouve être fort intéressé par une telle possibilité. L'histoire CAPE-OPEN est en marche.

En 1995, BP obtient le support du projet européen PRIMA (PRocess Industry Manufacturing Advantage). Parallèlement, le consortium allemand est sur le point de soumettre une proposition similaire à la suite de l'intervention de Hans-Horst. Ainsi, les deux propositions fusionnent pour donner naissance à l'initiative *Object Oriented CAPE* (OO-CAPE) sous la direction de Siegfried

Nagel (Bayer). En 1996, un consortium est alors créé, regroupant des utilisateurs, des développeurs et des universitaires issus du monde CAPE. L'initiative se nomme à présent *Open Standards for CAPE* (OS-CAPE) et la direction est confiée à Bertrand Braunschweig (IFP). Il devient alors nécessaire de faire approuver le projet par l'Europe, ce qui donne naissance, en 1997, au projet européen nommé *CAPE-OPEN*.

Mi 1999, à la fin du projet CAPE-OPEN, de nombreux progrès sont effectifs mais la majorité de la communauté reste dans l'expectative et pense qu'ils ne sont pas probants. Certains affirment que le seul résultat prouvé est une « semi-intégration » sur une démonstration faisant coopérer un « *mixer-splitter* » avec un serveur de calculs thermodynamiques de deux sources différentes : AspenTech et Hyprotech. En réalité, les résultats ne se cantonnent pas à cette démonstration. En effet, des modèles d'opérations unitaires de l'IFP en France, le solveur du PME orienté équations gPROMS [14] de l'Imperial College de Londres, la bibliothèque thermodynamique IK-CAPE [33] en Allemagne et un outil de partitionnement et de séquençage réalisé par l'INPT à Toulouse ont été intégrés, suivant les concepts définis dans le projet, dans la plate-forme CHEOPS [101] de RWTH Aachen. En outre, l'un des résultats les plus fondamentaux est la sortie publique de la pré-version 0.9 du standard CAPE-OPEN sous la forme de spécifications d'interfaces. On peut également dénombrer deux autres résultats d'importance à la suite du projet CAPE-OPEN :

1. Le projet *Global CAPE-OPEN* (GCO) fait suite au projet CAPE-OPEN. Ce projet a pour but de poursuivre les actions menées afin de développer un standard CAPE-OPEN fonctionnel. Cette action mène à la version 0.9.0 du standard qui sera implantée dans les environnements de simulation Aspen Plus et HYSYS fin 1999.

2. Un consortium regroupant des utilisateurs, des développeurs de logiciels et des universitaires est créé afin de faire évoluer le standard en fonction des besoins de la communauté.

Dès la fin du projet CAPE-OPEN, le projet *Global CAPE-OPEN* (GCO) (soutenu par la commission européenne dans son action Brite-EuRam, et endossé par le programme IMS) prend la relève de 1999 à 2002. L'objectif premier de ce projet est de promouvoir le standard CAPE-OPEN au niveau international. De plus, de nombreux progrès sont apportés au standard aussi bien d'un point de vue intégration des codes que de l'enrichissement et de l'amélioration des spécifications d'interfaces (version 1.0). Cependant, ce projet a une fin et la nécessité de créer une organisation supportant le standard se fait ressentir : « *la durée de vie des standards doit dépasser celle de ces projets, et il est donc nécessaire de mettre en place une instance chargée de leur gestion, de leur diffusion, de leur évolution, et de la certification de composants* » [22]. C'est pour répondre à ce besoin que les partenaires décident de créer, début 2001, un organisme à but non lucratif : le *CO-LaN*. Parallèlement à la création du CO-LaN, un nouveau projet européen voit le jour de mi 2001 à mi 2003 sous le nom de *Global CAPE-OPEN support* (GCO-support).

Le projet CAPE-OPEN est libre de devenir un grand standard dans le domaine de l'ingénierie des procédés assistée par ordinateur. A la fin du projet GCO, plus d'une trentaine de partenaires font partie du consortium et ceci sur trois continents (Europe, Etats-Unis et Japon).

2.3.3 Un découpage métier

Afin de représenter un procédé, le standard définit deux entités conceptuelles permettant de qualifier complètement une simulation de procédés. Chacune de ces entités est associée aux concepts métier CAPE définis précédemment.

D'un point de vue informatique, ces entités sont des composants. Le CO-LaN diffuse les spécifications d'interfaces pour les *middleware* DCOM [21] et CORBA [48]. Ces deux technologies implantent les concepts de réutilisabilité, intégration et interopérabilité définis précédemment.

2.3.3.1 Composant de Modélisation de Procédés

Les PMC définissent des entités métier fonctionnelles. Afin de créer une application au standard CAPE-OPEN, il est possible de restreindre l'étude à trois types de composants qui schématisent le comportement d'une simulation :

1. **Physical Property Package (PPP)** gère les calculs thermodynamiques tels que les calculs de flash (calcul de l'équilibre entre les phases (vapeur / liquide) du flux) ou de viscosité au moyen de l'interface `ICapeThermoPropertyPackage` ;
2. **Unit Operation (UO)** représente une opération unitaire de traitement physique telle qu'une colonne de distillation ou bien plus simplement une vanne. L'interface représentant cet objet est `ICapeUnit`.
3. **Solveur** est l'entité implémentant la méthode de convergence basée sur un solveur particulier (généralement de type Newton). L'interface CAPE-OPEN est `ICapeNumericSolver`.

Afin de réaliser le lien entre les composants UO et PPP, un troisième composant est nécessaire : *Material Object* (MO). Ce composant peut être directement superposé au concept de flux de matière, il représente l'état de la matière (composition, température, débit, pression, ...) à un instant t pour un flux reliant deux UO. Il est également le point d'accès aux calculs thermodynamiques.

Techniquement parlant, les composants MO sont des conteneurs de données fortement couplés aux composants UO sur lesquels ils interagissent. Tout composant MO détient une référence sur un serveur de calculs de propriétés thermodynamiques et transmet les appels initiés par les UO au serveur.

2.3.3.2 Environnement de Modélisation de Procédés

Le standard CAPE-OPEN définit très peu de spécifications pour les environnements de modélisation de procédés (PME). La seule fonctionnalité imputée à un PME est la gestion des interactions entre l'utilisateur final et les PMC. Un PME offre donc plusieurs fonctionnalités en rapport avec l'utilisateur :

1. Composition : permet de créer les PMC nécessaires à la définition du schéma de procédé. L'utilisateur choisit parmi un ensemble d'opérations unitaires, celles qui composeront son procédé chimique. Il choisit également le serveur de calcul de propriétés thermodynamiques à utiliser.

2. Configuration : permet de configurer l'ensemble des PMC du schéma de procédé. Cette fonctionnalité gère l'affectation des paramètres d'équipement, le choix de la méthode de résolution ainsi que la caractérisation de l'alimentation et autres flux de matière du schéma de procédé.
3. Exécution : contrôle l'avancement de la simulation : démarrer, arrêter, suspendre ...
4. Diagnostic / Visualisation : outils de diagnostic dont les messages sont générés par les différents PMC et outils de visualisation permettant de rendre compte du déroulement de la simulation ainsi que de ses résultats.

Un PME est l'entité en charge de la création et du référencement de tout les PMC. De ce fait, le standard CAPE-OPEN a naturellement défini son modèle conceptuel autour de cette entité.

2.3.4 Standard CAPE-OPEN en pratique

Dans cette section, les méthodes exposées par le standard CAPE-OPEN sont présentées afin de répondre aux exigences des méthodes de résolution présentées en section 2.2.4.

2.3.4.1 Simulations statiques

Quelque soit l'approche retenue, toute opération unitaire doit implanter l'interface `ICapeUnit`.

L'approche modulaire séquentielle est spécifiée dans le standard CAPE-OPEN, *via* le *package Sequential Modular Specific Tools* (SMST) [10]. En pratique, les simulateurs commerciaux n'utilisent pas cette fonctionnalité et implantent leurs propres modules de partitionnement / agencement / coupe de flux / séquençage. Ceci se justifie par le fait que ces techniques n'ont guère évolué ces dernières décénies. Avant l'émergence du standard, tous les environnements de simulation implantaient nativement ces fonctionnalités et peu ont trouvé d'intérêt à le rendre conforme au standard. L'intérêt du standard réside principalement dans l'intégration de PMC de type opération unitaire et de type serveur de calculs thermodynamiques.

Pour exécuter un calcul stationnaire selon l'approche modulaire séquentielle, l'interface `ICapeUnit` expose la méthode `Calculate` indiquant à toute opération unitaire de calculer ses sorties en fonction de ses entrées selon son propre modèle de résolution. La figure 2.5 présente le diagramme de séquence de l'appel à cette méthode sur l'interface `ICapeUnit`. On constate que de nombreux composants sont requis à la complétion de `Calculate` : certains sont dits utilitaires (*Port* / *Collection* / *Parameter*) et d'autres métier (*MaterialObject* et *PhysicalPropertyPackage*). Chaque UO est intrinsèquement liée aux composants utilitaires. Les calculs thermodynamiques sont, quant à eux, gérés par les MO. Ces derniers sont des conteneurs de données thermodynamiques telles que la pression ou encore la composition du flux. Une requête d'exécution d'un calcul de propriétés thermodynamiques sur ce conteneur entraîne l'appel des méthodes de calcul du serveur thermodynamique. Le serveur récupère ainsi les données dont il a besoin, réalise les calculs et dépose les résultats ainsi obtenus dans le MO concerné. Ainsi, il existe un couplage fort entre les composants métier suivant l'ordre logique UO $\Rightarrow$ MO $\Rightarrow$ PPP.

Dans un pré-traitement, le PME réalise le séquençage des opérations unitaires en fournissant un ordre total d'exécution selon les techniques précédemment citées en 2.2.4.1. Il est bon de noter

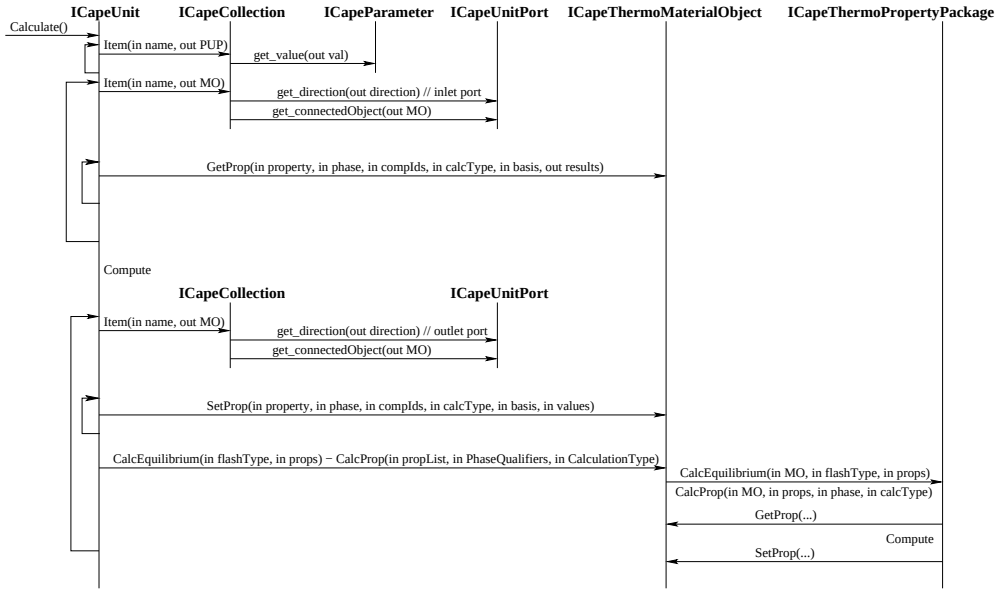


FIG. 2.5 – Diagramme de séquence de la méthode `Calculate` invoquée sur une opération unitaire [6] (*modifié*).

que le standard ne spécifie rien en ce qui concerne la résolution à proprement parler. Il préconise que le composant solveur gère le flot de contrôle de l'exécution selon l'ordre total d'exécution. Seules les fonctionnalités liées à l'invocation de la méthode `Calculate` sont spécifiées. Son invocation met à jour les *Material Object* de sortie en fonction des *Material Object* d'entrée, des paramètres d'équipement et des conditions opératoires.

L'approche orientée équations peut être implantée selon le standard en greffant l'interface `ICapeNumericESO` à l'interface `ICapeUnit`. Bien que soumise aux spécifications du standard, cette interface n'est que très peu utilisée en réalité (l'implémentation est difficile à concevoir). Le lecteur intéressé par la description et la mise en application des méthodes associées est renvoyé à [3, 87].

2.3.4.2 Simulations dynamiques

En l'état, une simulation dynamique peut aussi bien être résolue selon les approches modulaire séquentielle ou orientée équations. Cependant, la caractère itératif d'un calcul dynamique (selon la dimension temps) limite l'utilisation de l'approche modulaire séquentielle. Ainsi, à ses débuts, le standard CAPE-OPEN préconisait une méthode de résolution basée sur l'approche orientée équations. Loin de se satisfaire de cette méthode (et étant donné que peu d'industriels implantent cette technique au sein de leur environnement de simulation), le standard s'est orienté sur la définition d'une approche hybride en étendant l'interface standard `ICapeUnit` en `ICapeDynamicUnit`.

Cette interface propose trois méthodes pleinement (et volontairement) inspirées de la résolution dynamique de INDISS (RSI). Bien que fortement couplé à la méthode de résolution, le

standard ne fait que définir des interfaces métier offrant certains services de calcul. Cette section a pour but de les définir. La section 6.2 présentera la résolution implantée au sein de l'environnement de simulation INDISS.

L'interface `ICapeDynamicUnit` est une extension de l'interface `ICapeUnit` exposant alors la méthode `Calculate`. Dans le cas de simulations dynamiques, cette méthode permet à une opération unitaire de finaliser un pas de temps. Pour cela, le calcul des variables thermodynamiques des flux de sortie et des autres quantités internes à chaque UO est réalisé. Couramment, on dit que l'opération unitaire réalise son bilan énergétique.

`ICapeDynamicUnit` propose particulièrement deux méthodes spécifiques à la résolution dynamique : `StartTimeStep` et `ResolveFlowPressure`.

La méthode `StartTimeStep` indique à une opération unitaire que l'exécution d'un nouveau pas de temps est en cours. Le standard recommande que l'invocation de cette méthode engendre le calcul de toutes variables internes indépendantes du calcul débit-pression.

La méthode `ResolveFlowPressure` d'une opération unitaire résout la relation débit-pression en fonction du jeu de pressions appliqué sur le *Material Object* associé.

L'implantation de cette méthode diffère selon le type de l'opération unitaire. Le standard définit principalement deux classes d'opérations unitaires : les nœuds et les arcs.

Les arcs (*e.g.* vanne) modélisent des équipements sans accumulation possédant une entrée et une sortie et caractérisés par une perte de charge. Sans accumulation, les pressions en entrée/sortie sont différentes et le débit d'entrée est égal au débit de sortie selon la loi de conservation de la masse :

$$F_{\text{out}} = F_{\text{in}} + \text{accumulation}$$

La méthode `ResolveFlowPressure` invoquée sur ce type d'opération calcule le débit ainsi que ses dérivées par rapport aux pressions d'entrée et de sortie. Les dérivées seront utilisées afin de construire la matrice Jacobienne pour estimer les nouvelles pressions de chaque flux du procédé.

Les nœuds (*e.g.* mixer) représentent des unités pouvant posséder plusieurs entrées et plusieurs sorties. Ce type d'unité possède une accumulation, de ce fait, la somme des débits d'entrée est différente de la somme des débits de sortie. En fonction de la pression identique en entrée/sortie, ce type d'opération unitaire calcule sa compressibilité (accumulation à un instant t) ainsi que sa dérivée par rapport à la pression lors de l'appel de la méthode `ResolveFlowPressure`.

Le tableau 2.1 résume les propriétés de ces deux types d'opérations.

	Arc	**Nœud**
Débit	$F_{in} = F_{out}$	$F_{in} \neq F_{out}$
Pression	$P_{in} \neq P_{out}$	$P_{in} = P_{out}$
Accumulation	$\emptyset$	Oui

TAB. 2.1 – Propriétés des classes d'opérations unitaires pour la résolution dynamique

Pour représenter ce typage, le standard CAPE-OPEN spécialise l'interface `ICapeDynamicUnit` en `ICapeArcDynamicUnit` et `ICapeNodeDynamicUnit`. La figure 2.6 présente le diagramme de classe UML de la spécialisation `ICapeUnit`. Cette distinction d'interface repose sur la méthode d'accès aux données calculées par `ResolveFlowPressure`. Alors qu'un arc définit la méthode `GetFlowAndDerivatives`, un nœud expose la méthode `GetCompressibilityAndDerivative`.

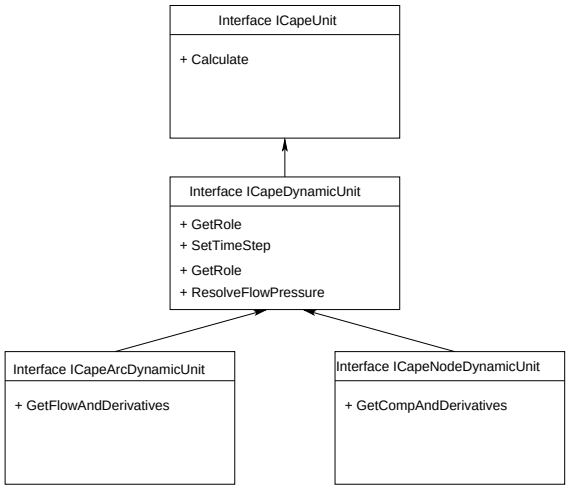


FIG. 2.6 – Diagramme de classe UML : extension de l'interface `ICapeUnit`.

La section suivante présente quelques environnements de simulation répondant en partie à la norme CAPE-OPEN.

2.3.5 Environnements au standard CAPE-OPEN

De nombreux simulateurs sont, à présent, conformes aux recommandations du standard. Cependant, ils ne respectent pas en totalité la norme. En effet, la majorité d'entre eux expose seulement une compatibilité sur l'intégration de composants tiers de type opération unitaire et serveur de calculs de propriétés thermodynamiques.

Etant donné que les recommandations du standard pour les interfaces dynamiques ne sont pas encore officiellement diffusées, seuls les environnements de simulation INDISS (RSI) et D-SPICE (Fantoft) les implantent. Ces deux entreprises sont les acteurs majeurs à la définition du caractère dynamique pour le standard CAPE-OPEN. Ainsi, la plupart des environnements de simulation autorise uniquement l'intégration d'opérations unitaires implantant l'interface `ICapeUnit` utilisées lors du calcul stationnaire.

PRO/II [64] (SimSci-Esscor), Hysys et AspenPlus [78] (Aspen Tech) et COCO [115] implémentent une méthode de résolution selon l'approche modulaire séquentielle nécessitant l'interface

`ICapeUnit` sur les opérations unitaires CAPE-OPEN.

AspenPlus permet également l'intégration d'opérations unitaires exposant l'interface `ICape-NumericESO` pour la résolution du procédé selon une approche orientée équations.

Seul ProSimPlus [112] définit une approche modulaire simultanée *via* l'intégration d'opérations unitaires CAPE-OPEN offrant l'interface `ICapeUnit`. Cette interface ne permet pas d'obtenir la matrice de sensibilité si bien que l'environnement est contraint de calculer une approximation par différences finies. Cette méthode est employée avec succès afin de traiter des problèmes de conception ou d'optimisation pour lesquelles cette approche est performante.

2.3.6 Etat d'avancement du projet

Depuis 1995, le standard CAPE-OPEN a réalisé un long chemin pour que les principaux environnements de simulation commerciaux du génie des procédés adoptent ses recommandations et spécifications d'interfaces (les environnements Hysys, AspenPlus et PRO/II représente 96% du marché pour le stationnaire). Le projet, actuellement mature, est largement reconnu dans le domaine de l'ingénierie des procédés assistée par ordinateur.

Le CO-LaN est le premier acteur de l'avancement du standard CAPE-OPEN. Dès sa création, de nombreux groupes de travail ont été créés : les SIG (Special Interest Groups). Ils permettent de faire vivre le standard en améliorant l'existant et en développant de nouvelles interfaces sur les propositions des membres. Au nombre de cinq, ils sont regroupés selon les thématiques suivantes :

1. **Thermodynamique :** implanter, entretenir et gérer les interfaces existantes ; extension pour les polymères et les solides.
2. **Interopérabilité :** tester l'interopérabilité et l'intégration des logiciels.
3. **Opérations unitaires :** implanter, entretenir et gérer les interfaces existantes ; extension au calcul dynamique.
4. **Méthodes et outils (en création) :** prendre en compte l'évolution vers le *middleware* .Net.
5. **Réacteur (création mi 2006) :** étendre le standard pour gérer les réacteurs de raffinage et l'ingénierie des réacteurs. Développer un prototype. S'intéresser aux coupes pétrolières.

Ces groupes de travail analysent les limitations actuelles du standard et apportent, le cas échéant, des modifications et extensions sur les spécifications d'interfaces. Par exemple, le CO-LaN vient de sortir une révision des interfaces thermodynamiques en réponse aux travaux du SIG Thermodynamiques.

Les SIG sont les points d'entrée avant tout processus de standardisation. Il est nécessaire au maintien de la cohérence globale du standard. Parallèlement à ces SIG, de nombreux laboratoires de recherche ainsi que des industriels s'investissent dans l'évolution du standard. Actuellement, plus d'une cinquantaine de membres adhèrent au CO-LaN mais aussi des organisations, associations et groupes d'utilisateurs tels que le groupe français IEP (Informatique & Procédés), la Société de Chimie Industrielle (SCI) et de la Société Française de Génie des Procédés (SFGP).

Le standard CAPE-OPEN répond aux objectifs d'intégration de composants tiers au sein d'environnements de simulation. Le domaine de l'ingénierie des procédés assistée par ordinateur peut bénéficier des progrès du standard. Néanmoins, à l'instar de tout développement d'applications de simulation numérique, cette science est limitée par la puissance des machines et ceci malgré l'accroissement constant de la puissance processeur.

2.4 Besoin de ressources de calcul

Depuis toujours, l'ingénierie des procédés assistée par ordinateur est confrontée aux limites des ressources de calcul des ordinateurs. Ainsi, les simulations de procédés reposent sur le compromis entre précision des modèles et temps de calcul nécessaire à leur résolution. L'accroissement de la puissance des ordinateurs personnels a engendré le développement de modèles plus réalistes. Malgré cela, la complexité des modèles est toujours en deçà des besoins. La simplification des modèles est alors indispensable afin de conserver un temps de résolution acceptable.

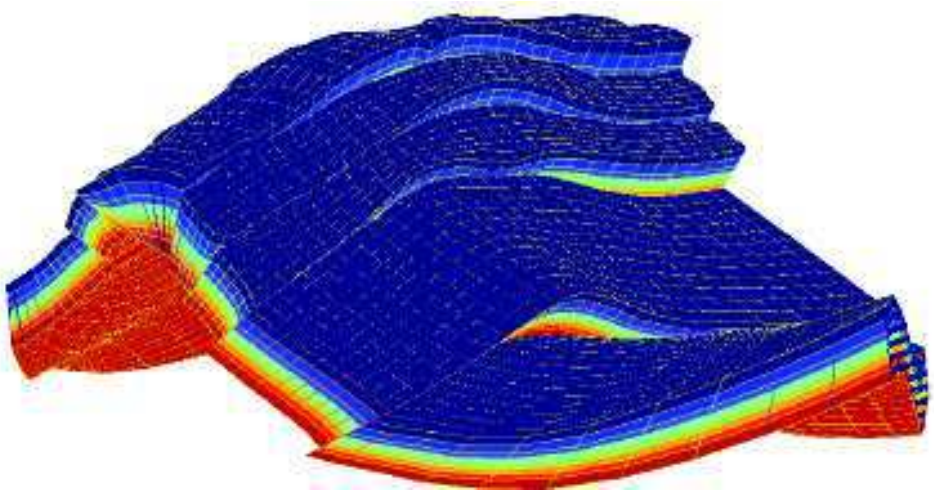


FIG. 2.7 – Maillage d'un réservoir où chaque couleur représente un intervalle de température.

Le modèle mathématique régissant le comportement d'un procédé est généralement basé sur une discrétisation spatiale et/ou temporelle. Plus le pas de discrétisation est petit, plus la précision du modèle est grande. La figure 2.7 présente un exemple de discrétisation spatiale sur un réservoir. Le grain du maillage influe directement sur la précision de la simulation modélisant les écoulements poly-phasiques.

Les exigences sur les propriétés à simuler s'accroissent de plus en plus afin de répondre, par exemple, aux besoins de maintenance d'un procédé. Par exemple, si l'on considère un *pipeline* localisé en eau profonde, il est important d'estimer le risque de formation d'hydrates dans les conduites. Ce critère est important car une *flowline*[1] obstruée entraîne une chute du débit de pro-

[1]Conduite flexible reposant sur le fond marin pour le transport des fluides de production et d'injection

duction d'un puits ce qui engendre indubitablement une perte financière. La simulation d'une telle propriété permet d'évaluer à quel instant il est nécessaire d'injecter des inhibiteurs ou d'arrêter la production d'un puits, ou encore de définir une stratégie permettant de dévier l'écoulement.

Si l'on considère non plus un *pipeline* mais un champ complet de production *offshore* que l'on doit simuler sur une durée de vie de quinze ans et que l'on considère un pas de temps grossier de l'ordre du jour, le temps de simulation d'un tel équipement devient considérable.

Actuellement, une barrière physique à l'augmentation de la fréquence de calcul des processeurs est sur le point d'être atteinte. La solution actuelle afin de développer toujours plus de puissance repose sur la mise en collaboration de plusieurs entités de calcul au sein d'une même machine. Les architectures matérielles multi-cœurs en sont la preuve. De telles architectures nécessitent, cependant, une nouvelle approche de programmation nommée programmation parallèle, qui est complexe à appréhender.

2.5 Bilan

Le domaine de l'ingénierie des procédés assistée par ordinateur tente de répondre à trois grande classes de problèmes : simulation, conception et optimisation. Notons que les travaux présentés dans cette thèse se basent principalement sur la résolution de problèmes de simulation si bien que dans la suite de ce document, nous emploierons le terme simulation de procédé comme étant la résolution d'un problème de simulation.

Selon le type d'étude envisagée, un calcul statique ou dynamique est préférable. Par exemple, si un ingénieur désire connaître l'efficacité d'un procédé (quantifiée en terme de rendement), seul le régime stationnaire est intéressant. *A contrario*, dès que la dimension temps entre en compte, il devient nécessaire d'adopter un mode de simulation dynamique.

Afin de résoudre la totalité de ces problèmes, le domaine définit trois méthodes de résolution. La première, nommée approche modulaire séquentielle, considère la résolution comme l'exécution de plusieurs sous-systèmes représentés par les opérations unitaires. En effet, chaque unité est simulée en suivant les dépendances de flux : la matière est traitée dans une unité puis transmise à une autre en vue de subir une nouvelle transformation. Chaque module utilise son propre modèle de résolution lui permettant d'être hautement spécialisé. Cette méthode expose de très bonnes propriétés de fiabilité et de convergence. Cependant, elle reste limitée à la solution de problèmes de simulation : les autres types de problèmes sont considérés comme l'exécution itérative d'un problème de simulation engendrant alors une charge de calcul trop importante.

La seconde, l'approche orientée équations, considère une simulation de procédés comme la résolution d'un large système d'équations. Les équations de chaque unité sont agglomérées pour former la totalité du procédé. Ce système non linéaire est résolu par la suite en utilisant un solveur global. Cette méthode permet de résoudre les problèmes d'optimisation et de conception assez rapidement.

Enfin, la troisième, l'approche modulaire simultanée, est hybride et cumule les avantages des deux autres. Les équations des flux de recyclage sont résolues simultanément avec celles des contraintes de spécification et d'optimisation. L'approche modulaire séquentielle est utilisée afin de construire la jacobienne ou une approximation pour chaque groupe irréductible. Malgré les

bénéfices d'une telle approche, son utilisation reste restreinte étant donné que l'obtention des dérivées partielles peut être extrêmement coûteuse.

Actuellement, on constate que les industriels utilisent majoritairement l'approche modulaire séquentielle et ce pour des raisons historiques. Tous leurs codes sont développés selon cette approche et sont prouvés fiables et robustes. Ainsi, peu d'industriels trouvent le temps, l'argent et les ressources humaines à investir pour porter leurs codes.

Avec l'émergence des langages à composant issus du génie logiciel, le CAPE a très vite considéré ces techniques comme particulièrement prometteuses pour leur domaine. Ainsi est né le standard CAPE-OPEN promouvant l'intégration de composants tiers au sein d'environnements propriétaires de simulation. Au moyen d'interfaces standardisées, CAPE-OPEN dresse une liste exhaustive des entités métier prenant part à la résolution des problèmes de simulation du génie des procédés. Initialement mitigée, la communauté est maintenant unanime sur les bénéfices du standard et les plus grands industriels fournissent à présent une certaine compatibilité. En effet, malgré la définition exhaustive retenue par le standard, l'engouement n'est quantifiable que pour les composants de type *opération unitaire* et de plus en plus sur ceux de type *serveur de calculs thermodynamiques*. Ceci étant, le contrat CAPE-OPEN est pleinement rempli.

Comme tout problème de simulation numérique, les simulations de procédés requièrent de nombreuses ressources de calcul afin d'offrir plus de précision. Un moyen, désormais incontournable, à l'obtention d'une plus grande capacité de calcul repose sur les architectures parallèles. Cependant, la programmation d'application pour ces machines est complexe. Le chapitre suivant présente les techniques de programmation parallèle.

Programmation sur architectures parallèles 3

3.1 Introduction

Le calcul parallèle doit être capable d'agglomérer la puissance de traitement d'un ensemble de processeurs et la capacité de stockage d'un ensemble de mémoires afin de permettre la résolution de problèmes de plus en plus complexes ou de manière plus performante.

Ce chapitre vise à décrire les différents modèles de programmation à travers une classification regroupant les différentes phases nécessaires à la conception d'une application parallèle. Ensuite, nous décrivons des environnements standards de programmation qui permettent d'exploiter le parallélisme matériel. Pour cela, la section suivante relate les différentes classes d'architectures parallèles sur lesquelles les environnements sont réputés performants. Puis, nous introduisons deux environnements de programmation issus de la recherche académique visant à abstraire totalement l'architecture en proposant au programmeur des interfaces pour la définition du parallélisme applicatif. Ces environnements permettent de programmer des applications parallèles dans un but de performance. Pour conclure, nous définissons les métriques qui qualifient les performances d'un programme parallèle.

3.2 Conception d'algorithmes parallèles et modèle de programmation

Dans cette section, nous présentons rapidement les différentes étapes nécessaires à la conception d'un algorithme parallèle ainsi qu'une classification des modèles de programmation vis-à-vis de leurs capacités à automatiser certaines étapes de conception. Cette classification servira de base à la description des environnements de programmation parallèle présentée dans la section suivante.

3.2.1 Etapes nécessaires à l'exécution efficace d'une application parallèle

Les techniques de programmation parallèle prennent racine dans les paradigmes de programmation concurrente [58, 24]. Ce type de programmation définit l'exécution *simultanée* de plusieurs activités de calcul qui coopèrent et qui se partagent certaines ressources. Elles sont regroupées

pour former des entités d'exécution séquentielle : les processus. Les modèles de programmation concurrente définissent alors comment les processus communiquent et quels sont leurs méthodes de synchronisation.

Les communications permettent aux processus de coopérer grâce à l'échange de données. Cette échange peut alors prendre deux formes majeures : communication à travers une mémoire partagée ou par l'envoi explicite de messages sur le réseau. Les synchronisations permettent de définir des phases d'attente devant lesquelles un processus reste bloqué tant qu'une condition n'est pas vérifiée. Par exemple, une synchronisation bloque l'exécution d'un processus tant que ce dernier ne dispose pas des données dont il a besoin pour effectuer ses traitements.

La programmation concurrente s'attache principalement à garantir que l'exécution répartie d'une application est cohérente en comparaison à son exécution séquentielle. Bien que cet objectif soit indispensable à la programmation parallèle, celle-ci a pour objectif premier les performances : réduire le temps d'exécution d'une application. La réalisation d'un tel objectif nécessite une gestion optimale des techniques de synchronisation et de communication liées à la programmation concurrente mais dépend fortement de critères applicatifs suivants :

1. l'extraction du parallélisme définit comment les instructions élémentaires d'une application sont regroupées afin de former des entités d'exécution plus complexes. Cette phase s'apparente à l'agglomération d'instructions élémentaires en procédures ou fonctions. Dans le cadre de la programmation parallèle, cette étape permet de grossir le grain applicatif en vue de faciliter l'étape de partitionnement.
2. l'ordonnancement des activités ou tâches permet de regrouper les tâches par affinité en vue de déployer les groupes formés sur les entités physiques d'exécution. Ces deux traitements sont définis ci-dessous :
   (a) le partitionnement permet de regrouper les tâches élémentaires selon une *certaine affinité*. Généralement, il s'agit d'apparenter les entités qui appartiennent à une suite séquentielle d'exécution. Cette étape a pour vocation de réduire le nombre de communications et donc les synchronisations nécessaires à l'exécution d'une tâche. Généralement, le nombre résultant de partitions est supérieur au nombre de processeurs disponibles sur l'architecture matérielle cible.
   (b) le placement définit la façon dont les partitions sont déployées sur l'architecture, c'est-à-dire quelles partitions sont affectées à quel processeur. Un objectif de cette étape peut être une répartition équitable de la charge de calcul sur les différents processeurs physiques.

Pour résumer, les aspects liés à la programmation parallèle regroupent ceux liés à la programmation concurrente et ceux propres aux critères applicatifs liés à l'obtention de performance. Sur l'ensemble des aspects nécessaires à l'obtention d'un programme parallèle efficace, les aspects recensés par la programmation parallèle qui possèdent un fort impact sur les performances sont les suivants :

- extraction du parallélisme ;
- ordonnancement :
  - partitionnement ;
  - placement ;

– communication ;
– synchronisation.

Les modèles de programmation parallèle se distinguent par le traitement explicite ou non par un programmeur des différentes étapes ci-dessus.

3.2.2 Classification des modèles de programmation

Skillicorn [104] classifie les modèles de programmation parallèle selon six classes allant du tout implicite au tout explicite en fonction de l'abstraction des traitements nécessaires à la programmation parallèle. La figure 3.1 représente cette classification où les détails d'implantation utilisés sont présentés sur un axe d'abstraction. Plus une classe est à gauche, plus son niveau d'abstraction est élevé. Leur description est proposée ci-dessous :

– Classe 0 : modèles de programmation parallèle où tous les traitements sont implicites. Ces modèles décrivent uniquement l'objectif d'une application et non pas de quelle manière il est possible de l'atteindre. Un développeur ne se préoccupe pas de savoir si l'application sera exécutée en parallèle.
– Classe 1 : modèles de programmation parallèle où seule l'extraction du parallélisme est explicite. Un programmeur doit définir où se trouve le parallélisme de son application.
– Classe 2 : modèles de programmation parallèle où seuls la découverte et le partitionnement sont explicites. Le programmeur doit clairement concevoir son application en la décomposant en différents flots d'exécution.
– Classe 3 : modèles de programmation parallèle où extraction, partitionnement et placement sont explicites. Le développeur doit décider de la meilleure affectation de chaque flot d'exécution sur les processeurs de l'architecture matérielle.
– Classe 4 : modèles de programmation parallèle où seul l'aspect synchronisation est implicite : l'échange des données entre processus est explicite.
– Classe 5 : modèles de programmation parallèle totalement explicite.

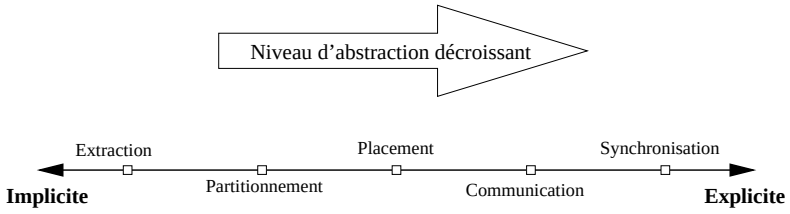


FIG. 3.1 – Classement des modèles de programmation selon la gestion des aspects liés à la programmation parallèle.

Les modèles de classe 0, où les aspects liés au parallélisme sont implicites, exposent la plus faible complexité de programmation. En effet, il suffit de programmer son application en utilisant les mêmes paradigmes liés à la programmation séquentielle. La totalité de la charge de travail est laissée aux compilateurs/environnements d'exécution. Malheureusement, la conversion auto-

matique d'un programme séquentiel en un programme parallèle est compliquée étant donné que le compilateur/environnement doit extraire le parallélisme et ceci en analysant la totalité des dépendances entre les instructions élémentaires de l'application. Un autre argument en défaveur à cette classe est qu'un algorithme parallèle n'est généralement pas l'algorithme séquentiel qui serait exécuté en parallèle. Il est nécessaire de remplacer les schémas séquentiels de calcul par des schémas parallèle. Cette approche, bien que largement étudiée dans les années 90, est actuellement restreinte à la parallélisation des nids de boucles [20]. Aussi, nous pensons que pour être efficace, une application doit être pensée et conçue comme parallèle dès le début. C'est pourquoi, par la suite, nous ne considérerons plus la décomposition comme un aspect pouvant être implicite à un modèle de programmation parallèle.

Soulignons une propriété largement admise dans la communauté du calcul parallèle : plus on abstrait les aspects de gestion du parallélisme, plus la généricité engendrée éloigne l'application des performances optimales sur une architecture cible. En effet, aussi sophistiqué que peut être le mécanisme d'abstraction, celui-ci ne peut difficilement remplacer les connaissances d'un développeur sur un problème particulier pour une architecture cible. Ainsi, pour un type d'application, il ne peut être envisageable de souscrire à un modèle trop « *simpliste* » et c'est pourquoi la majorité des applications parallèles sont actuellement développées suivant un modèle de programmation parallèle tout explicite.

Cette taxinomie repose sur l'inclusion des classes. Or, certains environnements de programmation parallèle, tels que Cilk [15] par exemple, abstraient les aspects partitionnement, placement et communication alors que les aspects extraction et synchronisation sont explicites. De nombreux travaux ont montré que les aspects partitionnement/placement sont les points essentiels qui permettent l'obtention des performances. C'est pourquoi nous adoptons, dans la suite de ce manuscrit, la classification simplifiée suivante :

- **Modèle de programmation parallèle bas niveau** : la gestion de la plupart des aspects parallèles sont à la charge du programmeur. Ainsi, le développeur de l'application doit prendre en considération la majorité des étapes liées à la programmation parallèle. En particulier, l'aspect regroupement doit être explicitement traité par le programmeur. Ces modèles offrent des mécanismes permettant d'exploiter le parallélisme effectif de l'architecture. Nous présenterons quelques outils implantant ce modèle dans la section 3.3.
- **Modèle de programmation parallèle haut niveau** : ces modèles abstraient les aspects liés au partitionnement/placement. Seules les caractéristiques extraction et synchronisation ou communication sont explicites. Selon ces modèles, un programmeur se concentre sur le parallélisme applicatif du programme qu'il développe. Leur présentation fera l'objet de la section 3.4.

Avant d'aborder la présentation des outils permettant la programmation du parallélisme matériel, la section suivante expose une taxinomie des architectures parallèles couramment utilisées.

3.2.3 Description des architectures parallèles

Une machine parallèle est caractérisée par la mise en coopération d'un ensemble de ressources de calcul permettant la résolution d'un même problème. Bien que cette définition soit

générique, il existe de nombreuses manières de concevoir une telle architecture. De nombreux travaux [34, 39, 109, 65] tentent de classifier les machines parallèles selon un ensemble de critères. Cependant, aucun consensus n'a vu le jour actuellement. Dans cette section, nous utilisons un sous-ensemble de la taxinomie de Flynn [39].

Flynn caractérise les architectures parallèles par le nombre de flots d'instructions et le nombre de flots de données qu'elles peuvent gérer. On distingue alors deux grandes classes d'architectures parallèles : les SIMD (*Single Instruction Multiple Data*) et les MIMD (*Multiple Instruction Multiple Data*).

- Les machines parallèles appartenant à la classe SIMD exécutent les mêmes instructions sur un ensemble de données différentes. Les machines vectorielles sont l'un des représentants de cette classe, de même que les anciennes machines Maspar ou ThinkingMachine CM1 et CM2. Bien que simple à programmer, elles ne permettent généralement que d'exploiter un parallélisme de données restreignant ainsi les classes de problèmes pouvant être traitées. De plus, de par leur conception, un fonctionnement synchrone de l'ensemble des unités matérielles de calcul est nécessaire. Ce fonctionnement est d'autant plus coûteux à réaliser que le nombre de processeurs est important. Ces architectures n'ont plus vocation à être des machines parallèles universelles offrant une grande puissance de calcul. Leur niche concerne désormais la conception de processeurs spécialisés tels que ceux utilisés actuellement dans les cartes graphiques.
- Dans une architecture MIMD, chaque processeur peut exécuter un flot d'exécution propre sur un ensemble de données privées. Ces architectures sont d'un usage général à la résolution de tout type de problème. Toutefois, cette généricité a un prix : l'effort nécessaire pour la programmation performante d'applications sur ce type d'architecture en est accru. De nos jours, ces architectures se retrouvent dans les machines les plus puissantes disponibles [110]. Cette classe d'architecture MIMD est sous-divisée en deux ensembles selon le mode d'adressage de la mémoire. Ainsi, on parle de **multi-processeurs** et de **multi-ordinateurs**. Les multi-processeurs, également nommés **architectures à mémoire partagée**, disposent d'un espace d'adressage global partagé par tous les processeurs. *A contrario*, les multi-ordinateurs, également appelés **architectures à mémoire distribuée**, possèdent une mémoire privée attachée à chaque processeur. Dans ce cas, l'ensemble des processeurs communiquent par un réseau d'interconnexion. Notons qu'il devient de plus en plus courant de rencontrer des machines hybrides : des multi-ordinateurs de multi-processeurs. Ceci est d'autant plus vrai avec la démocratisation des processeurs multi-cœurs.

3.2.3.1 Machines à mémoire partagée

Les architectures multi-processeurs ont la possibilité de disposer d'un espace d'adressage global de la mémoire, c'est-à-dire, tous les processeurs partagent les mêmes plages d'adresses. Ainsi, si un processeur écrit la valeur 38 à l'adresse physique $0xb7edec01$, tout autre processeur accédant à cette adresse lira par la suite cette même valeur. Bien que le système d'adressage soit global, les bancs mémoires peuvent être répartis sur différentes unités physiques.

La hiérarchie physique de la mémoire détermine ses temps d'accès. Dans le cas où chaque processeur met le même temps pour accéder à la mémoire centrale, l'architecture est appelée UMA

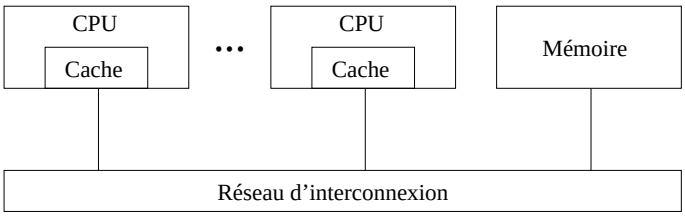


FIG. 3.2 – Architecture parallèle à mémoire partagée de type CC-UMA

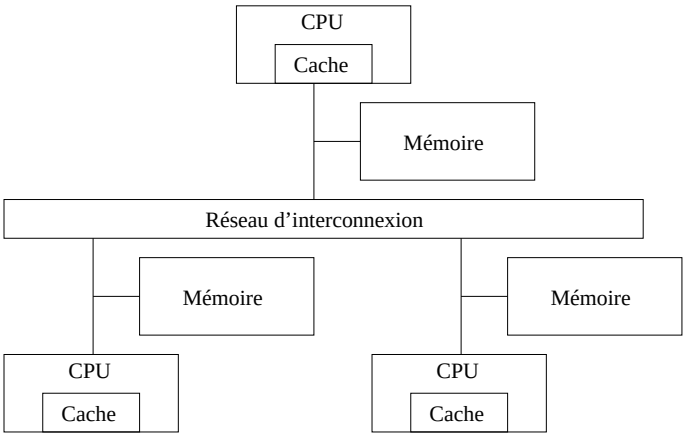


FIG. 3.3 – Architecture parallèle à mémoire partagée de type CC-NUMA

(pour *Uniform Memory Access*). Dans le cas contraire, elles sont appelées NUMA (*Non Uniform Memory Access*) : un processeur accède plus rapidement à sa mémoire locale qu'à celle d'un autre. Quelque soit ce temps d'accès, il est encore considéré comme trop coûteux par rapport à la rapidité de traitement des processeurs. Ainsi, les processeurs utilisent généralement un ou plusieurs niveaux de mémoire cache limitant l'attente pour accéder aux données référencées. Du fait des multiples copies potentielles dans les différents caches des processeurs, en cas de modification de l'une d'entre elle, un protocole permet de garantir leur mise en cohérence [55]. Ces architectures sont nommées CC-UMA (figure 3.2) ou CC-NUMA (figure 3.3) (pour *Cache Coherent*-UMA ou NUMA).

Bien que moins coûteuses à concevoir, les architectures CC-UMA sont de moins en moins produites actuellement (à l'exception des architectures multi-cœurs). En effet, l'organisation centralisée de la mémoire réduit les performances en terme de temps d'accès : les processeurs partagent le même réseau d'interconnexion, ce qui divise d'autant la bande passante allouée à chaque processeur. Ainsi, mêmes les architectures bi-processeurs évoluent sur des hiérarchies mémoires de type CC-NUMA (par exemple, les bi-processeurs Opteron sont de ce type).

Bien que l'on rencontre des machines composées de plusieurs centaines de processeurs, les algorithmes de mise en cohérence engendrent d'importantes perturbations pour des applications

de calculs réelles. Ainsi, il est courant de concevoir des architectures ne dépassant pas la centaine de processeurs. Ainsi, pour disposer d'un plus grand nombre de processeurs, les architectures à mémoire distribuée se sont imposées.

3.2.3.2 Machines à mémoire distribuée

Selon cette classe de machines, à l'instar des machines NUMA, chaque processeur possède sa propre mémoire. Tous les processeurs sont connectés à travers un réseau de communication. Cependant, l'espace d'adressage de la mémoire est propre à chaque processeur : un processeur A ne peut pas directement adresser la mémoire attachée à un processeur B. La modification par l'un des processeurs de sa propre mémoire ne provoque pas d'influence directe (d'un point de vue matériel) sur celle des autres processeurs. La mémoire est distribuée entre les processeurs sans aucune mise en cohérence des données. Afin qu'elles soient partagées entre les processeurs une communication explicite doit être initiée, soit par le processeur effectuant les modifications, soit par le processeur voulant accéder aux valeurs modifiées. La figure 3.4 illustre ce modèle d'architecture parallèle.

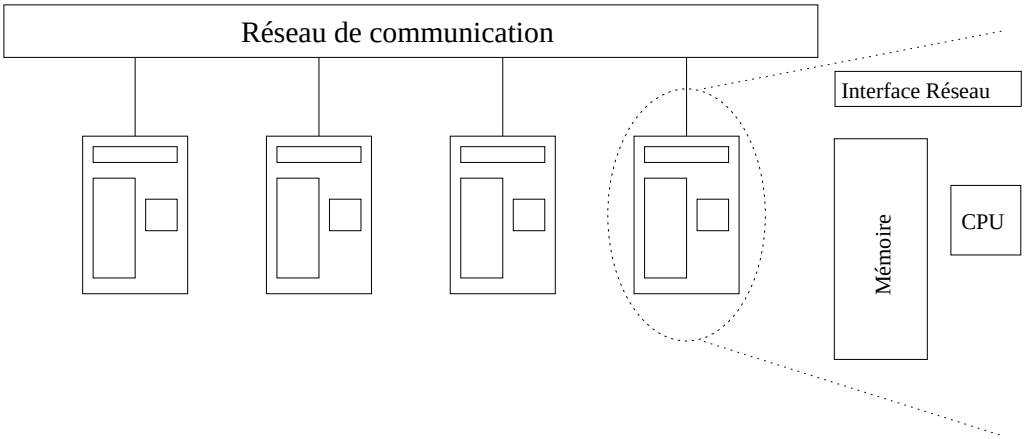


FIG. 3.4 – Architecture parallèle à mémoire distribuée. Chaque nœud est composé d'un processeur, d'une mémoire privée ainsi que d'une interface réseau permettant la communication des données *via* un réseau de communication. Ces communications doivent être explicites (gestion logiciel).

Cette classe d'architectures est intrinsèquement asynchrone : chaque processeur possède une horloge propre et évolue indépendamment des autres. Les communications jouent alors le rôle de point de synchronisation permettant un échange d'information.

De telles architectures peuvent atteindre une taille plus importante en terme de nombre de processeurs et de capacité mémoire que les architectures à mémoires partagées. Certaines machines dans la liste du top500 [110] peuvent atteindre plusieurs milliers de processeurs. Cependant, l'accès à la mémoire d'un processeur distant à travers le réseau de communication reste d'un à deux ordres de grandeur plus important que dans le cas des multi-processeurs et ceci malgré l'utilisation de réseaux haute performance tels que Myrinet [18], SCI [57], Quadrics [89] ou encore Infiniband [4]. Ces réseaux de communication, basés sur de la fibre optique, permettent d'atteindre

de très haut débit (de l'ordre de quelques Gigabits/s) avec de faibles latences (quelques microsecondes). Leurs performances dépendent fortement du protocole de communication : chacun de ces réseaux définit un protocole propre de communication en parallèle du protocole standard TCP.

Leur moindre coût à l'achat (comparé au multi-processeurs) en font des architectures prisées. Centres de recherche et entreprises voient dans ces architectures le moyen d'accéder aisément à une puissance de calcul importante. De plus, le fait qu'elles soient plus facilement modulaires et évolutives les rendent des plus attractives.

La majorité des grappes de PC[1] était représentée par ce type d'architecture, tel que le *Icluster-1* démantelé en 2004. A présent, les architectures installées ont évolué et fusionnent à différents niveaux ces deux types d'organisation de la mémoire (multi-processeurs et multi-ordinateurs).

3.2.3.3 Machines hybrides

L'émergence des processeurs double et même quatre cœurs dans l'informatique tout public se reporte nécessairement sur les ordinateurs composant les grappes de PC. Cette avancée technologique fera évoluer les architectures à mémoire distribuée vers des architectures hybrides où cohabiteront mémoire partagée et mémoire distribuée (Dans Grid5000 [52], le cluster Azur de Sophia ou Paraci de Rennes sont des bi-processeurs double cœurs).

Cependant, les grappes de PC n'ont pas attendu l'arrivée des processeurs multi-cœurs pour allier mémoire partagée et mémoire distribuée. La majorité des grappes composant Grid5000 comporte des nœuds bi-processeurs : on parle alors d'architectures de type CLUMPS (*CLUster of MultiProcessorS*). Chaque bi-processeur forme un nœud de l'architecture parallèle. Les nœuds sont ensuite interconnectés au moyen d'un réseau haute performance.

Le rapport performance prix de telles architectures les rendent attractives. Portées initialement par la recherche académique, ces architectures se sont petit à petit implantées dans les centres de recherche industriels.

Ces architectures définissent alors un certain type de parallélisme axé principalement sur la définition des communications. Les outils de programmation offrent des interfaces de programmation permettant d'exploiter ce parallélisme matériel.

3.3 Outils pour l'exploitation du parallélisme matériel

Dans cette section, nous présentons les outils et environnement de programmation de bas niveau permettant d'exploiter explicitement le parallélisme matériel de l'architecture cible.

Afin d'apprécier les différences d'abstraction, la description de chacun des outils présentés sera accompagnée du pseudo-code permettant de résoudre en parallèle l'intégrale d'une fonction f linéaire :

$$\int_a^b f(x)dx = \sum_{i=1}^{n-1} F_i \text{ où } F_i \simeq \int_{a_i}^{a_{i+1}} f(x)dx$$

F_i est calculée par une méthode telle que celle des trapèzes par exemple, n est le pas de discrétisation.

[1] *Cluster of Workstations* (COW), *Network of Workstations* (NOW), *commodity cluster* et cluster beowulf sont synonymes dans notre étude.

3.3.1 Processus légers

Ces environnements proposent des interfaces de programmation pour une architecture à mémoire partagée où les flots d'exécution communiquent au moyen de variables partagées situées dans une mémoire commune. Ces flots d'exécution sont généralement nommés processus légers (ou *threads*) marquant la distinction avec les processus lourds de type UNIX. Un processus lourd peut créer un ou plusieurs processus légers. Les différents processus légers au sein d'un processus lourd partagent son contexte mémoire permettant ainsi une gestion beaucoup moins coûteuse, par exemple sur les opérations de création ou de changement de contexte d'exécution.

La norme POSIX [61, 81] définit un standard d'interfaces pour la manipulation des processus légers. Les interfaces de programmation offertes par cette norme sont restreintes à l'exploitation de machines à mémoire partagée. *TreadMarks* [2] propose un modèle proche adapté à la programmation d'architectures à mémoire distribuée en développant un support logiciel pour gérer une mémoire partagée au dessus de la mémoire distribuée [70].

Selon ce modèle, les différents *threads* d'une application parallèle définissent des flots d'exécution concurrents. Du fait que les données soient partagées et ne nécessitent pas de communications explicites, ces environnements proposent principalement des synchronisations permettant, d'une part, de garantir la consistance de ladite mémoire et, d'autre part, de définir le contrôle indispensable à la coordination des flots d'exécution pour la résolution du problème.

La première technique de synchronisation, l'exclusion mutuelle, permet de qualifier une région critique au moyen de la mise en place d'un verrou limitant à un le nombre de *threads* exécutant cette région. Un *thread* demande l'acquisition d'un verrou et en retour d'appel, il détient le verrou. Le temps passé par le *thread* dans l'appel dépend alors de la disponibilité du verrou. La bibliothèque ne le lui concède que si ce dernier n'est pas déjà alloué à un autre flot d'exécution. Si tel est le cas, le *thread* est bloqué tant que le verrou n'a pas été relâché. Cette technique représente l'une des plus grandes sources d'erreurs en introduisant des situations d'inter-blocage, où un *thread* A attend un *thread* B en même temps que B attend A.

La seconde technique de synchronisation permet de bloquer un flot d'exécution sur la validation d'une condition. Si la condition est valide, le *thread* continue son exécution sans phase d'attente, si elle n'est pas vérifiée, un ou plusieurs autres *threads* déclenchent un événement permettant de lever cette condition.

La figure 3.5 présente une implantation de la résolution de l'intégrale d'une fonction linéaire selon ce modèle de programmation.

La création de parallélisme est définie explicitement par le programmeur au moyen de la création d'un processus léger. Chaque processus léger est ordonnancé sur les ressources physiques de calcul en suivant une politique d'ordonnancement du système d'exploitation et de l'implantation de la bibliothèque de *threads*. Par exemple, on trouve classiquement des ordonnancements basés sur un partage des ressources par quantum de temps : chaque *thread* occupe une ressource de calcul durant un certain temps puis le système ordonnance le prochain *thread* prêt, cette opération s'appelle la préemption. Bien que le nombre de *threads* soit indépendant du nombre de ressources physiques de calcul, il est à la charge du programmeur d'ajuster le nombre de processus légers à

Début du code

```
//Variable en mémoire partagée
mutex    : //mutex conditionnant l'accès à la variable résultat
résultat : //intégrale de la fonction f(x)

void Calc_Intégrale (début, fin){
  //Paramètres en entrée:
  //début : début de l'intervalle sur lequel calculer l'intégrale partielle
  //fin   : fin de l'intervalle sur lequel calculer l'intégrale partielle

  //Résultat partiel de l'intégration sur l'intervalle [début, fin]
  res_partiel = Intégrale(début, fin);

  //le thread courant accède en exclusion mutuelle à la donnée en accumulation
  pthread_mutex_lock(mutex);
  résultat += res_partiel;
  pthread_mutex_unlock(mutex);
}

//Programme principal
void main(p, a, b){
  //Paramètres en entrée:
  //p    : nombre de processeurs
  //a, b : intervalle

  //Calcul du pas de discrétisation n en fonction du nombre de processeurs
  n = (b-a)/p;
  //Tableau des identifiants de chaque thread à créer
  pthread_t thid[p];
  //Initialisation de la valeur résultat
  résultat = 0;

  //Création de p threads de calcul réalisant la méthode Calc_Intégrale
  for(i=0; i<p-1; ++i){
    pthread_create(thid[i], Calc_Intégrale, (a+i*n, a+(i+1)*n);
  }

  //barrière de synchronisation sur la terminaison de tous les threads
  for(i=0; i<p-1; ++i){
    pthread_join(thid[i]);
  }
  //Toutes les données ont été produites
  Afficher(résultat);
}
```

Fin du code

FIG. 3.5 – Pseudo code du calcul de l'intégrale selon le modèle de programmation multi-programmation légère.

la capacité de l'architecture utilisée et ceci en vue de limiter le temps passé dans la préemption.

Souvent considérées comme plus simples de programmation car masquant la définition des communications, les bibliothèques de multi-programmation légère restent majoritairement limitées au développement d'applications pour des architectures à mémoire partagée. En effet, *Tread-Marks* souffre d'une certaine inefficacité par rapport aux bibliothèques d'échange de messages [118].

3.3.2 Echange de messages : PVM/MPI

PVM (*Parallel Virtual Machine*) [49] et MPI (*Message Passing Interface*) [40, 79] sont deux standards *de facto* largement utilisés dans le développement d'applications parallèles. Historiquement, ils ont été développés dans le but de simplifier l'utilisation des interfaces propriétaires des premières machines parallèles en cherchant à rendre standard l'accès aux fonctionnalités existantes d'envoi et de réception de messages. Actuellement, MPI est considéré comme le standard permettant d'exploiter efficacement des réseaux aussi différents que TCP/IP, Myrinet, Quadric ou encore SCI que l'on retrouve dans de nombreuses architectures parallèles.

Alors que les environnements de type *thread* basent principalement leur modèle de programmation sur la description des synchronisations, les environnements de type envoi de messages reposent sur la description des communications entre les processus coopérants. Elle repose principalement sur la sérialisation/désérialisation des données ainsi que l'identification des processus émetteur et récepteur. Pour cela, de tels environnements proposent deux opérateurs définissant l'échange de message : l'envoi et la réception. Notons qu'ils impliquent une synchronisation entre processus : l'envoi et la réception, dans leur sens strict, définissent des opérations synchrones. Le mode de communication est de type rendez-vous, où les deux processus communicants s'accordent afin d'émettre les données.

Les opérations de réception et d'émission impliquent un blocage des processus jusqu'à ce que les données aient bien été reçues ou envoyées. Il existe plusieurs variantes à ces primitives de communication : les envois et réceptions non-bloquantes où les processus impliqués initie la communication sans s'attarder sur sa réalisation. Ils peuvent alors continuer leur calcul en concurrence avec la communication. Cependant, il devient nécessaire à un moment ou un autre de tester si la communication est terminée ou non. Ce test gère essentiellement le problème de la réutilisation d'une zone mémoire de niveau applicatif sur laquelle une communication opère.

Les environnements PVM/MPI offrent des opérations de communications collectives, fort nombreuses, qui permettent, par exemple, d'envoyer une donnée à tous les processus, ou bien de fusionner une donnée en provenance de tous les processus sur un processus. Ces opérations utilisent des algorithmes parallèles qui permettent d'initier des communications en parallèle entre processus différents.

Comme précédemment, la figure 3.6 propose une implantation de la résolution de l'intégrale d'une fonction linéaire suivant un tel modèle de programmation.

Avec ces environnements, le programmeur doit gérer la décomposition de son problème en processus possédant une partie des données du problème. Ensuite, l'échange des données dicte quels sont les envois et réceptions de messages nécessaires à la réalisation de son algorithme parallèle. L'exploitation des capacités des architectures à mémoire distribuée est des plus aisée étant donné que ces environnements décrivent explicitement la gestion du parallélisme effectif. Cependant, la programmation d'une application est complexe car le développeur doit concevoir son application en traitant tout les aspects liés au parallélisme de l'architecture. De plus, de par son modèle de conception, ces environnements se basent principalement sur l'identification des communications nécessaires à l'avancement de l'exécution des processus répartis, ainsi l'exploitation fine des capacités des architectures parallèles hybrides est difficile en utilisant directement PVM/MPI : ces architectures étant composées de nœuds multi-processeurs, le parallélisme intra-

Début du code

```
//On considère que seul le processus maître (rang 0) peut calculer le pas de discrétisation

void Calculer_Intégrale(mon_rang, nb_proc, a, b {
  //Paramètres en entrée:
  //mon_rang : rang du processeur
  //nb_proc  : nombre de processeur
  //a, b : intervalle

  //Calcul du pas de discrétisation n en fonction du nombre de processeurs
  n = (b-a)/p;

  if(mon_rang == 0){
    for(i=1; i<nb_proc-1; ++i){
      //Envoi à chaque processus (sauf 0) de l'intervalle à calculer
      MPI_Send(a+i*n, a+(i+1)*n);
    }
    //le processus 0 garde le travail [a,a+n]
    ai = a;
    bi = a+n;
  }
  else{
    //réception des intervalles à calculer
    MPI_Recv(ai, bi);
  }

  //Tous les processus réalisent le calcul de l'intégrale sur l'intervalle [ai, bi]
  res_partiel = Intégrale(ai, bi);

  //Communication collective: réduction de tous les résultats partiels
  //sur le processus 0 : la fonction de réduction est l'accumulation
  MPI_Reduce(0, résultat, res_partiel, MPI_SUM);
  if(mon_rang == 0)
    Afficher(résultat);
}
```

Fin du code

FIG. 3.6 – Pseudo code du calcul de l'intégrale d'une fonction avec le modèle de programmation par envoi de messages.

nœuds passe par l'utilisation d'une bibliothèque de processus légers comme vu précédemment. L'exploitation de ces architectures hybrides nécessite un modèle de programmation hybride.

3.3.3 Appel de procédure à distance

L'appel de procédure à distance fut développé en vue de simplifier le paradigme par envoi de messages. Il permet de s'affranchir du côté basique des communications en mode message *via* l'utilisation d'une structure de programmation familière (l'appel de procédure). Ainsi, la sérialisation des données est implicite au programmeur qui définit simplement une invocation de procédure avec ses paramètres effectifs.

Début du code

```
//Contrat IDL Serveur
interface Serveur{
   oneway void async_Calc_Intégrale(in int a, in int b);
}

//Pseudo-code serveur
//Attribut:
client : référence sur le composant client

void async_Calc_Intégrale(a, b){
   //Paramètres en entrée:
   //a, b : intervalle

   res_partiel = Intégrale(a, b);
   Client.Retourne_Résultat(res_partiel);
}

//Contrat IDL Client
interface Client{
   oneway void Retourne_Résultat(in float res_partiel);
}

//Pseudo-code client
//Attribut:
mutex    : mutex conditionnant l'accès à la variable résultat
résultat : intégrale de la fonction f(x)
serveurs : références sur les composants serveur, taille p

void main(p, a, b){
   //Paramètres en entrée:
   //p    : nombre de processeurs
   //a, b : intervalle

   //Calcul du pas de discrétisation n en fonction du nombre de composants serveur
   n = (b-a)/p;
   résultat=0;
   for(i=0; i<p; ++i){
      //Appel de la méthode asynchrone
      serveurs[i].async_Calc_Intégrale(a+i*n, a+(i+1)*n);
   }
   //Synchronisation marquant la fin de l'exécution de la méthode
   //sur chaque serveur serveur[i] au moyen d'une sémaphore ou attente conditionnelle
   Attendre_Tous();
   Afficher(résultat);
}

void Retourne_Résultat(res_partiel){
   //Paramètres en entrée:
   //res_partiel : résultat partiel

   //Prise de verrou pour l'accumulation dans la variable résultat
   pthread_mutex_lock(mutex);
   résultat+=res_partiel;
   pthread_mutex_unlock(mutex);
   //Signalisation du callback à destination du thread main
   Signal_Fin();
}
```

Fin du code

FIG. 3.7 – Pseudo code du calcul de l'intégrale avec le modèle de programmation RPC en considérant des appels de méthodes asynchrones.

RPC (Remote Procedure Call) [107] (1976) est la première norme ayant formalisée une interface de programmation sur l'appel de procédure à distance. Une invocation engendre une communication synchrone, c'est-à-dire que le client (l'appelant) reste bloqué tant que le serveur (l'appelé) n'a pas fini l'exécution de la procédure.

CORBA [82] et DCOM [21] offrent une variante du paradigme RPC basée sur la notion d'objets ou composants distribués. Ils offrent une approche RPC intégrée dans les systèmes d'objets répartis. Alors que CORBA est une norme définie par l'OMG (*Object Management Group*) qui est un consortium ouvert, DCOM est la technologie propriétaire de Microsoft.

L'idée clé régissant ces systèmes repose sur la notion d'interfaces permettant de définir les services qu'un composant met à disposition du monde extérieur. C'est à partir de celles-ci qu'une application (*i.e.*, le client) peut invoquer des méthodes sur des objets (*i.e.*, les serveurs). Afin de recenser l'ensemble des opérations offertes par un composant sans pour autant disposer du code, l'environnement de programmation offre un langage de définition d'interfaces IDL (*Interface Description Language*) permettant de décrire, d'une manière standardisée et indépendante de tout langage de programmation, les services offerts ainsi que leurs attributs en entrée et en sortie nécessaires à leur exécution. Le langage IDL permet d'exprimer, sous la forme d'un contrat, la coopération entre les fournisseurs et les utilisateurs de services, en séparant l'interface de l'implantation des objets et en masquant les divers problèmes liés à l'interopérabilité, l'hétérogénéité et la localisation de ceux-ci [48].

Ce langage IDL introduit également la *direction* des communications. Il identifie alors au moyen d'un qualificateur spécifique les données qui sont nécessaires en entrée (*in*), celles qui sont produites (*out*) ou celle qui sont modifiées (*inout*). Cette qualification permet de communiquer uniquement les données nécessaires entre le client et le serveur pour la réalisation de l'invocation.

Généralement, l'invocation d'une méthode sur un composant génère la création d'un *thread* en charge de réaliser le traitement associé. Ainsi, plusieurs méthodes sur un composant peuvent être exécutées en concurrence. Il devient alors indispensable de considérer les aspects d'exclusion mutuelle sur les attributs de l'objet.

Ce paradigme fut utilisé avec succès pour exécuter des applications réparties ou la performance n'était pas le critère majeur. Généralement, ces environnements sont utilisés selon un paradigme de programmation de type client-serveur où le client dicte l'exécution en déléguant le traitement de certaines fonctionnalités à des serveurs. L'exécution est alors centralisée autour du client. Ce caractère prend alors deux visages : l'un lié à la sémantique de contrôle et l'autre lié à la sémantique de communication des données.

– La sémantique de contrôle définit les synchronisations entre les composants et principalement entre le client et les serveurs. Une invocation de méthode est une instruction bloquante où le client attend que le serveur ait fini le traitement associé à la méthode. Selon cette propriété, on se rend compte des limitations d'une telle approche dans un environnement d'exécution parallèle : il ne développe que peu (voir pas) de parallélisme de par les synchronisations fortes existantes entre les processus appelant et appelé. Ainsi, afin de dégager un tant soit peu de parallélisme et donc produire des flots d'exécution concurrents (entre le client et les divers serveurs), un programmeur doit coupler invocations synchrones et multiprogrammation légère. Cette solution n'est guère performante car elle requiert un *thread* client pour chaque invocation en attente de résultats. Cependant, cette sémantique n'est

pas rédhibitoire à l'obtention de performance. En effet, CORBA et DCOM implantent des techniques d'invocations de méthodes asynchrones où le flot de contrôle client redevient actif dès que les données à envoyer se trouvent dans le tampon d'émission. Néanmoins, ce parallélisme a un prix : les synchronisations sur le processus client doivent être traitées explicitement.

– La sémantique de communication est un facteur bien plus limitatif. En effet, la sémantique de passage de paramètres couple les flots de contrôle et de données. Les paramètres de type *in* sont transmis du client au serveur, en contrepartie, le serveur fournit les paramètres *out* au client. Bien que définissant une optimisation locale (un seul message est nécessaire au contrôle et aux données), elle ne définit pas une optimisation globale où un client invoque plusieurs méthodes sur un ensemble de serveurs. Tous les paramètres de retour sont ainsi communiqués au client et ceci même si ce dernier ne fait que retransmettre ces paramètres vers un autre serveur : non seulement le flot de contrôle est centralisé sur le client, mais la communication l'est également.

La figure 3.7 présente le pseudo-code du calcul de l'intégrale d'une fonction linéaire selon ce modèle en utilisant une technique d'invocations de méthodes asynchrones.

Alors que les environnements de multi-programmation légère et d'échange de message offrent des interfaces permettant d'exploiter au plus près les propriétés des architectures matérielles, les environnements orientés RPC offrent une API de plus haut niveau couplant mémoires partagée et distribuée. Toutefois, ces trois interfaces de programmation n'automatisent pas les problématiques de partitionnement et de placement. Les environnements de programmation de haut niveau répondent à ce besoin d'abstraction.

3.4 Environnements pour l'exploitation du parallélisme applicatif

Dans la section précédente, nous avons présenté les outils pour l'exploitation du parallélisme effectif d'une architecture matérielle parallèle. A présent, nous introduisons deux environnements de programmation parallèle de haut niveau offrant une abstraction complète du parallélisme matériel. Selon le modèle de programmation offert, un développeur d'application ne doit plus traiter des problèmes liés à l'exploitation du matériel, ses efforts doivent être orientés sur l'algorithmique parallèle et sur la détection du parallélisme de l'application considérée.

A l'instar des outils de bas niveau, nous agrémentons leur description en réutilisant l'exemple du calcul d'intégrale d'une fonction linéaire utilisé tout au long de la section précédente.

3.4.1 Cilk

Cilk[2] [15] est un environnement de programmation de haut niveau visant les architectures à mémoire partagée. Il se présente comme une extension au langage C offrant certaines primitives pour exprimer un parallélisme de type fonctionnel. Ce parallélisme et les synchronisations nécessaires à la cohérence de l'exécution sont décrits explicitement par le programmeur.

[2]Cilk est actuellement dans sa version 5. Le manuel de référence [108] présente un historique des versions.

Début du code

```
float Calc_Intégrale(a, b, Seuil){
  //Paramètres en entrée:
  //a, b  : intervalle
  //Seuil : pas de discrétisation minimal
  //Paramètre en sortie:
  //res_partiel : résultat partiel de l'intégrale sur l'intervalle [a, b]

  if(b-a <= Seuil){
    res_partiel = Intégrale(a, b);
  }
  else{
    //Calcul du pas de discrétisation n
    n = (b-a)/2;
    res_partiel1 = spawn Calc_Intégrale(a, n);
    res_partiel2 = spawn Calc_Intégrale(n, b);
    sync;
    res_partiel = res_partiel1 + res_partiel2;
  }
  return res_partiel;
}

//Programme principal
void main(a, b, Seuil){
  //Paramètres en entrée:
  //a, b  : intervalle
  //Seuil : pas de discrétisation fixé par l'utilisateur

  résultat = spawn Calc_Intégrale(a, b, Seuil);
  sync;
  Afficher(résultat); //affiche le vecteur
}
```

Fin du code

FIG. 3.8 – Pseudo code du calcul de l'intégrale d'une fonction linéaire avec le modèle de programmation Cilk.

L'idée clé développée par Cilk est d'offrir à tout programmeur un environnement de programmation simple lui permettant de définir le parallélisme selon deux mot-clés : `spawn` et `sync`. Le programmeur définit explicitement le parallélisme d'une application en préfixant les méthodes par le mot-clé `spawn`. Cependant, cette syntaxe ne permet pas de détecter les dépendances de données entre les diverses tâches concurrentes. Ainsi, Cilk est adapté à un parallélisme de type série-parallèle où une tâche mère génère un ensemble de tâches filles totalement concurrentes les unes des autres. Les seules synchronisations autorisées mettent en jeu une tâche mère et ses filles. Le modèle d'exécution induit par un tel schéma définit trois étapes : division, exécution et agglomération. La division est gérée par le mot-clé `spawn` et la phase d'agglomération par `sync`. Le modèle mémoire suppose un espace d'adressage global accessible par toutes les tâches où les données sont communiquées par effet de bord. Une synchronisation définit une barrière globale à une tâche mère spécifiant que toutes les tâches filles se sont terminées.

Cilk offre une technique d'extraction du parallélisme à l'exécution et par conséquence, il en va de même pour le regroupement et le placement des tâches. Toute méthode précédée du mot-clé

`spawn` se trouvant dans la portée du flot de contrôle courant et précédant le mot-clé `sync` est définie comme prête. Ainsi, l'exécution d'une tâche mère est concurrente avec l'exécution de ses tâches filles jusqu'à la rencontre du mot-clé `sync`.

La figure 3.8 présente une implantation du calcul de l'intégrale (calcul présenté en début de section précédente) avec le modèle de programmation proposé dans Cilk.

Bien que Cilk présuppose une architecture à mémoire partagée, une extension à une ancienne version de Cilk (Cilk-2), nommée Cilk-NOW [17], fut développée afin de traiter les architectures hybrides (*e.g.* CLUMPS).

3.4.2 Athapascan

L'environnement de programmation parallèle Athapascan/KAAPI [45, 42] est développé dans le cadre du projet MOAIS au sein du laboratoire ID-IMAG (Informatique et Distribution) maintenant dénommé LIG.

Athapascan [44] est l'interface applicative (API) au dessus de KAAPI (présenté dans le chapitre 9) qui offre un modèle de programmation parallèle de haut niveau où la description du parallélisme et l'identification des données à communiquer sont explicites. A l'instar de Cilk, deux mots-clés permettent de décrire une application parallèle selon un parallélisme fonctionnel : `Fork` et `Shared`.

Le mot-clé `Shared` définit alors une variable appartenant à une mémoire visible par toutes les tâches quelque soit leur processus d'exécution. Cette mémoire est appelée *mémoire globale*.

Le mot-clé `Fork` s'apparente au `spawn` de Cilk dans le sens où il permet de définir une tâche pouvant *potentiellement* être exécutée en parallèle. Il diffère de Cilk par la gestion des synchronisations entre les tâches. En effet, les synchronisations selon Athapascan sont implicites et déduites des accès des tâches aux variables de la mémoire globale. Le modèle de programmation défini par Athapascan repose sur celui exposé par RPC. En effet, le parallélisme est de type fonctionnel où une procédure définie l'instruction atomique de l'application. Comme pour CORBA ou DCOM, les directions des communications entre les tâches et la mémoire globale sont explicites.

Les tâches créées sont considérées comme n'ayant pas d'effet de bord. Ainsi, les seules interactions avec l'environnement sont effectuées à travers les paramètres des procédures. Les données accédées par une procédure sont spécifiées dans la liste des paramètres formels selon un typage spécifique définissant les droits d'accès aux données (lecture `Shared_r`, écriture `Shared_w`, modification `Shared_rw`). L'algorithme présenté en figure 3.9 représente le calcul de l'intégrale d'une fonction $f(x)$ suivant l'API Athapascan.

La sémantique d'un programme Athapascan est intuitive car directement basée sur un ordre d'exécution proche d'une exécution séquentielle : tout accès en lecture voit la dernière écriture en suivant l'ordre séquentiel d'exécution.

Cilk et Athapascan masquent l'architecture matérielle dans le sens où aucune référence au nombre disponible de processeurs n'est faite au niveau de leur API. En effet, d'après les exemples présentés sur les figures 3.8 et 3.9, le programme est construit en développant le parallélisme jus-

Début du code

```
void Afficher(a1::Shared_r résultat){
  print(résultat.read());
}

void Somme(a1::Shared_r res_partiel1, a1::Shared_r res_partiel2, a1::Shared_w res_partiel){
  res_partiel.access() = res_partiel1.read() + res_partiel2.read();
}

void Calc_Intégrale(a, b, Seuil, a1::Shared_w res_partiel){
  if(b-a <= Seuil)
    res_partiel.access() = Intégrale(a, b);
  else{
    a1::Shared res_partiel1;
    a1::Shared res_partiel2;
    //Calcul du pas de discrétisation n
    n = (b-a)/2;

    a1::Fork<Calc_Intégrale> (a, n, res_partiel1);
    a1::Fork<Calc_Intégrale> (n, b, res_partiel2);
    a1::Fork<Somme> (res_partiel1, res_partiel2, res_partiel);
  }
}

//Programme principal
void main(a, b, Seuil){
  //a, b  : intervalle
  //Seuil : pas de discrétisation fixé par l'utilisateur

  a1::Shared résultat; //valeur de l'intégrale de f(x)

  //identifie le processeur maître : initialement, tous les processeurs exécutent le main ;
  //cependant seul l'un d'entre eux doit générer le parallélisme
  if(Processeur_maître){
    a1::Fork<Calc_Intégrale> (a, b, Seuil, résultat);
    a1::Fork<Afficher> (résultat);
  }
}
```

Fin du code

FIG. 3.9 – Pseudo code du calcul de l'intégrale d'une fonction $f(x)$ suivant le modèle de programmation Athapascan.

qu'à atteindre un seuil fixé par l'utilisateur. Au contraire, pour les environnements dits de bas niveau, les exemples des figures 3.5, 3.6, 3.7 font tous état du nombre de processeurs afin de décomposer le travail sur les entités de calcul. Un des avantages motivant l'utilisation des environnements de haut niveau décharge au programmeur le calcul du repliement (algorithmes d'ordonnancement) du parallélisme des tâches de son application sur le parallélisme matériel. En contrepartie, la classe des applications traitée est restreinte à celles des applications de type série-parallèle sous les hypothèses énoncées dans [42, 16].

Les environnements de haut niveau ont été explicitement conçus afin d'exécuter des applications parallèles d'une manière performante. La section suivante présente les deux principales métriques qualifiant l'exécution d'un programme sur une architecture parallèle.

3.5 Métriques qualifiant les performances d'une application parallèle

Parmi les principaux critères d'évaluation d'une application parallèle, nous retenons principalement deux métriques : l'accélération et l'efficacité. Ces deux métriques sont généralement utilisées dans la communauté où la rapidité est le facteur prédominant à toute application parallèle.

3.5.1 Accélération

Le **temps d'exécution séquentiel** d'une application, noté T_s, correspond au temps écoulé entre le début et la fin de son exécution sur une architecture monoprocesseur. T_s traduit le coût de la méthode de calcul utilisée par l'application et ne contient aucun surcoût lié au parallélisme. Son **temps d'exécution parallèle**, noté T_p, correspond à la durée d'exécution de l'algorithme parallèle sur p processeurs. Cette durée correspond au temps écoulé entre le moment où le calcul parallèle débute et le moment où le dernier processeur termine son exécution.

Lors de l'évaluation d'un système parallèle, l'une des métriques permettant de quantifier la performance d'une application parallèle est l'accélération. Elle est définie comme le gain de performance obtenu *via* la parallélisation d'une application par rapport à sa meilleure implantation séquentielle. L'accélération S_p est formellement définie comme le rapport entre le temps requis pour résoudre un problème sur une architecture monoprocesseur (en considérant sa version séquentielle la plus performante) et le temps mis pour résoudre ce même problème sur une machine parallèle :

$$S_p = \frac{T_s}{T_p} \tag{3.1}$$

La caractérisation de T_s est soumise à interprétation. Il est courant de retrouver dans la littérature deux méthodes permettant de calculer sa valeur selon le choix de l'algorithme utilisé. Il peut être considéré comme étant le meilleur algorithme séquentiel connu ou comme étant l'algorithme parallèle s'exécutant sur un unique processeur. Dans notre étude, nous utiliserons T_s comme calculé à partir de l'algorithme parallèle.

3.5.2 Efficacité

Comme l'accélération, l'efficacité, spécifie une information quantitative. L'efficacité est formellement définie comme représentant l'utilisation moyenne des processeurs pour une application parallèle et mesure le rendement de l'utilisation des processeurs.

Idéalement, en utilisant p processeurs, une accélération de p doit être obtenue. En pratique, ce comportement idéal n'est jamais atteint étant donné que les processeurs participant à l'exécution ne passent pas 100% de leur temps d'activité dans les calculs de l'algorithme. Nous pouvons lister deux facteurs limitant l'efficacité. Le premier facteur provient directement de la nature de l'application. Généralement, une application n'est pas 100% parallèle : elle comporte des phases

intrinsèquement séquentielles où seul un ensemble restreint de processeurs est utilisé. Sur les processeurs inactifs, cela se traduit par une attente. Le second facteur provient des sur-coûts liés à la gestion du parallélisme. Les synchronisations ainsi que les communications n'interviennent pas dans une exécution séquentielle. Malgré les techniques de recouvrement des communications par des calculs ou bien encore de *parallel slackness* [114], synchronisations et communications engendrent un certain surcoût.

L'efficacité E_p peut être vue comme une normalisation du facteur d'accélération de l'algorithme parallèle ou encore comme le rapport entre l'accélération effective et l'accélération optimale. L'efficacité est une valeur comprise entre 0 et 1 mais qui est généralement exprimée sous la forme d'un pourcentage.

$$E_p = \frac{S_p}{p} \tag{3.2}$$

Cette valeur est couramment utilisée afin de qualifier le comportement d'une application ou d'un environnement de programmation lorsque le nombre de processeurs p augmente. En effet, p étant intégré dans le calcul de l'efficacité, l'étude du passage à l'échelle est alors envisageable.

3.6 Bilan

L'efficacité des algorithmes parallèles se rapporte classiquement à des modèles théoriques de machines dans lesquelles les caractéristiques réelles des architectures sont simplifiées. Un modèle de programmation parallèle permet d'abstraire une architecture à travers un ensemble de primitives ou d'instructions masquant la complexité des architectures.

Néanmoins, ils n'abstraient pas tous l'architecture de la même manière. Nous avons présentés certains *aspects* nécessaires au développement d'un algorithme parallèle nous permettant d'introduire une taxinomie des modèles de programmation se distinguant par les abstractions ou fonctionnalités qu'ils offrent.

Basés sur ces modèles, les environnements de programmation parallèle offrent des interfaces de programmation permettant d'exploiter plus ou moins finement le parallélisme de l'architecture et de l'application. Les environnements bas niveaux s'attardent à gérer uniquement le parallélisme effectif de l'architecture matérielle. Pour cela, ils exposent des primitives liées à la gestion des synchronisations et du mouvement des données. Ainsi, la décomposition du problème en activités visant à être distribuées sur les différents processeurs est laissée totalement à la charge du concepteur de l'application. Les environnements de haut niveau ont tenté le pari d'abstraire complètement l'architecture matérielle et de laisser le concepteur se concentrer sur l'algorithmique parallèle.

Les environnements de haut niveau ne se sont jamais réellement imposés à la communauté. En effet, l'abstraction générée par un modèle haut niveau engendre généralement la gestion d'un parallélisme restreint (par exemple de type série-parallèle). Néanmoins, vu la facilité du développement d'algorithmes et les performances qu'ils obtiennent pour ce type de parallélisme, ils sont à la base de solution minimisant le coût global de développement sur grappe ou grille de calcul. Le cœur de ces environnements repose sur leur capacité à placer et ordonnancer les calculs sur les différents processeurs.

Les deux métriques qualifiant l'exécution parallèle d'une application se basent sur le calcul de l'accélération et de l'efficacité. L'accélération qualifie la performance brut de l'exécution parallèle. L'efficacité ramène cette valeur au nombre de processeurs utilisé pour obtenir un rendement. Les critères conditionnant l'obtention de performance reposent, évidemment, sur l'utilisation d'un environnement de programmation performant et adapté à l'architecture matérielle cible. Toutefois, une exécution performante est fortement influencée par les critères de placement et d'ordonnancement des activités de l'application. L'étude des algorithmes réalisant de tels traitements est un axe de recherche actif. Le chapitre suivant présente les techniques et algorithmes de partitionnement et de placement des tâches élémentaires en vue de replier le parallélisme applicatif sur une architecture parallèle.

Ordonnancement 4

4.1 Introduction

Une architecture parallèle est définie comme un ensemble de ressources offrant une grande puissance de calcul. Une application parallèle utilise ces ressources afin de résoudre un même problème. Pour cela, l'algorithmique parallèle propose des paradigmes de programmation qui permettent de définir suffisamment de concurrence entre les activités de l'application afin d'exploiter les architectures parallèles. L'un des grands challenges lié à l'obtention de performances sur une architecture parallèle repose sur l'ordonnancement de ces activités.

L'ordonnancement permet généralement de calculer une politique globale et locale d'exécution. La politique globale d'exécution affecte un site d'exécution à chaque activité de l'application parallèle : on parle de phase de regroupement ou encore de partitionnement.

La politique locale définit quelle sera l'ordre total d'exécution des activités d'une partition affectée à un processeur. Il convient de noter que les dépendances de données définissent seulement un ordre partiel. En effet, à un instant t, il est possible que plusieurs activités soient prêtes à être exécutées, c'est-à-dire que les dépendances de données pour chacune d'entre elles sont respectées.

Pour réaliser de tels traitements, les algorithmes d'ordonnancement nécessitent une vision à plus ou moins long terme du futur de l'exécution. Un graphe de tâches définit la représentation abstraite de l'exécution de l'application.

Suivant la nature du programme, cette représentation peut être ou non déterminée à l'avance. Les auteurs de [46] définissent trois grandes classes d'applications générant ou non un représentation prévisible :

1. **Applications régulières**. Cette classe regroupe les applications pour lesquelles il est possible de déterminer une représentation abstraite avant le démarrage de l'exécution.
2. **Applications irrégulières**. Le schéma d'exécution des activités de ce type d'applications est imprévisible. La représentation ne peut être construite qu'à l'exécution même.
3. **Applications semi-régulières**. L'application est structurée en une succession de phases où chacune d'elles définit une application régulière. Cependant, leur enchaînement est imprévisible.

Le fait de disposer d'une représentation de l'exécution d'une application avant son démarrage conditionne l'utilisation des classes d'ordonnanceurs.

Une approche statique définit une construction du graphe de tâches avant toute exécution : ainsi la classe des ordonnanceurs statiques peut être employée. Cette classe calcule les politiques globales et locales sur la représentation complète de l'exécution. Les algorithmes nécessitent généralement un graphe pondéré par les coûts d'exécution et de communication. A partir de ces informations et du graphe complet de l'exécution, ils calculent les politiques globale et locale afin d'approcher un critère à atteindre tel que la minimisation du temps de complétion de l'application.

Au contraire, une approche dynamique construit le graphe au moment de l'exécution. Généralement, le graphe est *causalement* connecté à l'exécution où l'appel d'une instruction parallèle de l'interface de programmation engendre la création d'une tâche dans le graphe. Ainsi, à tout moment, un algorithme d'ordonnancement ne possède qu'une vision restreinte du futur de l'exécution. Avec cette connaissance limitée, un ordonnanceur dynamique définit les politiques d'exécution des tâches connues.

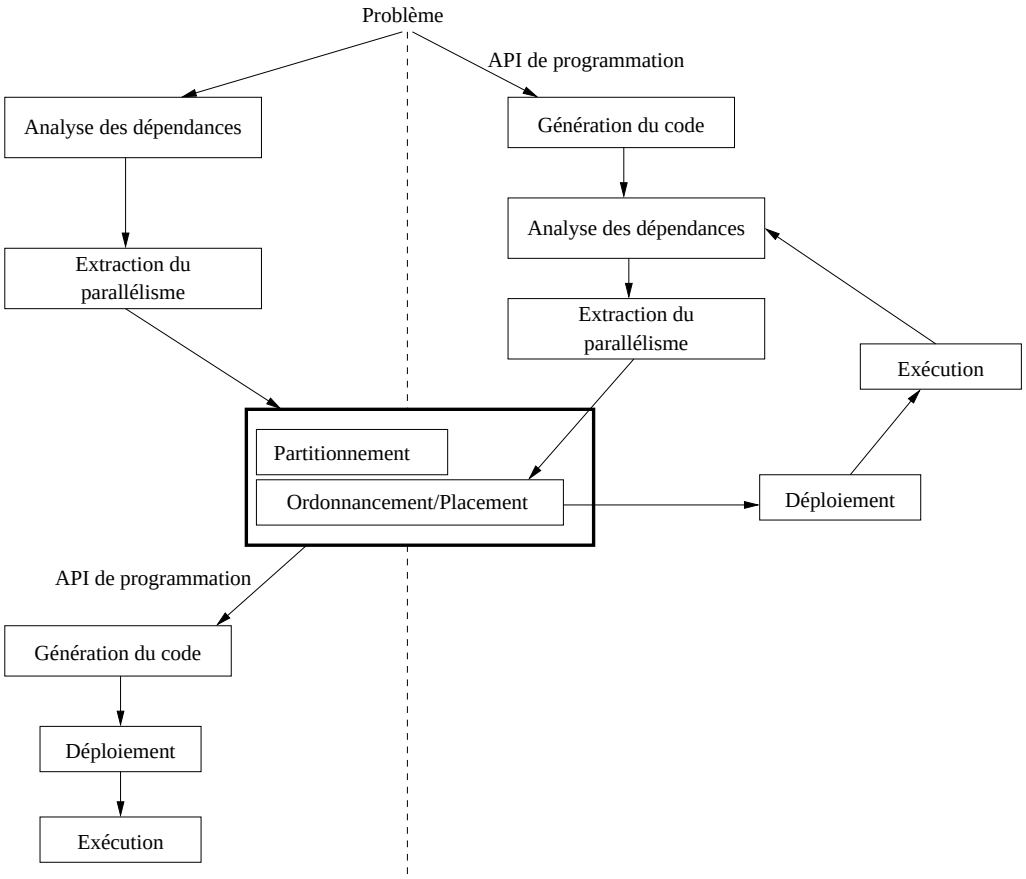
Dans ce chapitre, nous exposons le schéma d'exécution d'une application parallèle selon les deux classes d'ordonnanceurs. Etant donnée que tout algorithme d'ordonnancement se base sur la connaissance à plus ou moins long terme du futur de l'exécution, la section 4.3 est consacrée aux différentes représentations possibles offrant de telles spécifications. Enfin, nous décrivons les deux classes d'ordonnanceurs et plus particulièrement la classe statique en nous appuyant sur la présentation de certains algorithmes.

4.2 Schéma d'exécution d'une application parallèle

Dans le chapitre précédent, nous avons étudié quelques environnements de programmation permettant de faciliter l'exploitation des architectures parallèles. Selon le modèle, les problématiques d'ordonnancement sont, soit à la charge du programmeur, soit masquées par l'environnement de programmation haut niveau. Malgré l'abstraction offerte par les environnements haut niveau, l'ordonnancement des tâches de l'application fait partie intégrante de leur modèle d'exécution. La figure 4.1 présente le schéma d'exécution suivant les classes d'ordonnanceurs utilisées : statique et dynamique.

Quelque soit l'approche retenue, les mêmes traitements sont appliqués. A partir de la définition du problème, l'analyse des dépendances de données entre les tâches permet d'extraire le parallélisme d'une application sous la forme d'une représentation abstraite : un graphe de tâche. Ce graphe est indispensable à tout algorithme d'ordonnancement. L'ordonnanceur fixe alors un site d'exécution ainsi que la date de début calculée de chaque tâches du graphe en vue de les déployées sur le processeur physique qui en réalisera l'exécution. Ce schéma d'exécution définit complètement la classe des ordonnanceurs statiques étant donné que le graphe total est connu.

Pour les classes d'applications irrégulières où cette représentation ne peut être obtenue à l'avance, le graphe ordonnancé ne représente qu'un sous-graphe de l'exécution totale. L'exécution des tâches en crée de nouvelles qui, soumises au processus d'analyse des dépendances et d'extraction du parallélisme, génèrent un nouveau sous-graphe qui est à son tour soumis à l'ordonnanceur.



(a) Approche statique **(b) Approche dynamique**

FIG. 4.1 – Schémas d'exécution selon l'utilisation des deux classes d'ordonnanceurs des tâches de l'application : une approche statique (a) ; et une approche dynamique (b).

Ainsi tout algorithme d'ordonnancement nécessite la définition d'un graphe de tâches en entrée afin de calculer les politiques nécessaires à une exécution performante des tâches. La section suivante présente les trois types de représentation possibles.

4.3 Représentation abstraite de l'exécution

L'exécution d'un algorithme parallèle peut être décrite comme l'exécution d'une succession d'instructions sur des données partagées suivant un ordre permettant d'en respecter les dépendances. Cette description permet de définir un graphe où les nœuds représentent les tâches (*i.e.*, les instructions) et les arêtes représentent les mouvements de données. Ce graphe peut être, soit un graphe de dépendance (noté *DG*), soit un graphe de précédence (noté *PG*) ou soit un graphe de flot

de données (noté *DFG*). Une représentation est généralement adaptée à la résolution d'un critère de performance particulier (minimisation du *makespan* ou du volume de données à communiquer). En pratique, la représentation sous forme d'un graphe de précédence est préférée car elle expose un degré suffisant de détails pour définir synchronisation et échange de données.

4.3.1 Graphe de dépendance

Un graphe de dépendance DG est défini par l'ensemble $\{V, E\}$ où V représente l'ensemble des sommets et E l'ensemble des arêtes. V et E définissent respectivement l'ensemble des tâches et l'ensemble des dépendances entre ces dernières. Ainsi, l'existence d'une arête (u, v) entre les tâches u et v entraîne une relation de dépendance entre ces deux tâches. Ce graphe est non-orienté et ne permet pas de représenter les contraintes de précédence. La figure 4.3 (a) présente une telle représentation associée au programme de la figure 4.2.

Début du code

```
void func(int x){
  int y1, y2, y3, y4, y5;

  n1(in x, out y1, out y2);
  n4(in x, out y3);
  n5(in x, out y4);
  n2(inout y1);
  n3(in y2, out y5);
  n6(inout y5, in y3, in y4);
  n7(in y1, in y5);
}
```

Fin du code

FIG. 4.2 – Pseudo code d'un algorithme exposant un schéma de dépendance des calculs.

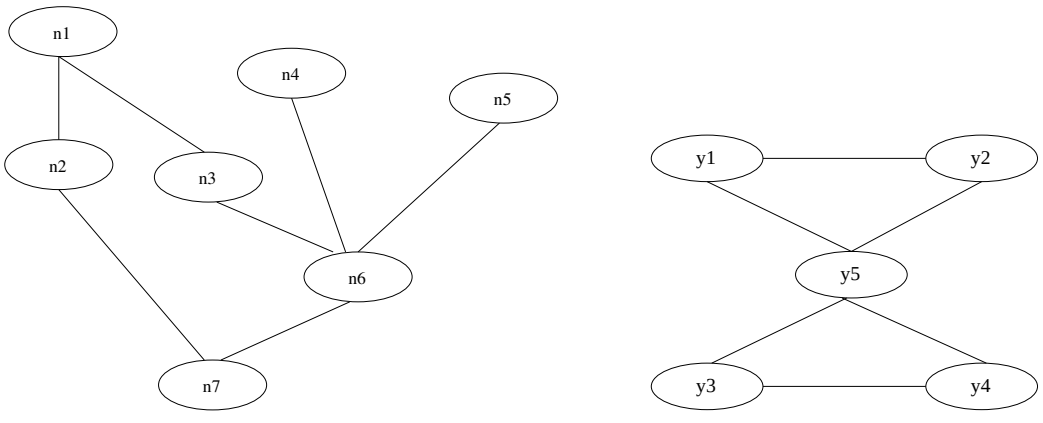


FIG. 4.3 – Graphe de dépendance de l'algorithme 4.2. En (a), graphe de dépendance des tâches ; en (b) graphe de dépendance des données.

Il est possible de considérer comme représentation abstraite non plus le graphe de tâches mais son dual : le graphe de données. Nous verrons plus en avant dans quel cas le graphe dual peut être plus approprié à la représentation de l'exécution. La figure 4.3 (b) représente le graphe de données de l'algorithme 4.2. Ce graphe est défini par $DG_{\text{dual}} = \{V, E\}$ où l'ensemble des sommets V représentent l'ensemble des données et l'ensemble des arêtes E définit l'ensemble des dépendances entre les données. Ainsi, une arête (d_1, d_2) entre les données d_1 et d_2 décrit qu'il existe au moins une tâche qui dépend de ces deux données (soit en entrée, soit en sortie, soit en entrée/sortie).

Cette représentation se base exclusivement sur la description des communications entre les diverses tâches d'une application. Sur des schémas complexes d'exécution où les tâches sont fortement couplées, la détection des tâches concurrentes est impossible. Par exemple, en considérant l'algorithme 4.2, les tâches `n1`, `n4` et `n5` sont concurrentes : leur exécution ne dépend pas de données produites par une tâche en prédécesseur. Or, selon cette représentation, on remarque qu'il existe une chaîne reliant ces différents sommets si bien qu'il est impossible de définir si les tâches `n1`, `n4` et `n5` sont concurrentes ou non.

Nous verrons dans la section suivante comment la bibliothèque de partitionnement METIS [68] utilise une telle représentation afin de générer un partitionnement dans un but de répartition équitable de la charge entre les sites d'exécution.

4.3.2 Graphe de précédence

Dans la représentation d'un programme par un graphe de précédence, les tâches sont représentées par les sommets du graphe. Les dépendances entre les tâches sont représentées par des arcs orientés. L'existence d'un arc (u, v) d'une tâche u à une tâche v entraîne que v ne peut être exécutée sans les données produites par la tâche u. Comme pour la représentation sous la forme d'un graphe de dépendance, les arcs définissent les relations de dépendance entre les tâches mais apportent également une information sur l'ordre d'exécution.

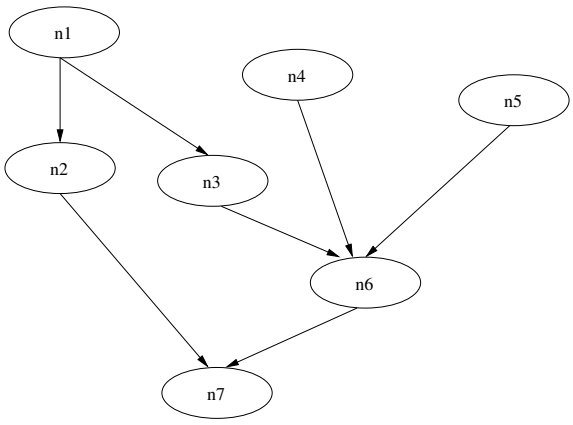


FIG. 4.4 – Graphe de précédence de l'algorithme 4.2. Les tâches sont représentées par des cercles et les arcs définissent les mouvements de données.

Reprenons l'algorithme 4.2 défini précédemment, son graphe de précédence est présenté sur la figure 4.4. Ce graphe s'apparente fortement au graphe de dépendance présenté sur la figure 4.3 (a). En effet, seule l'orientation des arêtes est ajoutée. Cependant, ce graphe G, orienté et acyclique permet de définir un **ordre partiel strict** sur l'ensemble V des tâches. Cette relation, noté $\prec$ traduit la précédence entre les tâches V. Ainsi, s'il existe un chemin dans G de t à t' alors $t \prec t'$. Une autre manière de voir ceci est que deux tâches t et t' suivent une relation de précédence, $t \prec t'$, si la tâche t doit être exécutée avant la tâche t'.

Si $\neg(t \prec t')$ et $\neg(t' \prec t)$, alors les tâches t et t' sont indépendantes signifiant que l'ordre, dans lequel les tâches t et t' sont exécutées, est sans importance. Ainsi, en opposition à la précédente représentation, il est possible de qualifier ces tâches comme concurrentes. Cependant, elle n'identifie pas les données à communiquer. C'est pourquoi, afin de schématiser cette information, la représentation sous forme de graphe de flot de données est nécessaire.

4.3.3 Graphe de flot de données

Le graphe de flot de données (DFG pour *Data Flow Graph*) associé à une exécution est le graphe $G = \{V, E\}$ tel que les tâches V_t et les données V_d forment l'ensemble des nœuds $V = V_t \cup V_d$, et les accès des tâches aux données E forment l'ensemble ϵ des arcs. Ce graphe est orienté et biparti étant donné que $\epsilon \subset (V_t \times V_d) \cup (V_d \times V_t)$: un nœud tâche est connecté à un ou plusieurs nœuds données ; un nœud donnée est connecté à une ou plusieurs tâches. Un arc dans le graphe définit une synchronisation de type lecture ou écriture entre une donnée et une tâche. Soient une tâche $t \in V_t$ et une donnée $d \in V_d$, alors :

- l'arc $(t, d) \in \epsilon$ détermine un droit d'accès en écriture de la tâche t sur la donnée d.
- l'arc $(d, t) \in \epsilon$ définit un droit d'accès en lecture de la tâche t à la donnée d.

En pratique, cette représentation est plus fine que la précédente étant donné que l'identification des données à communiquer est explicitement représentée. Cette représentation est particulièrement adaptée aux approches dynamiques où la connaissance du graphe n'est pas prévisible *a priori*. En effet, il permet de décrire explicitement quelles sont les données à communiquer nécessaires à la cohérence de l'exécution.

La représentation graphique de l'algorithme 4.2 sous la forme d'un graphe de flot de données est présentée sur la figure 4.5. Sur ce graphe, les variables $y1$ et $y5$ sont toutes deux représentées par deux données. Dans la construction de cette représentation à partir du code de la figure 4.2, les données du graphe représente différentes versions des variables $y1$ et $y5$. La tâche $n2$ consomme la donnée $y1_1$ correspondante à la version en entrée et produit une nouvelle version de la variable $y1_2$. Il en va de même pour la tâche $n6$ avec la variable $y5$. Les données du graphe représente des variables en assignation unique.

4.4 Placement / Ordonnancement

L'obtention de performances nécessite le calcul des politiques globale et locale d'exécution pour chaque tâche composant le graphe de tâches.

- Le calcul de la politique globale affecte un site d'exécution à chaque tâche. Ce traitement, couramment appelé placement, regroupe les tâches selon leur affinité à communiquer. Le

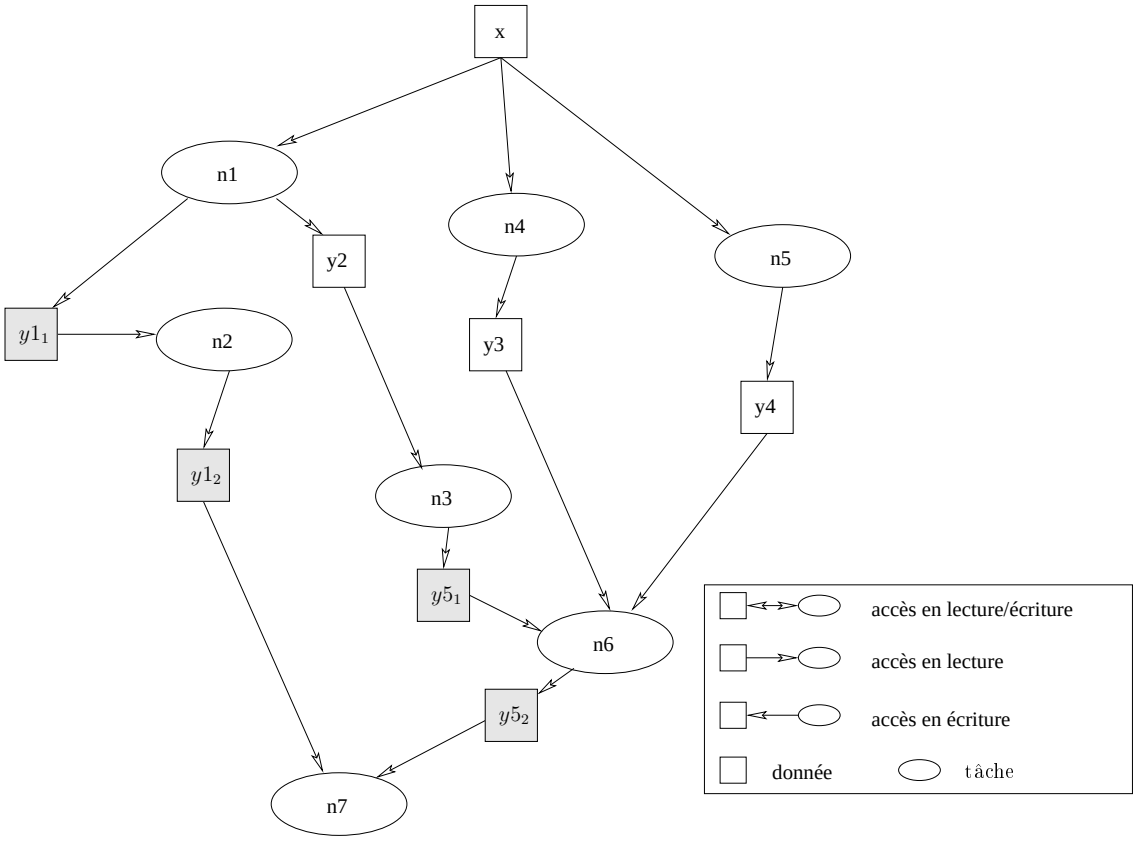


FIG. 4.5 – Graphe de flot de données de l'algorithme 4.2. Les tâches sont représentées par des cercles et les données par des rectangles. L'orientation des arcs marque les accès des tâches aux données.

placement de deux tâches sur un même site vise à annuler le temps de communication nécessaire au respect des contraintes de dépendance des données.

– Un ordonnanceur fixe également la date de début de chaque tâche permettant de déduire la politique locale d'exécution. En effet, à un instant t, il est possible que deux tâches soient prêtes à être exécutées. Ce critère définit un ordre total d'exécution pour chaque site.

Il convient de remarquer qu'un site d'exécution représente généralement un processeur physique. En effet, les algorithmes d'ordonnancement traités définissent leurs calculs pour des architectures matérielles homogènes et monoprocesseur. D'autres, prennent également en entrée un modèle d'architecture, leur permettant de tenir compte de la topologie d'interconnexion des machines, par exemple.

Afin de calculer placement et ordre d'exécution, deux classes d'ordonnanceurs sont définies selon la connaissance ou non du graphe de tâches de l'application :

1. Les algorithmes statiques permettent de calculer un ordonnancement avant l'exécution de l'application à partir d'une représentation abstraite sous la forme d'un graphe de tâches. Afin de proposer un bon ordonnancement, le graphe est généralement pondéré par les coûts d'exécution et de communication.
2. Les algorithmes dynamiques proposent de calculer un ordonnancement au cours de l'exécution de l'application. Etant donné qu'ils ne présupposent pas une connaissance *a priori* du graphe ainsi que des coûts associés à l'exécution des tâches et aux communications, ces algorithmes sont très populaires.

Bien que les algorithmes dynamiques semblent plus simples d'utilisation, la politique de placement peut être mal adaptée à l'exécution globale de l'application. L'une des heuristiques les plus employées actuellement repose sur un algorithme de liste distribuée de type vol de travail (*work stealing*) [16]. Selon cet algorithme, les processeurs inactifs cherchent à récupérer du travail sur d'autres processeurs. Dans le cas d'applications de type série-parallèle, le nombre de vols et donc de communications est faible. Cependant, sur des graphes plus généraux, il peut vite devenir le goulot d'étranglement de l'exécution. Cilk et KAAPI, cités en section 3.4 page 3.4, utilise une telle technique d'ordonnancement.

Dans notre étude, nous ne considérons que des graphes prévisibles si bien que les approches statiques sont plus adaptées. Cette section vise ainsi à décrire quelques algorithmes d'ordonnancement statique.

Majoritairement, ces algorithmes sont spécialisés dans la résolution de stratégies précises : ils visent à minimiser une certaine fonction objectif.

4.4.1 Fonctions objectifs

Il est possible de dénombrer deux grandes stratégies qui motivent le calcul d'un ordonnancement [32].

1. L'objectif le plus élémentaire consiste à minimiser le temps global d'exécution ou *makespan* de l'application considérée. Pour cela, dès qu'un processeur devient inactif, et qu'il existe une tâche prête à être exécutée, celle-ci lui est affectée.
2. L'équilibrage de charge visent à maintenir une charge de travail équivalente sur chaque processeur de l'architecture parallèle. Sur une topologie quelconque, cette stratégie ne permet pas de garantir que le temps d'exécution parallèle sera minimal. Nous verrons néanmoins qu'une telle stratégie peut être utilisée à bon escient pour des classes de problèmes spécifiques.

Lorsque l'on considère un cadre large d'applications, les algorithmes de *makespan* (en pratique, implantés sous la formes d'algorithmes gloutons ou de liste) permettent d'obtenir un temps d'exécution très proche de l'optimal et ceci d'autant plus que l'application expose un degré de parallélisme important. C'est pourquoi, ils sont majoritairement employés afin de calculer un ordonnancement. Cette propriété fut prouvée par Graham [51] et est définie comme suit :

Propriété 4.1 *Soit une machine parallèle homogène exposant un espace d'adressage unique supposé sans coût de communication. Soient T_1 et T_∞ représentant respectivement la durée d'exécution séquentielle et la durée d'exécution parallèle sur un nombre infini de processeur (cette durée correspond au temps nécessaire à l'exécution du chemin critique du graphe). Alors, tout ordonnancement de type partage de charge sur p processeurs a une durée d'exécution T_p bornée par :*

$$\max(\frac{T_1}{p}, T_\infty) \leq T_p \leq \frac{T_1}{p} + T_\infty$$

4.4.2 Algorithmes de minimisation du *makespan*

Les problèmes d'ordonnancement dans le domaine du calcul parallèle sont NP-difficiles sauf dans de rares cas spécifiques [30]. C'est pourquoi des heuristiques sont proposées afin d'obtenir une solution en un temps raisonnable. De nombreuses ont été développées théoriquement mais, en pratique seules celles basées sur des algorithmes de liste de type glouton sont implémentées [50, 75].

Définition 1 *Un algorithme glouton construit une solution de façon incrémentale, en choisissant à chaque étape la direction qui semble la plus prometteuse. Ce choix, localement optimal, ne présuppose aucunement que le solution globale sera optimale. La caractéristique fondamentale sous-jacente à ce type d'algorithmes repose sur l'absence de remise en cause des choix préalablement réalisés.* □

Définition 2 *Une heuristique de type liste fournit une solution dont la qualité, i.e. le temps d'exécution de l'ordonnancement, est à un facteur connu de la solution optimale. Ce facteur est appelé garantie de performance.* □

Les algorithmes de liste maintiennent une liste des tâches prêtes à ordonnancer selon leur priorité. Les grands avantages des algorithmes de liste sont la simplicité et l'existence de garanties de performance. Un algorithme de liste répète les traitements suivants jusqu'à l'allocation de toutes les tâches :

1. Les tâches prêtes sont placées dans une liste de priorité. Une tâche est prête lorsque tous ses prédécesseurs ont été déjà alloués à un processeur ;
2. Un processeur approprié est choisi ;
3. La tâche en tête de la liste est allouée sur le processeur choisi.

L'idée clé de l'heuristique de liste repose sur l'exécution des tâches disponibles présentes dans une liste de tâches prêtes et classées par priorité. Ainsi, la différence majeure entre les algorithmes est dictée par l'affectation des priorités aux tâches prêtes : la tâche ayant la plus forte priorité sera exécutée en premier. Une propriété intrinsèque à de tels algorithmes est qu'on ne laisse pas délibérément un processeur inactif alors qu'il existe des tâches prêtes non encore allouées.

4.4.2.1 ETF

L'algorithme ETF (*Earliest Task First*) [60] calcule l'ordonnancement des tâches d'un graphe de précédence sur une machine à mémoire distribuée composée de p processeurs identiques reliés les uns aux autres par un réseau de communication complètement maillé et considéré sans contention. Ce graphe explicite les relations de dépendance entre les tâches qui correspondent aux communications de données.

Le principe d'ETF est d'obtenir pour chaque tâche la date de démarrage au plus tôt. Ainsi, il considère deux listes : la première représentant les tâches prêtes non encore affectées et la seconde définit l'ensemble des processeurs. Pour cela, les délais de communication engendrés par le placement d'une tâche sur un processeur p_i sont considérés ainsi que et la date de disponibilité de p_i. Le calcul de l'ordonnancement teste toutes les combinaisons (tâches, processeurs) afin d'affecter les tâches aux processeurs qui pourront les exécuter au plus tôt.

La complexité d'ETF est en $O(v^2p)$ [60] où v représente le nombre de nœuds du graphe et p le nombre de processeurs utilisés. Il offre une garantie de performance $(2 - \frac{1}{p})T_\infty + C_\infty$ [9] avec T_∞ le temps d'exécution minimal sur un nombre non borné de processeurs (chemin critique du graphe) et C_∞ la somme maximale des temps de communication sur le plus long chemin du graphe.

4.4.2.2 DSC

DSC [124] (*Dominant Sequence Clustering*) fut initialement pensé pour traiter le problème de regroupement des tâches en partition. Pour cela, il s'appuie sur un graphe de précédence pondéré sur les nœuds et sur les arcs. Comme ETF, il considère une architecture à mémoire distribuée composée de p processeurs homogènes interconnectés par un réseau de communication complètement maillé et sans contention.

DSC vise à agglomérer les tâches d'un graphe de précédence pondéré dont l'objectif est la minimisation du *makespan*. Dans sa forme la plus élémentaire, il permet de générer un nombre non borné de partitions.

Le principe de cet algorithme repose sur l'augmentation du grain des tâches de l'application par la création de groupes. A chaque étape, l'algorithme consiste à fusionner la tâche de plus haute priorité avec un groupe qui minimise sa date de démarrage. Cette algorithme diffère d'ETF dans le sens où la priorité de chaque tâche est une fonction considérant la date de démarrage au plus tôt ainsi que la somme du travail restant à effectuer sur la plus longue séquence de calcul le reliant à la fin du programme. Ainsi, la plus haute priorité revient à la tâche qui appartient au chemin critique à un instant t. Ce chemin est nommé séquence dominante.

Cet algorithme expose une complexité en $O((v + e) \log v)$ où v représente les nœuds et e les arcs. La garantie de performance est $1 + \frac{1}{\rho}$ avec ρ étant la granularité du graphe de précédence, c'est-à-dire le rapport calcul/communication.

Nous avons présenté DSC sous la forme d'un algorithme de regroupement pour un nombre illimité de partitions/processeurs. Ainsi, la bibliothèque PYRROS [122] qui s'appuie sur DSC a motivé la mise en place d'une technique de repliement des k partitions sur les p processeurs effectifs de l'architecture. La thèse de Yang [121] détaille l'algorithme d'équilibrage de charge utilisé qui répartit les groupes de manière à ce que chaque processeur ait sensiblement la même charge de calcul. La complexité d'un tel algorithme est en $O(vlogv)$.

4.4.3 Algorithmes de type équilibrage de charge

Le but d'un algorithme de type équilibrage de charge consiste à produire des groupes de tâches de poids sensiblement équivalent. Ces algorithmes nécessitent un graphe de dépendance pondéré afin de réaliser un tel traitement.

Généralement, un deuxième critère est retenu permettant de minimiser les communications entre les groupes formés.

Il existe de nombreux algorithmes de partitionnement de graphe [38] mais très peu se montrent à la fois efficaces et applicables à faible coût. Une classe d'algorithmes particulièrement intéressante est celle des algorithmes de partitionnement multi-niveaux, notamment ceux proposés dans METIS et SCOTCH.

4.4.3.1 METIS et SCOTCH

METIS [68] est une bibliothèque offrant plusieurs méthodes de partitionnement de graphes. Elle se base sur une représentation de type graphe de dépendance.

METIS a pour objectif d'agglomérer les tâches d'un graphe de dépendance pondéré en générant des partitions équilibrées et ceci tout en minimisant la somme des poids des arêtes entres les différents groupes. Il présuppose que toutes les tâches sont créées au début de l'exécution et démarrées immédiatement. Ainsi, il regroupe les tâches selon leur affinité à communiquer : les tâches qui communiquent beaucoup sont alors regroupées au sein d'une même partition.

METIS considère également une architecture matérielle homogène ainsi qu'une topologie de communication complètement maillée. La résolution du problème nécessite trois étapes :

1. Regroupement (*coarsing phase*) : cette phase réduit la taille du graphe afin que l'étape de partitionnement puisse être effectuée efficacement. Elle repose alors sur le regroupement des tâches opérant sur un même ensemble de données, c'est-à-dire que les tâches reliées par des arêtes ont un poids fort. Ce traitement est répété jusqu'à ce que le graphe ait atteint une taille suffisamment faible pour que l'algorithme de partitionnement soit efficace.
2. Partitionnement (*partitioning phase*) : le partitionnement du graphe réduit est calculé. Plusieurs algorithmes peuvent être utilisés comme par exemple Kernighan-Lin [71] ou Fiduccia-Mattheyses [37].
3. Raffinement (*uncoarsing phase*) : le partitionnement calculé à l'étape précédente sur un graphe restreint est raffiné en le projetant pas à pas vers le graphe initial.

Les travaux de Karypis et Kumar [67] donnent une analyse théorique des algorithmes multi-niveaux ne considérant que des graphes correspondant à des schémas de calcul itératifs.

De nombreux problèmes de simulation (notamment en décomposition de domaines) se ramènent à un calcul itératif où, à chaque pas de l'itération, un point de l'espace est mis à jour à partir des valeurs de ses voisins ainsi que de sa valeur à l'itération précédente. Suivant une classe d'application, la minimisation du surcoût des communications conduit donc à regrouper les points qui sont voisins de manière à distribuer équitablement les groupes entre les processeurs.

Un tel schéma de calcul peut être représenté par un graphe de dépendance où les nœuds représentent les points de l'espace et les arcs représentent les activités qui utilisent ces points. Selon cette définition, le graphe de dépendance est le graphe dual du graphe de tâches où les nœuds sont les données et les arcs représentent les traitements qui interagissent avec les données. Cette représentation est couramment adoptée étant donné que le graphe de tâches engendrerait un pauvre ordonnancement des calculs.

L'algorithme de partitionnement SCOTCH [88] vise les mêmes classes d'applications selon une technique sensiblement identique. Toutefois, il permet également de calculer un ordonnancement en considérant une machine cible hétérogène en terme de puissance processeur et de bande passante réseau. La machine cible est alors modélisée par un graphe non orienté annoté des métriques puissance des processeurs et capacité des liens d'interconnexion.

4.5 Bilan

Afin d'offrir une exécution performante, un environnement doit fournir soit dynamiquement, soit statiquement, une représentation abstraite de l'exécution. Une représentation statique ne peut être mise en œuvre que dans le cadre des applications régulières où le futur du calcul est connu *a priori*. Pour les applications irrégulières, seule une représentation dynamique peut être employée. Ces classes d'applications conditionnent fortement la classe d'ordonnanceur à utiliser.

Quelque soit le type d'ordonnanceur, il repose sur la connaissance du futur à plus ou moins long terme de l'exécution sous la forme d'un graphe. Les trois types de graphe exposent chacun des propriétés particulières. Un graphe de dépendance permet de répondre à une problématique d'équilibrage de charge. Un graphe de précédence est, quant à lui, indispensable à l'objectif du minimisation du *makespan*. Comme le graphe de dépendance, il expose les contraintes de dépendance entre les tâches mais représente également les besoins de synchronisation entre les tâches. Le graphe de flot de données est la représentation exposant le plus de détails. A partir de ce dernier, il est possible de reconstruire un graphe de précédence tout en apportant des propriétés utiles lors de l'exécution de l'application.

Un ordonnanceur dynamique fournit une bonne garantie de performance pour des applications exposant un parallélisme de type série-parallèle. Pour des applications dont le schéma de calcul est plus général, un ordonnanceur statique est préférable dans la mesure les estimations des coûts d'exécution des tâches et des coûts de communication des données sont connues à l'avance. Dans la suite de ce document, nous traitons le cas d'applications régulières où ces estimations peuvent

être obtenues.

Bien qu'il existe une recherche active autour des heuristiques pour l'ordonnancement d'applications, peu sont implantées effectivement. Pour des applications itératives de type décomposition de domaine, un algorithme de type équilibrage de charge peut être utilisé avec une bonne performance à l'exécution. Cependant, lorsque l'on considère des classes d'applications plus générales, l'heuristique de prédilection repose sur des algorithmes de liste glouton tels que DSC ou ETF. Ces algorithmes sont intéressants de par leur « *faible* » complexité et leur garantie de performance.

Ces trois premiers chapitres présentent le contexte d'étude de cette thèse où informatique parallèle et ingénierie des procédés assistée par ordinateur sont exposées séparément. Le chapitre suivant propose une étude des travaux de recherche relatifs à la distribution des simulations CAPE ainsi que ceux ayant pour objectif de proposer des environnements parallèles de couplage de codes applicables à la simulation des procédés.

Simulation des procédés et calcul parallèle 5

5.1 Introduction

Les applications de simulations numériques sont depuis toujours confrontées au compromis entre précision et temps d'exécution. Dans le domaine du CAPE, les recherches visant exploiter la pleine puissance des architectures matérielles (en vue de réduire les temps de simulation) se sont développées dès les prémices des architectures hautes performances. Cette prise d'intérêt a débuté quelques années après la sortie du Cray-1 en 1976, sur lequel Duerre et Bumb [36] ont installé l'environnement de simulation commercial modulaire séquentiel d'ASPEN. Ils ont obtenu une accélération comprise entre 2 et 3 par rapport à une station de travail IBM 4341-2. Cependant, ce gain n'était qu'un leurre : il n'était dû qu'à la différence de fréquence processeur entre les deux architectures. Néanmoins, de cette manière détournée, le HPC venait de s'offrir une place au côté de l'ingénierie des procédés.

Moe et Hertzberg [80] présentent un état de l'art complet des techniques utilisées pour paralléliser les applications de simulation des procédés. Bien qu'il soit ancien (1994), la majorité des techniques référencées est toujours d'actualité. La section 5.2 présente les travaux relatifs à l'introduction du calcul parallèle dans les simulations numériques du génie des procédés.

Les techniques actuellement prisées pour la conception de simulateurs reposent sur le couplage de codes hautement spécialisés. Par exemple, dans le domaine du CAPE, il est nécessaire de considérer des simulations multi-physiques où calculs thermodynamiques et hydrodynamiques doivent cohabiter pour la résolution de la simulation. Peu de travaux issus du domaine CAPE proposent des environnements d'exécution parallèle de simulation. Les recherches se focalisent principalement sur la parallélisation des méthodes de résolution numérique. La section 5.3 élargie notre présentation aux environnements parallèles de couplage de code.

5.2 Parallélisme et génie des procédés

Cette section présente les techniques de parallélisation des trois approches de résolution suivantes présenté dans le chapitre 2, page 33 :

– l'approche modulaire séquentielle ;

– L'approche orientée équations ;
– L'approche modulaire simultanée.

5.2.1 Approche modulaire séquentielle : parallélisme structurel

D'après les articles traitant du parallélisme appliqué à la simulation de procédés, l'approche modulaire séquentielle est la moins adaptée à une exécution parallèle [116, 56]. Ceci est intrinsèquement dû à l'approche de résolution qui, comme son nom l'indique, est séquentielle. Cependant, quelques travaux ont été menés afin de quantifier le gain envisageable.

Dans [13], Best décrit une approche visant à distribuer les ensembles d'opérations unitaires sur les nœuds de l'architecture. Son architecture est composée de quatre transputers Inmos (architecture propriétaire à mémoire distribuée). La distribution des opérations unitaires sur les entités d'exécution qui influence fortement l'accélération de l'exécution distribuée, est fixée par le concepteur de l'application.

L'obtention de performances pour les simulations statiques repose sur deux critères.

– Pendant les périodes d'inactivité des processeurs (*i.e.* les processeurs sont en attente des valeurs d'entrée), certains calculs thermodynamiques sont effectués. Dans cette approche, il est nécessaire de disposer de modules « *opération unitaire* » paramétrables et événementiels. Les codes existants ne peuvent pas être utilisés en l'état et les modifications sont complexes.
– La simulation statique génère une faible accélération : trois parmi les quatre cas tests exposent une accélération maximale 2. Cette accélération est obtenue sur le nombre maximal de processeurs (quatre). Seul le dernier cas engendre une accélération de $3,54$ sur quatre processeurs. Cette bonne efficacité est due majoritairement à un **parallélisme structurel** ou **topologique** spécifié par le schéma de procédé.

La simulation dynamique repose sur l'emploi d'une méthode d'intégration explicite. Selon cette méthode, les valeurs du pas de temps suivant $t+1$ sont calculées à partir des états et des valeurs d'entrée du pas de temps courant t. Ainsi, pour le calcul d'un pas de temps, chaque opération unitaire peut être exécutée en concurrence et une mise à jour globale est initiée à chaque fin de pas de temps. Toutefois, cette approche ne doit être utilisée qu'avec parcimonie et dans des cas bien précis où les modèles des opérations unitaires ont des constantes de temps grandes par rapport au pas de temps de la simulation. Dans le cas contraire, des problèmes de stabilité numérique et de convergence seront levés. D'après ce constat, cette méthode d'intégration n'est pas utilisée dans les environnements commerciaux de simulations.

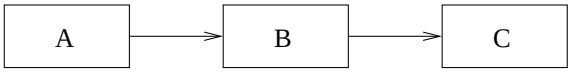


FIG. 5.1 – Un flowsheet simple acyclique.

Dans [95], l'auteur présente la simulation dynamique d'un procédé acyclique où chaque opération unitaire est exécutée en parallèle. Suivant cette approche, l'application définit un parallélisme de type *pipelining*. Dans le *flowsheet* simple de la figure 5.1, dès que l'opération unitaire finit l'exécution de son premier pas de temps, elle peut directement commencer l'exécution du

deuxième (tous ses flux en entrée sont définis). Alors que le second pas de temps de A est exécuté, l'exécution du premier pas de temps pour le module B peut être réalisée en concurrence. De la même manière, lors de l'exécution du troisième pas de temps de A, B exécute son deuxième et C le premier. Cette procédure est représentée sur la figure 5.2 où le modèle de coût associé à l'exécution pondère les tâches avec un même poids et les temps de communication sont considérés nuls. Ainsi, à la troisième itération, il existe un parallélisme total entre les trois modules (A, B et C) composant le *flowsheet* (ce temps correspond au remplissage du pipe).

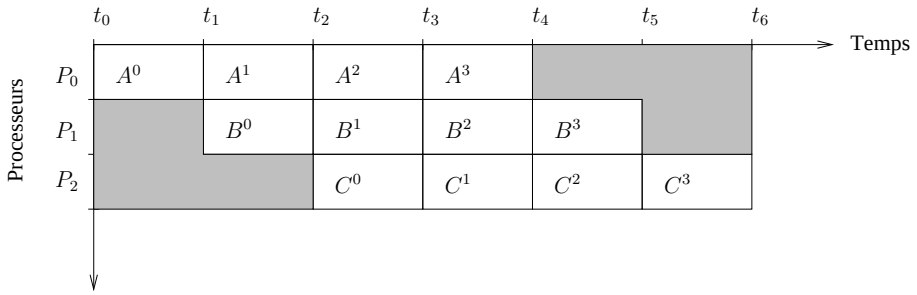


FIG. 5.2 – Diagramme de Gantt représentant la simulation dynamique (donc itérative) du *flowsheet* simple de la figure 5.1. Les temps d'exécution des blocs sont considérés identiques et les temps de communication des données sont nuls.

Si la charge de travail de chaque opération unitaire est sensiblement identique, l'efficacité atteignable est asymptotiquement de 100%. Dans le cas où le *flowsheet* est cyclique, il devient possible d'agglomérer les opérations unitaires constituant le cycle sous la forme d'un unique *super*-module et de reformuler une topologie acyclique. Cependant, la charge de travail entre les modules est alors déséquilibrée. En effet, considérons toujours le même modèle de coût (temps d'exécution des tâches égal à 10 et temps de communication nul), le super-module nécessite alors un temps d'exécution n fois supérieur aux autres où n est le nombre de modules unitaires composant le super-module. Ce déséquilibre de charge entraîne une chute sensible de l'efficacité due en partie à la nécessité d'introduire des périodes de synchronisation : les modules unitaires produisent trop rapidement les données par rapport au super-module les consommant et doivent alors être temporisés.

Cette technique est également utilisée par les auteurs de [105]. Dans leurs travaux, l'accent est mis sur l'intégration de trois sous-procédés couplés au moyen de la bibliothèque d'envoi de messages PVM. Le procédé simulé est de type *batch* (procédé discontinu) dont la transformation de matière s'effectue par phases successives séparées (on parle de stockage tampon de produits intermédiaires entre les différentes étapes actives). Ainsi, le procédé est naturellement divisé aux points définissant les sous-procédés distincts. Ces sous-procédés peuvent contenir des recyclages et sont représentés par des super-modules. La topologie engendrée au niveau d'abstraction super-modules est acyclique. Le facteur d'accélération obtenu est de $2,08$ sur trois machines.

5.2.2 Approches orientées équations : solveurs parallèles

Les travaux traitant de la parallélisation des problèmes en algèbre linéaire sont nombreux. Les simulations numériques du CAPE bénéficient de cette recherche et les rejoignent lorsqu'une approche orientée équations est utilisée.

Si l'on revient à la formulation des méthodes de simulation vues dans le chapitre 2, la simulation statique peut être considéré comme un sous-problème de la simulation dynamique. Les deux modèles sont composés de systèmes d'équations algébriques non-linéaires et le mode dynamique comporte en plus des équations différentielles (DAE pour *Differential Algebraic Equations*). Les systèmes DAE pour la simulation de procédés sont généralement raides (constantes de temps largement différentes) et des méthodes d'intégration implicite (*e.g.*, méthodes de type BDF [23] - *backward differentiation formula*) doivent être utilisées pour leur résolution. Chaque étape de la résolution de ce système implique de résoudre un système d'équations algébriques non-linéaires. Ces problèmes sont généralement traités *via* l'utilisation de méthodes de type Newton. Pour chaque pas de l'itération, les systèmes d'équations linéaires doivent être résolus. Ces systèmes n'exposent aucune propriété permettant de faciliter leur traitement : les matrices de coefficients sont creuses, asymétriques et n'ont pas de diagonales dominantes.

Le temps requis pour résoudre les systèmes d'équations linéaires domine largement le temps global de résolution, la parallélisation des traitements à ce niveau est donc privilégiée. Historiquement, les architectures visées furent les machines vectorielles où *a priori* les opérations vectorielles sont optimisées. Toutefois, les matrices issues de la simulation de procédés sont largement creuses alors que les architectures vectorielles nécessitent une représentation dense pour réaliser les calculs efficacement. Deux techniques permettent de résoudre de tels systèmes.

- L'emploi de méthodes itératives (*e.g.*, GMRES [100]) couplées à un préconditionneur est remis en question dans le contexte de la simulation des procédés [31].
- Les méthodes directes sont généralement employées dans le domaine du CAPE où un système de la forme $Ax = b$ est à résoudre.

Sur une architecture vectorielle, la méthode frontale est couramment employée [126]. Basée sur une factorisation LU, cette méthode décompose la matrice A initiale en sous-systèmes denses sur lesquels la bibliothèque d'algèbre linéaire BLAS peut être appliquée. Ses performances dépendent fortement de l'ordonnancement des lignes et colonnes de la matrice A [125]. Cependant, cette méthode n'expose pas (ou peu) de parallélisme. Pour remédier à ce problème, l'approche multifrontale est employée [77, 102].

5.2.3 Approche modulaire simultanée

Dans le cas des simulations statiques, un fort degré de parallélisme peut être obtenu par la coupe de tous les flux, ce qui permet d'exécuter potentiellement chaque opération unitaire en parallèle. Le système d'équations des flux est alors résolu globalement tout en conservant une méthode d'intégration implicite. Initialement énoncée par Chen *et al.* dans [28], seuls les travaux présentés dans [116] font état d'une telle technique. Néanmoins, cette approche peut être lente

malgré le parallélisme généré. En effet, même si toutes les opérations unitaires sont exécutées en parallèle, la validation du critère de convergence nécessite un grand nombre d'itérations ainsi que de nombreuses communications de type « *all to one* » à destination du solveur global. De plus, il est souvent nécessaire de modifier les codes afin que chaque opération unitaire fournisse les dérivées partielles des variables de sorties en fonction de toutes les variables d'entrée afin de construire la jacobienne du système nécessaire au solveur.

Dans le cas de simulations dynamiques, l'approche modulaire simultanée est également nommée **approche d'intégration modulaire indépendante**. Un axe de recherche important [53, 72] est lié au développement d'algorithmes efficaces de coordination permettant aux modules de disposer de leurs entrées et d'intégrer les équations de leur modèle sur un intervalle de temps fixé. Le coordinateur de l'application (PME) fixe un intervalle de temps commun où un solveur itère jusqu'à l'obtention de la convergence des contributions de chaque opération unitaire. Une fois la convergence obtenue, le calcul de la fenêtre de temps suivante peut être réalisé.

L'accélération et le passage à l'échelle d'une telle approche dépendent fortement du choix de partitionnement des modules qui doit être choisi de telle sorte à minimiser les interactions entre les sous-systèmes et à équilibrer leur charge de calcul. De plus, l'algorithme de coordination ne doit pas engendrer un échange de données excessif. Ceci présuppose que la convergence de la méthode itérative de coordination soit atteinte avec un nombre minimal d'itérations. Ainsi, des méthodes itératives de type *waveform* peuvent être employées [74]. Afin d'extraire un parallélisme suffisant, une méthode de relaxation de type Jacobi est utilisée [19]. Selon ce schéma itératif, chaque sous-système est intégré indépendamment et en parallèle sur chaque pas de temps. Toutefois, cette méthode expose un taux de convergence lent [11] qui peut être amélioré par un surcoût à l'exécution [47] lié à de nombreuses communications. Ainsi, les auteurs de [1] proposent un algorithme de coordination pour l'implantation parallèle d'une approche d'intégration modulaire basée sur une relaxation itérative de type Jacobi. Les variables d'entrée sont approximées par une interpolation par splines cubiques. Selon leurs travaux, seules les communications à la fin de l'intervalle de temps sont nécessaires : les exécutions de chaque module pour la fenêtre de temps sont indépendants.

Nous venons d'étudier les différentes méthodes étudiées dans le domaine du CAPE permettant de dégager un degré de parallélisme plus ou moins important, selon les approches de résolution. Cette première section s'est donc principalement basée sur l'étude des techniques algorithmiques de parallélisation de codes. La tendance actuelle dans le domaine de l'ingénierie des procédés assistée par ordinateur consiste à employer une approche de programmation modulaire par composant favorisée par la spécification du standard CAPE-OPEN. Dans la section suivante, nous recensons et étudions quelques environnements parallèles de couplage de codes dont les concepts développés peuvent être appliqués aux environnements de simulation des procédés.

5.3 Environnements d'exécution parallèle de couplage de codes

La conception d'une application de simulation numérique peut faire interagir deux types de composants : séquentiel et parallèle.

5.3.1 Composant séquentiel

Typiquement, les environnements de type RPC tels que CORBA [82], DCOM [21] ou bien encore JAVA RMI [106], offrent un modèle à composants séquentiels. Selon cette approche, chaque composant est associé à un espace d'adressage propre et les communications sont réalisées à travers un appel de méthode dont l'exécution est séquentielle. Afin de disposer d'un certain parallélisme, il est nécessaire d'utiliser des invocations de méthodes asynchrones. Abordée en section 3.3.3, cette technique est complexe à implanter.

Basé sur le moteur exécutif KAAPI [66] et sur le modèle de programmation CORBA, HOMA [54] exploite automatiquement, le parallélisme entre plusieurs invocations de méthodes dont la sémantique est synchrone. Pour cela, l'analyse du contrat IDL à travers la direction des communications (*in*, *out* et *inout*) ainsi que du code applicatif client permet de détecter à l'exécution les invocations concurrentes. De plus, il implante une stratégie de *communication par nécessité* [43] où seuls les échanges de données nécessaires entre les serveurs sont réalisés (*i.e.*, sans l'intervention du client). Selon cette approche, le grain d'un composant peut être, soit un code élémentaire, soit un code complet de simulation numérique. Bien qu'intéressante, cette approche reste trop proche du modèle de programmation CORBA où le flot de contrôle est centralisé sur l'application de couplage.

D'autres environnements visent à coupler des simulateurs complets hétérogènes. CHEOPS [101] et OPERA [97] ont été développés dans cette optique.

CHEOPS a pour but de coupler des environnements de simulations dynamiques du génie des procédés. Pour cela, il préconise une approche modulaire séquentielle. Cependant, afin de dégager un certain niveau de parallélisme, cet environnement utilise une méthode d'intégration explicite [72] qui peut présenter des instabilités numériques.

OPERA est une projet européen visant à concevoir une plate-forme de couplage de simulateurs hétérogènes temps réel. Le critère temps réel a conduit à l'utilisation de machines parallèles. Pour cela, le concept de « *lookahead* » [27] est implanté. Il impose que chaque simulateur définisse un horizon de temps pour lequel aucun changement d'états n'est possible. La fin d'un pas de temps entraîne une mise à jour globale du système : les auteurs parlent d'échanges de données orientés temps. Selon cette méthode, toutes les données transitent par l'application de couplage pour être retransmises aux environnements distribués. Une autre méthode d'échanges de données, orientée signal, permet la communication directe entre les simulateurs sans intervention de l'application de couplage. Les précédences sont définies par le *plan d'exécution* calculé au moyen de l'application Microsoft MSProject. Bien qu'intéressant, ce mode de communication reste limité par la vue dont dispose OPERA. En effet, la synchronisation liée au mouvement de données bloque l'exécution de l'ensemble du simulateur alors que certaines opérations sont potentiellement exécutables. De plus, l'utilisation du logiciel MSProject n'est pas une solution viable pour répondre aux besoins d'ordonnancement.

5.3.2 Composant parallèle

Au même titre qu'un composant encapsule un code séquentiel, certains travaux de recherche s'orientent actuellement sur la définition de composants parallèles où la réalisation d'un appel de méthode fait intervenir un code parallèle. Les environnements PARDIS [69], PaCO++ [96] et la spécification *Data Parallel CORBA* [83] de l'OMG visent à proposer la notion d'objets parallèles.

Ces trois approches se basent sur la spécification CORBA. Alors que PARDIS et *Data Parallel CORBA* introduisent une modification de la norme (peu d'ORB supportent alors ces modifications), PaCO++ respecte les spécifications de l'OMG. Ces environnements définissent principalement des schémas de communications parallèles entre composants. Pour cela, ils identifient les données impliquées dans ces communications et implantent des schémas de redistribution entre les codes parallèles. Malgré les avantages de ces environnements pour l'exécution de codes parallèles, la séparation entre sémantiques de communication et de contrôle n'est pas traitée : les flots de contrôle et de données restent centralisés sur le client.

Ces environnements sont intéressants si les composants d'une application de couplage de codes sont parallèles. Ces environnements auront plus d'impact lorsque les opérations unitaires seront développées dans une logique de programmation parallèle de type parallélisme de données où, par exemple, les propriétés physique indépendantes serait calculées en parallèle.

5.4 Bilan

L'ingénierie des procédés assistée par ordinateur s'est très vite intéressée aux techniques de programmation parallèle. Selon l'approche de résolution, le parallélisme exploitable est plus ou moins important. Une approche modulaire séquentielle offre un parallélisme restreint correspondant à la topologie du schéma de procédé. L'approche orientée équations est la plus étudiée et bénéficie des travaux sur la parallélisation des traitements en algèbre linéaire. L'approche hybride (modulaire simultanée) peut dans certains cas engendrer un haut degré de parallélisme où chaque opération unitaire peut potentiellement être exécutée en parallèle. Cependant, pour être efficace, elle nécessite que chaque module calcule les dérivées des paramètres de sortie en fonction de tous les paramètres d'entrée. En pratique, peu de modules offrent ces informations.

Lors de l'exécution de simulations dynamiques, les méthodes d'intégrations numériques extrapolent les données futures pour limiter les dépendances entre les modules. Ces approches nécessitent une bonne répartition de la charge afin d'être efficaces. Parfois, des problèmes de stabilité numérique se pose lorsque le problème est raide.

Le tableau 5.1 récapitule l'origine du parallélisme des trois approches de résolution suivant le mode d'exécution (statique ou dynamique).

Malgré une recherche active pour la parallélisation des simulations de procédés, peu d'applications industrielles ont vu le jour. En effet, la simulation industrielle est fortement dominée par l'approche modulaire séquentielle qui expose le moins de parallélisme. L'objectif de cette thèse est de concevoir un environnement d'exécution parallèle pour des cas applicatifs métier dont les composants sont implantés en respectant la norme CAPE-OPEN. L'approche de résolution est donc dictée par le PME et les interfaces des opérations unitaires à disposition. Pour les simulations statiques, il s'agit d'une approche modulaire séquentielle alors que pour les simulations dynamiques,

	SMA	EO	Hybride
Statique	Parallélisme structurel	Résolution d'un système linéaire. Approches frontale et multifrontale	Coupe de tous les flux + solveur centralisé
Dynamique	Parallélisme structurel + méthode d'intégration explicite		Méthodes d'intégration et d'extrapolation

TAB. 5.1 – Récapitulatif des méthodes de résolution permettant une exécution parallèle pour les différentes approches de résolution (modulaire séquentielle (SMA), orientée équation (EO) et modulaire simultanée (hybride)) du CAPE.

une approche originale est employée (*c.f.* section 6.2).

En marge de l'algorithmique parallèle, une recherche orientée *middleware* est menée en vue de proposer des environnements parallèles performants de couplage de codes selon un paradigme de programmation RPC.

Le prototype HOMA est développé dans le but de détecter les invocations de méthodes concurrentes pour des applications de couplage de codes au moyen de l'analyse des directions des paramètres du langage IDL. La sémantique de contrôle reste proche du modèle de programmation CORBA qui est centralisé sur le client. Au contraire, il propose un mécanisme permettant de ne générer que les mouvements de données nécessaires entre les serveurs. Dans le contexte CAPE-OPEN, le PME est l'application de couplage qui invoque des méthodes sur chaque composant métier (UO / solveur / Thermo). Toutefois, les spécifications du standard masquent le flot de données à travers le composant utilitaire *Material Object* ce qui rend l'utilisation de HOMA impossible en l'état.

OPERA permet de décentraliser une partie du flot de contrôle dont les dépendances sont dictées au moyen de MSProject. Cependant, l'unité élémentaire est l'environnement de simulation. Le grain de parallélisme est important ce qui limite le parallélisme potentiel de l'exécution.

Les environnements de couplage de codes parallèles (PARDIS, PaCO++ et *Data Parallel Corba*) ne sont pas exploitables car ils nécessitent des composants parallèles. Les composants métier dont nous disposons ne sont que séquentiels. Dans l'avenir, il sera bon d'étudier l'impact d'une telle approche sur les simulations du CAPE.

Afin d'obtenir un gain à la distribution des simulations CAPE, l'approche de résolution qui semble la mieux adaptée est orientée équations. Les avancées touchent tous les domaines liés à la simulation numérique ce qui mène à une recherche active.

L'approche modulaire séquentielle, quant à elle, ne propose qu'un faible degré de parallélisme qui est dicté par la topologie du schéma de procédé à simuler. Toutefois, c'est la méthode la plus utilisée dans le domaine.

L'approche modulaire simultanée propose un bon compromis. Elle permet d'exécuter toutes les opérations unitaires en parallèle. Toutefois, afin d'être performante, elle nécessite que chaque opération puisse calculer les dérivées partielles des variables de sorties en fonction de toutes les

variables d'entrée afin de construire la jacobienne du système.

En pratique, le standard CAPE-OPEN propose *simplement* des spécifications d'interfaces qui permettent d'implanter ces approches de résolution. Toutefois, le domaine de la simulation des procédés est largement dominé par l'approche modulaire séquentielle. Nos travaux portant, non pas sur la conception d'algorithmes parallèles numériques pour la simulation des procédés, mais sur la conception d'un environnement d'exécution distribuée, nous sommes contraints par l'utilisation d'une approche et d'un environnement de simulation. Le parallélisme est donc principalement de type structurel.

A l'instar du prototype HOMA, notre approche vise à automatiser et à masquer la complexité de distribution des calculs sur les ressources de calcul. Pour cela, nous nous appuyons sur des algorithmes d'ordonnancement qui permettent de calculer un placement des composants de l'application sur les processeurs.

La partie suivante définit l'environnement d'exécution parallèle implanté pour la simulation des procédés dont les composants sont au standard CAPE-OPEN. Il se base sur le moteur exécutif INDISS-RT développé par RSI.

II

Un environnement d'exécution distribué pour le CAPE

Simulation des procédés sous INDISS 6

6.1 Introduction

INDISS (*INDustrial and Integrated Simulation Software*), développé par RSI, tient une place majeure dans l'industrie pétrochimique *via* des outils de conception permettant la résolution de simulations dynamiques de procédés continus. RSI est spécialisé dans la commercialisation de simulateurs[1] préconfigurés. Historiquement, ces simulateurs étaient majoritairement destinés à l'entraînement des techniciens sur des unités de production. Ainsi, le facteur temps réel était privilégié au détriment de la précision numérique. Le partenariat IFP / Total à travers le projet TINA entraîne une modification des spécificités d'INDISS. A présent, INDISS doit s'ouvrir aux problèmes de simulations et de conceptions où le facteur précision numérique doit prévaloir sur l'aspect temps réel. De plus, selon ce partenariat, INDISS nécessite de répondre au standard CAPE-OPEN. Depuis 2003, RSI développe une branche parallèle à son environnement de simulation INDISS visant à répondre à ce cahier des charges.

Le premier objectif de cette thèse est d'étudier la faisabilité d'une approche de calcul distribué pour les simulations CAPE dont les composants répondent au standard CAPE-OPEN. INDISS, étant compatible avec le standard CAPE-OPEN et implémentant les futures interfaces CAPE-OPEN pour le dynamique, s'est imposé très vite comme l'environnement le plus approprié à une extension pour gérer un mode d'exécution distribuée. De plus, de par la filiation avec l'IFP, son code nous était ouvert facilitant d'autant plus l'intégration de notre approche.

INDISS se compose de deux entités distinctes : INDISS et INDISS-RT.

- INDISS se définit principalement par une interface homme-machine gérant les interactions avec l'utilisateur. Il permet de construire un schéma de procédé à partir de bibliothèques d'opérations unitaires qu'elles soient propriétaires ou au standard CAPE-OPEN. Comme tout environnement de conception et de simulation de procédés, il offre de nombreuses méthodes de configuration telles que le choix du serveur pour le calcul des propriétés thermodynamiques ou bien encore l'ajustement des paramètres des opérations unitaires (*e.g.* hauteur et nombre de plateaux d'une colonne de distillation).
- INDISS-RT est le moteur exécutif implantant l'approche de résolution. Il comprend également un module de gestion de graphe implantant, entre autre, un algorithme de coupe de

[1] Suivant ce contexte, un simulateur fait référence à schéma de procédé ou *flowsheet*

flux permettant de rendre le schéma de procédé acyclique, propriété indispensable à l'approche de résolution comme vu dans le chapitre 2 page 33.

Ce chapitre introduit l'approche de résolution adoptée par RSI pour la résolution des problèmes de simulation des procédés. Le fait qu'INDISS ait été initialement pensé pour répondre au besoin de formation des techniciens a fortement dirigé la conception de son environnement d'exécution pour les simulations dynamiques, au détriment des méthodes liées à la résolution statique. De ce fait, l'exécution statique d'un procédé selon un mode statique nécessite l'implantation d'une approche de résolution spécifique. Toutefois, à partir d'une des étapes définissant une simulation dynamique, il est possible de résoudre une simulation statique dont le schéma de procédé est acyclique selon une approche modulaire séquentielle. Enfin, nous détaillons les contraintes de dépendance entre les entités d'exécution permettant de dégager un premier niveau de parallélisme directement exploitable.

6.2 La résolution dynamique

La résolution dynamique d'un procédé selon INDISS-RT suit une variante de l'approche modulaire simultanée présentée en section 2.2.4.3 page 36. Formellement, une approche modulaire simultanée se caractérise par l'imbrication de deux niveaux de calcul. Le *niveau module*, résout les modèles des opérations unitaires selon les paramètres fixés par le niveau (*flowsheet*). Généralement, chaque module est un ensemble d'opérations unitaires définissant un groupe irréductible (*c.f.* page 33) donc appartenant à un recyclage. Afin de proposer un nouveau jeu de valeurs aux modules, le niveau *flowsheet* requiert le calcul de la matrice jacobienne associée à l'ensemble des modules. Cette matrice est caractérisée par les sensibilités de l'ensemble de valeurs thermodynamiques caractérisant complètement les flux (*c.f.* théorème de Duhem pages 31). Ainsi, le calcul de l'erreur se base sur un système complètement défini thermodynamiquement.

L'approche adoptée par RSI détermine l'erreur uniquement sur le respect de la loi de conservation de la masse. De plus, le niveau *flowsheet* ne considère pas les équations de recyclage et de spécification mais les équations de l'ensemble des flux. Lorsque le critère de convergence est atteint, seuls les débits et pressions sur l'ensemble du *flowsheet* sont définis : le système n'est pas thermodynamiquement défini. Une exécution séquentielle est alors nécessaire à la mise à jour des bilans énergétiques sur l'ensemble du *flowsheet*.

L'exécution dynamique dans INDISS-RT suit un schéma séquentiel itératif de résolution en trois étapes dont la séquence des opérations est présentée en figure 6.1. RSI étant actif dans la définition du standard CAPE-OPEN pour les simulations dynamiques, chaque étape correspond à l'appel des méthodes CAPE-OPEN correspondantes sur l'ensemble des opérations unitaires du schéma de procédé. Ainsi, les étapes *FirstCompute*, *NetworkCompute* et *LastCompute* sont respectivement liées aux méthodes `StartTimeStep`, `ResolveFlowPressure` et `Calculate`[2] des composants de type opération unitaire. Malgré la correspondance explicite entre les étapes de résolution définies par INDISS-RT et les méthodes exposées par les opérations unitaires CAPE-OPEN, le standard ne préconise pas d'approche de résolution particulière : il définit des méthodes

[2] Ces méthodes ont été présentées en section 2.3.4.2 page 43

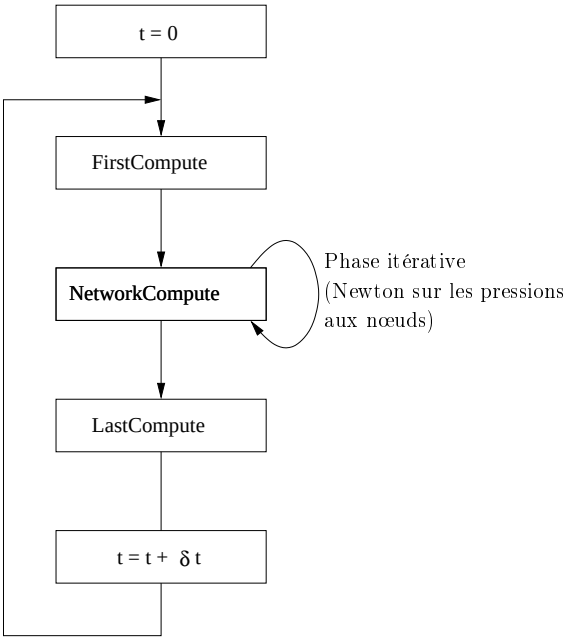


FIG. 6.1 – Enchaînement des étapes de la résolution dynamique dans INDISS-RT

sur chaque composant offrant des fonctionnalités spécifiques. Cette section vise ainsi à caractériser l'utilisation des composants CAPE-OPEN couplée à l'approche de résolution développée par RSI.

6.2.1 FirstCompute

La figure 6.2 présente l'exécution de l'étape *FirstCompute* sur un cas test simple. Cette première étape permet de prendre en considération les changements de pression sur les limites de réseau ou, par exemple, de traiter l'ouverture d'une vanne. Elle correspond à une phase de prise en compte du caractère interactif de la simulation et d'initialisation du réseau en fonction de ces nouveaux paramètres. Par exemple, la pression d'entrée P_{in} peut être modifiée ou bien encore le nouveau coefficient de débit d'une vanne (Cv) relatif à son pourcentage d'ouverture est calculé.

La réalisation de cette étape correspond à l'appel des méthodes CAPE-OPEN `StartTime-Step` sur l'ensemble des opérations unitaires du schéma de procédé. L'appel est ordonné selon les priorités affectées par INDISS-RT sur la représentation acyclique du *flowsheet* obtenues après la coupe des flux de recyclage.

6.2.2 NetworkCompute

La figure 6.3 présente la méthode *NetworkCompute* sur le même cas test. Cette étape vise à répondre au critère de convergence par le calcul d'erreur sur les bilans matières en chaque nœud du réseau. Rappelons la définition fournie en 2.3.4.2 page 43 :

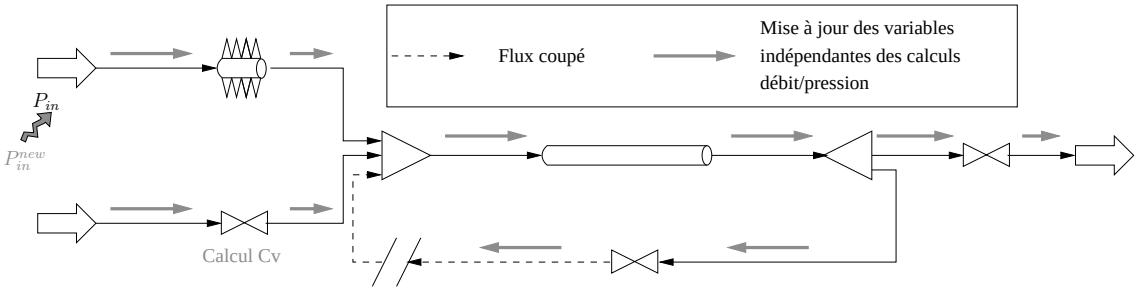


FIG. 6.2 – 1ère étape (*FirstCompute*) de la résolution dynamique dans INDISS-RT permettant d'initialiser le réseau selon les débits et pressions tout en tenant compte des interactions utilisateur.

Définition 3 *Un* **nœud** i *est une opération unitaire caractérisée par une accumulation. Ainsi, d'après la loi de conservation de la masse :*

$$D_i^{in} = D_i^{out} + Accumulation_i \tag{6.1}$$

avec,

- D_i^{in} *(kg/s) étant la somme des débits entrant au nœud* i
- D_i^{out} *(kg/s) étant la somme des débits sortant au nœud* i
- $Accumulation_i$ *(kg/s) étant l'accumulation au nœud* i *(valeur positive ou négative)*

□

L'accumulation représente la variation de la charge au cours du temps, elle est calculée au moyen de la compressibilité C (en kg). Ainsi, l'erreur au nœud i au pas de temps t se quantifie de la sorte :

$$E_i = (D_i^{\text{out}} - D_i^{\text{in}}) \times dt + \mathrm{C}_i - C_i^{\text{init}} \tag{6.2}$$

Le calcul des débits d'entrée/sortie à chaque nœud est effectué par les opérations unitaires de type arc (*i.e.* équipements sans accumulation) en amont et en aval. En fonction des pressions en entrée/sortie, chaque arc calcule le débit le traversant. L'alternance nœud/arc est donc indispensable au calcul des erreurs sur les bilans massiques aux nœuds. Les nœuds calculent leur compressibilité en fonction de la pression en entrée/sortie.

En fonction des débits et des compressibilités, le solveur peut alors calculer l'erreur sur le bilan matière en chaque nœud. La figure 6.3 présente un tel schéma de résolution.

Les valeurs numériques compressibilité et débit sont calculées sur l'invocation de la méthode CAPE-OPEN `ResolveFlowPressure` exposée par chaque opération unitaire. Un arc calcule le débit le traversant ainsi que sa dérivée par rapport aux pressions d'entrée et de sortie. En outre, un nœud calcule sa compressibilité ainsi que sa dérivée par rapport à la pression.

Pour calculer l'erreur globale ($\sum_{i \in \text{nœuds}} E_i^2$), le solveur appelle sur chaque arc la méthode CAPE-OPEN `GetFlowAndDerivatives` et sur chaque nœud `GetCompressibility-AndDerivative`. Si l'erreur calculée n'est pas en deçà d'un seuil fixé, le solveur (généralement

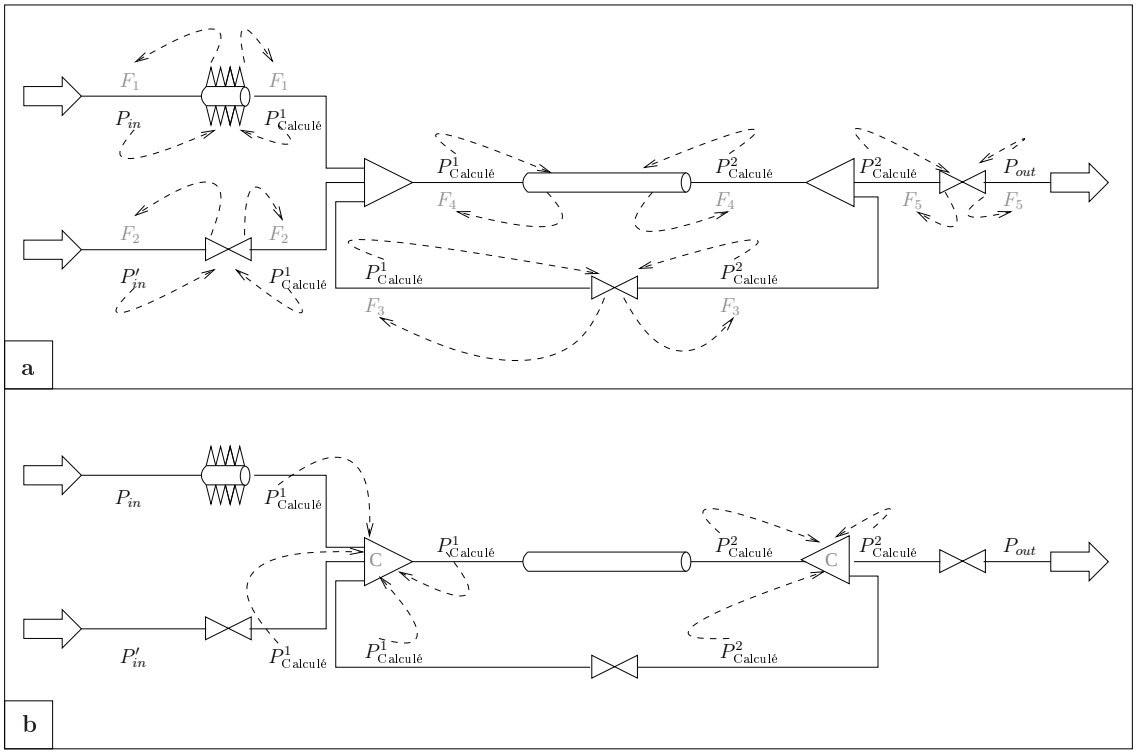


FIG. 6.3 – 2ème étape (*NetworkCompute*) de la résolution dynamique INDISS-RT permettant de calculer l'erreur sur les bilans massiques à chaque nœud du réseau. A partir des pressions fixées par le solveur, les opérations unitaires de type arc calculent le débit les traversant et mettent à jour leurs *Material Object* connectés (a) alors que celles de type nœud calculent leur compressibilité (valeur interne à l'opération) (b).

de type Newton) propose une nouvelle pression ($P^{x}_{\text{Calculated}}$), calculée au moyen des dérivées, en chaque nœud et itère jusqu'à ce que l'erreur satisfasse la contrainte de convergence.

En pratique, le schéma d'exécution de l'étape *NetworkCompute* se décompose en deux sous-traitements. Dans un premier temps, chaque arc calcule son débit ainsi que les dérivées relatives aux pressions d'entrée/sortie. Puis, dans un second temps, chaque nœud calcule sa compressibilité et la dérivée relative à la pression d'entrée/sortie. Le schéma d'exécution de l'étape *NetworkCompute* implanté dans INDISS-RT suit alors le diagramme présenté en figure 6.4.

6.2.3 LastCompute

La figure 6.5 présente la dernière étape de la résolution dynamique pour un pas de temps t. Cette dernière étape permet de calculer le bilan énergétique selon une approche séquentielle. Cette étape fixe les paramètres thermodynamiques de chaque *Material Object*. A l'instar de *FirstCompute*, la résolution de cette étape est ordonnée et requiert l'appel de la méthode CAPE-OPEN

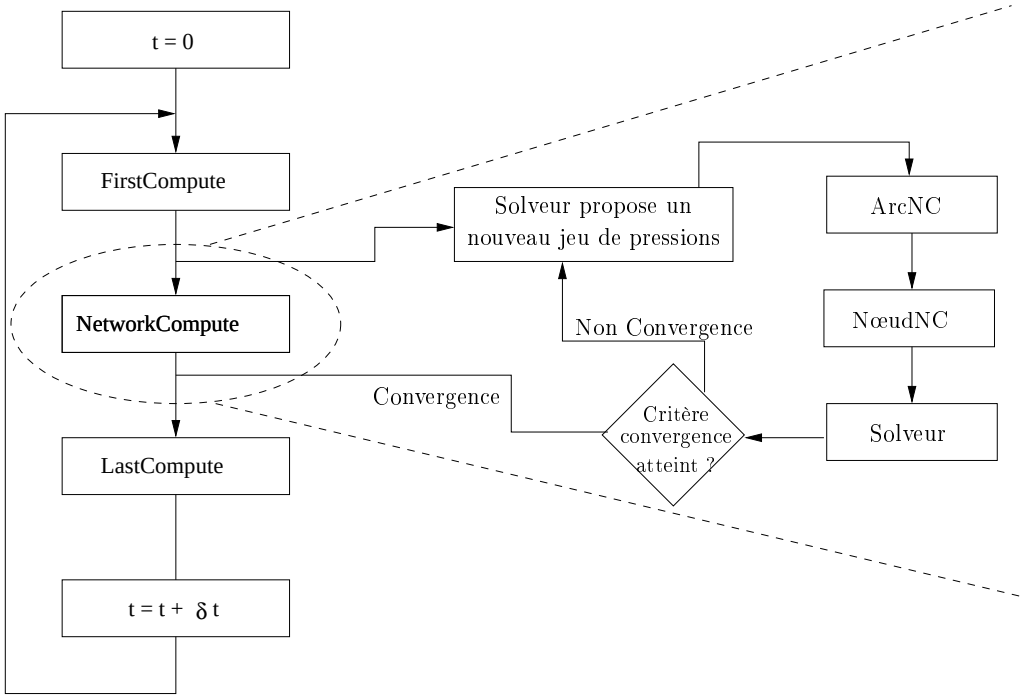


FIG. 6.4 – Décomposition de l'étape *NetworkCompute* : 1/ les méthodes CAPE-OPEN `ResolveFlowPressure` et `GetFlowAndDerivatives` sont appelées sur chacun des arcs du schéma de procédé (*ArcNC*) ; 2/ `ResolveFlowPressure` et `GetCompressibilityAndDerivative` sont appelées sur chacun des nœuds (*NœudNC*) ; 3/ Le solveur vérifie si le critère de convergence est atteint et itère dans le cas contraire.

`Calculate` sur chaque opération unitaire.

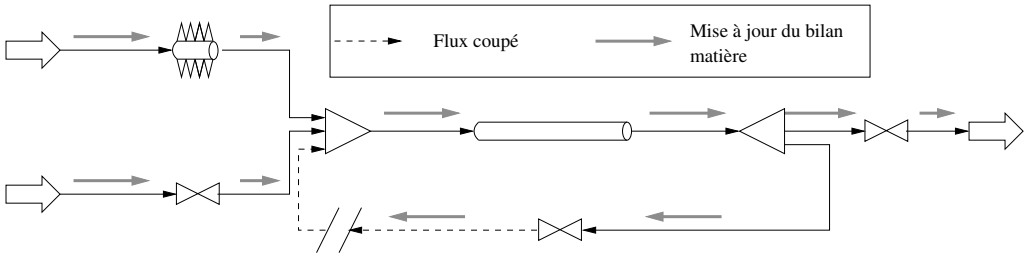


FIG. 6.5 – Dernière étape (*LastCompute*) de la résolution dynamique INDISS réalisant le bilan énergétique. En fonction de ses flux d'entrée, chaque opération unitaire calcule ses flux de sortie ainsi que ses paramètres internes.

6.3 La résolution statique

La résolution statique considérée dans nos travaux repose sur une approche modulaire séquentielle. INDISS-RT ne gère pas cette approche pour laquelle un solveur global itère sur les variables caractérisant le flux coupé jusqu'à ce que le critère de convergence soit atteint.

INDISS-RT considère la résolution statique comme l'exécution d'un unique pas de temps d'une simulation dynamique. Toutefois, cette approche n'est guère satisfaisante d'un point de vue numérique : le système n'est pas globalement défini. Pour que les flux soient complètement définis, les masses initiales totales de chaque constituant ainsi que deux variables indépendantes thermodynamiquement doivent être spécifiées. Or, la convergence des recyclages est assurée par la minimisation de l'erreur sur les bilans massiques totaux (étape *NetworkCompute*), et une résolution modulaire séquentielle est appliquée sur le graphe acyclique obtenu après coupe des recyclages. Selon cette méthodologie, les opérations unitaires ayant en entrée des flux coupés utilisent des grandeurs inadéquates (du pas de temps $t-1$) avec les débits/pressions calculés par le solveur (du pas de temps t). En effet, seules les débits/pressions correspondent au pas de temps t, les autres variables thermodynamiques (*e.g.* enthalpie, débit molaire partiel de chaque constituant) sont les valeurs calculées au pas de temps $t-1$.

C'est pourquoi, RSI a débuté l'intégration d'une approche modulaire séquentielle dans leur environnement de simulation INDISS-RT. Cependant, à l'époque où les expériences ont été menées, cette méthode de résolution n'était pas opérationnelle. Nous avons donc implémenté un substitut simple de cette approche.

Dans le cas où le schéma de procédé est sans recyclage, l'étape *LastCompute*, où la méthode CAPE-OPEN `Calculate` est invoquée sur l'ensemble des opérations unitaires suivant un ordre prédéfini, qualifie une approche modulaire séquentielle. A partir de ce constat, notre premier environnement se base alors sur une telle approche de résolution afin de résoudre une simulation statique. Le schéma de procédé doit alors être sans recyclage.

6.4 Analyse du flot de données du schéma d'exécution

A partir de ces définitions, la première étape nécessaire à une exécution distribuée est de détecter les précédences entre les entités d'exécution ainsi que les dépendances de données qu'elles suscitent. La suite décrit de manière exhaustive toutes les dépendances de données entre chaque opération unitaire.

6.4.1 Les contraintes de précédence de l'exécution

L'analyse des contraintes de précédence se base sur le schéma d'exécution dynamique en trois étapes défini par INDISS-RT et présenté ci-dessus en section 6.2. Pour les simulations statiques, elles sont déduites de la dernière étape *LastCompute* comme décrit dans la section précédente.

Pour décrire les relations de précédence, nous adoptons la notation suivante :

- G^c est le graphe cyclique représentant le schéma de procédé avec recyclages.
- G est le graphe représentant le schéma de procédé après l'application d'un algorithme métier de coupe de flux : ce graphe est acyclique.

- N_k est une opération unitaire de type nœud, c'est-à-dire un équipement caractérisé par une accumulation.
- A_l est une opération unitaire de type arc, c'est-à-dire un équipement sans accumulation.
- BL_m est une opération unitaire de type limite de réseau, c'est-à-dire un équipement (conceptuel) fixant les conditions limites sur lesquelles le procédé agit.
- X_i est une opération unitaire quelconque : $\{X_i\} = \{A_k\} \cup \{N_l\} \cup \{BL_m\}$.
- $T_i^m(t)$ est l'évaluation de la méthode CAPE-OPEN m sur l'opération unitaire X_i au pas de temps t où $m \in \{STS, RFP, C\}$ où STS, RFP et C sont respectivement les trois méthodes CAPE-OPEN `StartTimeStep`, `ResolveFlowPressure` et `Calculate` présentées en section 2.3.4, page 42.
- $T_s(t)$ est la phase où le solveur est appelé. Elle correspond à la validation du critère de convergence et le cas échéant, à fixer un nouveau de pression à chaque *Material Object*.

La formalisation des contraintes de précédence suit une représentation courante où la tâche est l'entité d'exécution indivisible. Ainsi, chaque appel de méthodes CAPE-OPEN sur un composant représente une tâche. L'énumération suivante définit les relations de précédence entre les tâches composant l'exécution d'une simulation dynamique. Le symbole $\prec$ représente une relation d'ordre : $T_i \prec T_j$ signifie que la tâche T_i doit s'exécuter avant la tâche T_j.

1. Sérialisation des invocations de méthodes sur un composant. Les contraintes suivantes définissent les dépendances des états partagés au sein d'un même composant. Les méthodes exposées par un même composant peuvent *communiquer via* leurs attributs, l'ordre des appels doit être conservé afin de maintenir la cohérence de l'exécution. Pour chaque opération unitaire X_i, les tâches T_i^m s'ordonnent suivant le schéma d'exécution des étapes de la section 6.2 :
   (a) $T_i^{STS}(t) \prec T_i^{RFP}(t)$
   (b) $T_i^{RFP}(n) \prec T_i^{RFP}(n+1)$: l'appel de `ResolveFlowPressure` est itératif au sein du calcul d'un même pas de temps (*c.f.* figure 6.4), ainsi l'appel de RFP à l'itération $n+1$ sur un composant X_i ne peut débuter qu'après l'exécution de RFP sur ce même composant à l'itération n pour le même pas de temps t.
   (c) $T_i^{RFP}(n_{last}) \prec T_i^C(t)$: le dernier appel (n_{last}) de RFP (symbolisant que le solveur à valider le critère de convergence) doit être effectué avant de débuter le calcul de la dernière méthode CAPE-OPEN `Calculate` (correspondant à la dernière étape *LastCompute*).
   (d) $T_i^C(t) \prec T_i^{STS}(t+1)$: le calcul du pas de temps $t+1$ pour une opération unitaire X_i ne débute que si la dernière étape (*LastCompute*) pour la date t est finie.
2. Dépendance de données sur la réalisation des étapes *FirstCompute* et *LastCompute* pour chaque composant X_i du schéma de procédé selon les contraintes d'écoulement. Cela revient à utiliser l'approche modulaire séquentielle (*c.f.*, section 2.2.4, page 33) sur le graphe G acyclique.
   (a) $T_i^{STS}(t) \prec T_j^{STS}(t)$ si X_i est un prédécesseur de X_j dans G
   (b) $T_i^C(t) \prec T_j^C(t)$ si X_i précède X_j dans G

3. Dépendance sur le début du calcul du solveur. La première itération du solveur nécessite les compressibilités initiales de chaque opération unitaire de type nœud.
   (a) $T_i^{STS}(t) \prec T_s(t)$ si $X_i \in \{N_l, BL_m\}$
4. Traitement itératif de l'étape *NetworkCompute* décomposée (figure 6.4, page 104). La représentation des précédences suit une notation particulière où le pas de temps t est remplacé par l'itération courante n du solveur.
   (a) Le solveur fixe un nouveau jeu de pression en tout point du schéma de procédé.
   $T_s(n) \prec T_i^{RFP}(n+1)$ si $X_i \in \{A_l, N_k\}$
   (b) Précédence entre les sous-étapes *ArcNC* et *NœudNC*
   $T_i^{RFP}(n) \prec T_j^{RFP}(n)$ si $X_i = A_l$, $X_j = N_k$ et X_i est un successeur ou prédécesseur direct
   (c) Le solveur nécessite les compressibilités et débits ainsi que leurs dérivées relatives à la pression pour chaque opération unitaire de type arc et nœud calculés par la méthode CAPE-OPEN `ResolveFlowPressure`
   $T_i^{RFP}(n) \prec T_s$ si $X_i \in \{A_l, N_k\}$
5. Le solveur a validé le critère de convergence, la dernière étape *LastCompute* peut débuter.
   (a) $T_s(t) \prec T_i^{C}(t)$ si X_i n'a pas de prédécesseur dans G

A partir, de cette spécification exhaustive des contraintes de précédence, la section suivante recense les dépendances de données correspondantes. Il convient de noter que certaines précédences sont redondantes. Par exemple, la contrainte 1b ($T_i^{RFP}(n) \prec T_i^{RFP}(n+1)$) est garantie toujours vrai par les contraintes 4a ($T_s(n) \prec T_i^{RFP}(n+1)$) et 4c ($T_i^{RFP}(n) \prec T_s$). En effet, le solveur nécessite la contribution de chaque itération n afin de proposer le nouveau jeu de pression indispensable au calcul de l'itération $n+1$. Toutefois, elles permettent de qualifier totalement le flot de données attaché au schéma de calcul.

6.4.2 Les dépendances de données

D'après les spécifications du standard CAPE-OPEN, le flot de données est masqué en partie par le concept de *Material Object* attaché à chaque opération unitaire. Cette entité est un composant informatique où seul le contrat IDL spécifiant les méthodes associées est connu. A moins d'intercepter les appels de mise à jour de ce composant en provenance soit d'un composant opération unitaire, soit d'un composant serveur de calculs thermodynamiques, une connaissance *a priori* du flot de données est impossible. Tout composant opération unitaire accède en lecture/écriture aux *Material Object* auxquels il est associé. Malgré cela, les pré/post-conditions issues des spécifications du standard nous permettent de réduire l'ensemble des données à communiquer.

Dans le majorité des cas, il est nécessaire de transférer la totalité des *Material Object* connectés en sortie à une opération unitaire. Ce mouvement de données correspond aux dépendances intra-étapes *FirstCompute* et *LastCompute* où le schéma d'exécution suit le sens physique d'écoulement de la matière dans le procédé. Seule l'étape *NetworkCompute* limite le volume de communication aux simples valeurs débit, compressibilité ainsi que leurs dérivées relatives à la pression.

Contrainte	Données impliquées
(2a) $T_i^{STS}(t) \prec T_j^{STS}(t)$	MO
(1a) $T_i^{STS}(t) \prec T_i^{RFP}(t)$	Etat
(3a) $T_i^{STS}(t) \prec T_s(t)$	Compressibilité Initiale
(4b) $T_i^{RFP}(n) \prec T_j^{RFP}(n)$	Débit
(1b) $T_i^{RFP}(n) \prec T_i^{RFP}(n+1)$	Etat
(4c) $T_i^{RFP}(n) \prec T_s$	Débit / Compressibilité + Dérivées
(4a) $T_s(n) \prec T_i^{RFP}(n+1)$	Pression
(1c) $T_i^{RFP}(n_{last}) \prec T_i^C(t)$	Etat
(5a) $T_s(t) \prec T_i^C(t)$	$\emptyset$
(2b) $T_i^C(t) \prec T_j^C(t)$	MO
(1d) $T_i^C(t) \prec T_i^{STS}(t+1)$	Etat

TAB. 6.1 – Définition du flot de données associé à chaque contrainte de précédence.

La contrainte de précédence 5a n'engendre pas de dépendance de données. En effet, il s'agit d'une contrainte fonctionnelle où le solveur vérifie le critère de convergence. S'il est atteint, toutes les tâches disposent des données nécessaires à la réalisation de la dernière étape *LastCompute* et ceci grâce aux précédences 4b et 4a où chaque flux, et *a fortiori* les flux coupés, dispose des valeurs débit et pression liées au système convergé.

Dans ce qui précède, aucune contrainte n'est liée à la visualisation des résultats. Ainsi, une nouvelle contraintes est nécessaire mettant en jeu la totalité des *Material Object* et des paramètres de sortie de chaque opération unitaire :
soit T_v la tâche de visualisation, alors $T_i^C \prec T_v$ pour toute opération unitaire X_i.

L'ensemble des contraintes de précédence et de dépendance de données forme un graphe de flot de données. La figure 6.6 illustre cette représentation pour le cas test utilisé pour de la description des trois étapes de la résolution INDISS-RT en section 6.2. L'étape *NetworkCompute* est développée sur deux itérations. Chaque étape et sous-étape, définie à partir de la figure 6.4, est identifiée. De plus, une tâche symbolisant le module de visualisation/récolte de résultats est présente.

Le section suivante définit le parallélisme exposé par de telles contraintes.

6.5 Sources de parallélisme

D'après les contraintes de précédence présentées ci-dessus, trois sources de parallélisme existent dans le cas de l'exécution d'une simulation dynamique et une seule est conservée pour l'exécution d'une simulation statique.

La première source, qui est présente dans les deux modes de simulation, est un parallélisme structurel entre les opérations unitaires. Il exploite la topologie du schéma de procédé. La figure 6.7

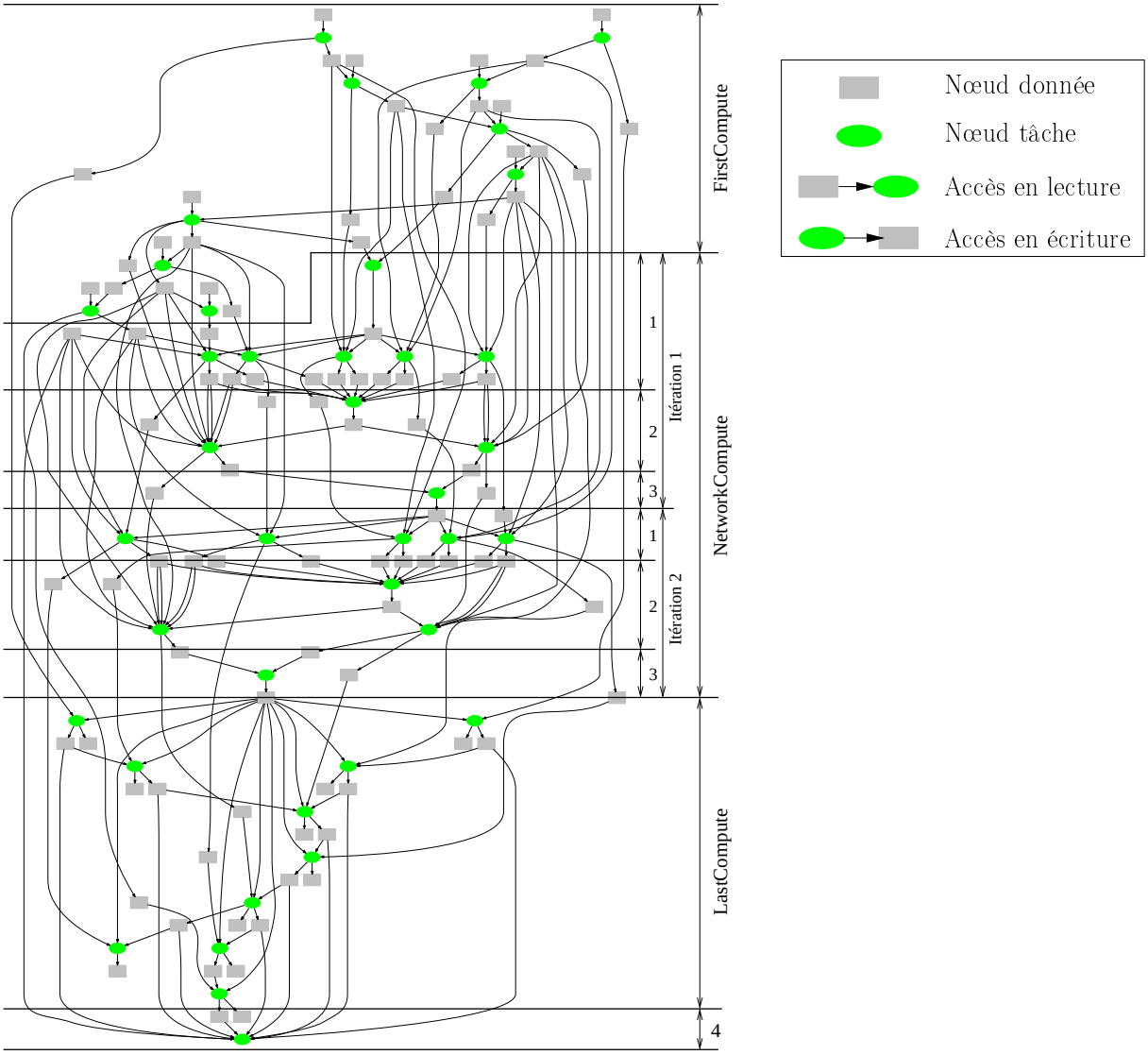


FIG. 6.6 – Graphe de flot de données du cas test exemple utilisé précédemment où l'étape *Network-Compute* est développée sur deux itérations. Cette étape se décompose en ArcNC (1), NodeNC (2), et solveur (3). A la fin de chaque pas de temps, tous les *Material Object* et les paramètres de sortie des UO sont affichés pour visualisation (4).

présente sous la forme d'un graphe de flot de données le parallélisme potentiel pour le cas test exemple utilisé jusqu'à présent. Chaque tâche représente une opération unitaire. Leur regroupement représente un site d'exécution possible et marque ainsi les tâches pouvant être exécutées en concurrence.

Ce parallélisme est l'unique source présente dans le cas de simulations statiques où le graphe de la figure 6.7 caractérise complètement le schéma d'exécution.

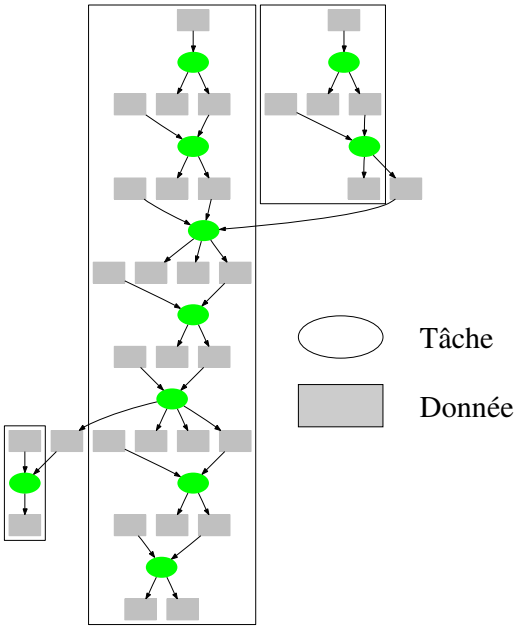


FIG. 6.7 – Parallélisme structurel exposé par la topologie du schéma de procédé.

Pour les simulations dynamiques, les étapes *FirstCompute* et *LastCompute* peuvent bénéficier de ce parallélisme. Une seconde source est également présente et repose sur les propriétés d'exécution des méthodes CAPE-OPEN `ResolveFlowPressure`. Un parallélisme complet existe par leur invocation sur les deux ensembles formés par les opérations unitaires de type arc et de type nœud. Toutefois, en pratique, leur temps d'exécution est faible comparé aux temps de calcul des deux autres méthodes métiers (de l'ordre de 5%) si bien que l'exploitation de ce parallélisme ne peut être qu'inefficace (temps de communications supérieur au temps d'exécution).

Enfin, la dernière source est un parallélisme entre pas de temps. La condition nécessaire au début de l'exécution du pas de temps $t+1$ repose sur le respect de l'ordre d'exécution des invocations de méthode sur un composant. La figure 6.8 présente le graphe de flot de données associées à l'enchaînement des méthodes CAPE-OPEN `Calculate` au pas de temps t et `StartTimeStep` au pas de temps $t+1$.

Le parallélisme exposé est de type *pipeline*. La figure 6.9 présente le gain potentiel sur un graphe simple en considérant les temps d'exécution de chaque tâche et de communication uniformes et de valeur respectives 10 UT (unité de temps) et 1 UT.

L'exécution de simulations itératives engendre généralement des anti-dépendances [26]. Ce sont des *fausses dépendances de données* car il est aisé de les supprimer en considérant comme indépendantes les différentes versions (ou données) des variables. La simulation dynamique expose un schéma d'exécution itérative dans lequel il existe des anti-dépendances sur les *Material Object* entre les étapes *LastCompute* et *FirstCompute*. Par exemple, considérons l'exemple précé-

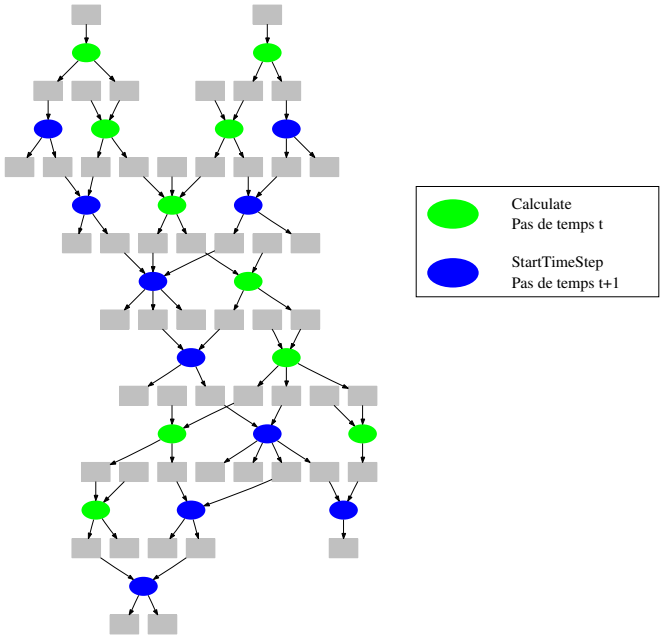


FIG. 6.8 – Graphe de flot de données représentant les dépendances inter pas de temps. Les tâches vertes et bleues représentent respectivement les réalisations des méthodes CAPE-OPEN `Calculate` et `StartTimeStep`.

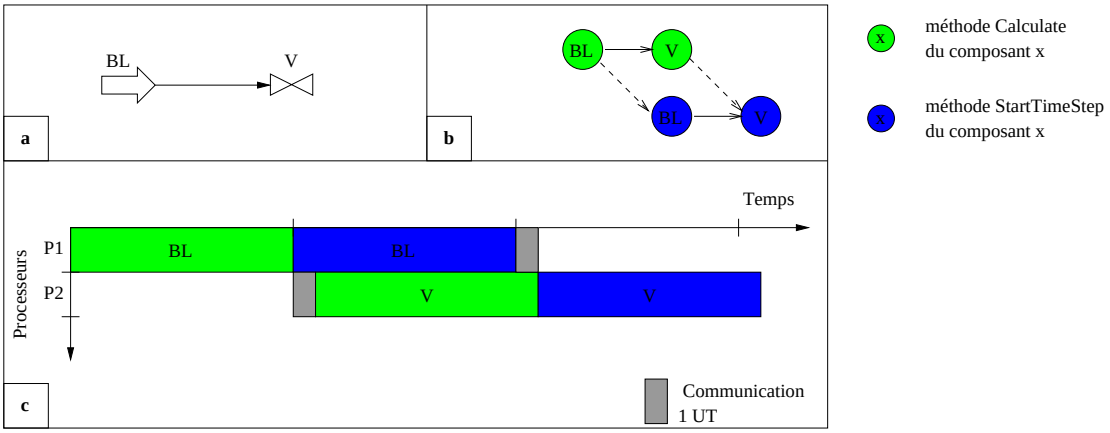


FIG. 6.9 – Parallélisme inter pas de temps de type *pipeline*. Le schéma de procédé simple (a) définit un graphe de flot de données (b) où chaque composant est affecté à un site d'exécution particulier. Le diagramme de Gantt (c) estime le temps de complétion pour un coût d'exécution de 10 unités de temps (UT) pour chaque tâche et un coût de communication de 1 UT.

dent de la figure 6.9. La version du *Material Object* consommée par la méthode `Calculate` de

V et celle produite par la même méthode sur le composant BL. Il en va de même pour la méthode `StartTimeStep`. Il est essentiel de s'assurer que les méthodes de V utilisent la bonne version du *Material Object* et non une version future.

Dans le cas où un parallélisme structurel est employé, le moteur exécutif doit également traiter les anti-dépendances. Un moyen simple de les résoudre est la mise en place d'une barrière de synchronisation à la fin de chaque pas de temps ce qui garantit l'assignation unique des *Material Object* sur chaque itération.

6.6 Bilan

Ce chapitre a présenté en détail l'algorithme de résolution utilisé par RSI dans INDISS. Nous avons décrit les dépendances entre les tâches ce qui nous permet de construire une représentation de l'exécution sous la forme d'un graphe de flot de données.

Cette représentation est la base de l'étude du parallélisme engendré par le schéma d'exécution d'une simulation dynamique dans INDISS. Nous considérons deux sources, parmi les trois identifiées, pouvant apporter un gain lors d'une exécution distribuée :

- un parallélisme structurel est engendré par la topologie du schéma de procédé.
- un parallélisme inter pas de temps est un parallélisme de *pipelining*.

Le parallélisme structurel est celui qui permet, dans des conditions idéales où le parallélisme du schéma de procédé est fort, d'obtenir un rendement important. En outre, il est simple à mettre en place. Les anti-dépendances peuvent être traitées par la mise en place d'une barrière de synchronisation à la fin de chaque pas de temps.

La chapitre suivant traite des techniques mises en œuvre afin de concevoir un environnement d'exécution parallèle pour les simulations de procédés basé sur un parallélisme structurel en se basant sur l'étude de ce chapitre. La chapitre 8 présentera les résultats expérimentaux obtenus sur une grappe de calcul.

Environnement d'exécution parallèle pour les simulations de procédés 7

7.1 Introduction

Ce chapitre traite des techniques implantées permettant de distribuer et d'exécuter une simulation de procédés sur une grappe de calcul. Cette distribution nécessite l'identification du parallélisme et le choix d'un placement efficace des divers composantes de l'application. Ceci se traduit par la construction d'un graphe de tâches qui représente la structure de données élémentaire à tout algorithme d'ordonnancement. A partir de cette représentation, l'ordonnanceur calculera un placement logique de chaque tâche du graphe. Une fois déployées, les partitions logiques forment des flots d'exécution sur les processeurs de la grappe de calcul. La section 7.2 décrit la démarche permettant de transformer un schéma de procédé en divers flots d'exécution.

La création des p flots d'exécutions affectés à chacun des p processeurs de l'architecture matérielle ne suffit pas à qualifier une exécution distribuée, *a fortiori* lorsque l'application est une simulation numérique complexe exposant de nombreuses caractéristiques propres au métier. Pour traiter les simulations de procédés, notre approche définit deux entités conceptuelles fortement inspirées du schéma conceptuel d'INDISS-RT : *coordinateur central* et *coordinateur local*. Ces deux entités sont implantées sous la forme deux moteurs exécutifs distincts ayant en charge un flot de contrôle particulier. La section 7.3 traite des propriétés exécutives de ces deux entités.

La section 7.4 conclue ce chapitre par les limites des techniques retenues.

7.2 Du schéma de procédé aux flots d'exécution

Cette section vise à définir les méthodes permettant l'automatisation du processus de transformation d'un schéma de procédé quelconque en structure permettant son exécution sur une architecture à mémoire distribuée. Présenté en figure 7.1, ce processus suit trois étapes :

1. Construction du graphe abstrait de l'exécution du schéma de procédé. Pour cela, il est nécessaire de définir l'élément d'exécution ainsi que les flots de données requis et produits par chacun d'entre eux. Chacun de ces éléments représente des nœuds et des arcs dans le graphe généré.
2. Ordonnancement. Le graphe de précédence est la brique de base à tout algorithme d'ordonnancement. Il permet de générer autant de partitions logiques que de ressources de calcul

disponibles tout en minimisant une fonction objectif (qui est dans notre cas la minimisation du temps d'exécution).

3. Déploiement des partitions logiques sur les ressources de calcul. Cette étape réalise également un pré-traitement visant à identifier les communications qui auront lieu à l'exécution. A réception de ces informations, un service s'exécutant sur chaque nœud de la grappe crée le flot d'exécution associé.

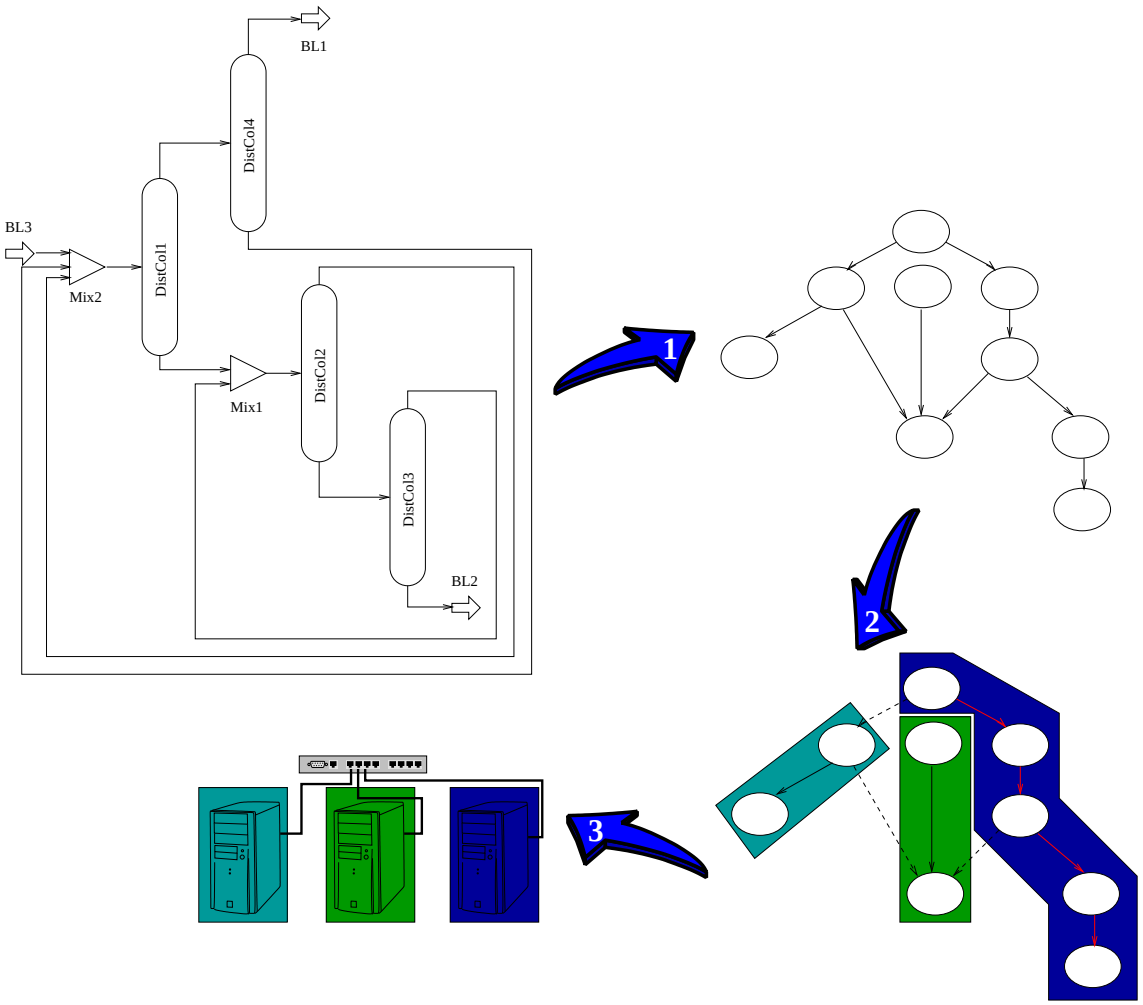


FIG. 7.1 – Processus en trois étapes de transformation d'un *flowsheet* en flots d'exécution distribués. Etape 1 : définition de l'élément d'exécution du graphe abstrait associé à un *flowsheet*. Etape 2 : ordonnancement du graphe selon le critère minimisation du *makespan* et création des partitions logiques. Etape 3 : distribution des partitions sur les différents processeurs et création des flots d'exécution associés.

La suite de cette section expose plus amplement chacune de ces étapes.

7.2.1 Etape 1 : transformation du schéma de procédé en une représentation abstraite de l'exécution

Le processus d'élaboration d'une application parallèle passe par la détection des instructions unitaires qui pourront potentiellement être exécutées en parallèle (section 3.3). Les environnements de programmation parallèle classiques traitent le parallélisme selon un paradigme de programmation procédural où les contraintes de dépendance entre instructions sont totalement définies au travers des paramètres d'appel de méthodes. Par nature, ces instructions n'ont pas d'état : la durée de vie des variables dépend du temps d'activation de la méthode. Or l'approche par composants, à l'instar de la programmation orientée objet, suppose que tout appel de méthodes sur un composant peut à tout moment utiliser ses attributs qui composent son état.

Ainsi, bien que l'élément d'exécution reste l'appel de méthodes sur les composants, il est nécessaire de tenir compte des propriétés intrinsèques liées au modèle de programmation : un composant est une unité d'exécution et de déploiement. L'aspect modulaire contraint le choix cet élément aux composants de type opération unitaire. Le graphe de précédence qui en découle n'est autre que le schéma de procédé orienté selon les contraintes d'écoulement des flux. La présence potentiel de recyclage nécessite la construction, au préalable, du graphe de tâche acyclique au moyen d'un algorithme métier de coupe de flux. Le graphe de précédence représente alors le schéma d'exécution des étapes *FirstCompute* et *LastCompute*. Ce choix se justifie lorsque l'on considère les proportions de temps d'exécution de ces étapes. Le tableau 7.1 présente ces pourcentages. Le temps d'exécution de l'étape *NetworkCompute* est négligeable comparé aux deux autres méthodes.

	FirstCompute	*NetworkCompute*	*LastCompute*
% de la charge de calcul	10% (10 à 80)	5% (5)	85% (15 à 85)

TAB. 7.1 – Rapport du temps moyen d'exécution de chaque étape sur le temps total d'exécution de la simulation. Le premier pourcentage représente le temps de calcul pour des opérations unitaires de RSI ; le second correspond à ce qui est consommé par l'une de ces opérations qui est le « *pipeline* » IFP.

Notre approche privilégie alors un parallélisme structurel. Comparé au parallélisme inter pas de temps, le placement généré par l'ordonnanceur paraît *a priori* plus *stable*. En effet, la charge de calcul de la première étape *FirstCompute* varie fortement (de quelques microsecondes à plusieurs secondes). Cette variation est étroitement liée au modèle de résolution d'une opération unitaire de notre cas test et sera présentée en section 8.3. En pratique, nous verrons dans la partie III que le choix d'un parallélisme inter pas de temps ou structurel a peu d'impact sur l'ordonnancement et donc sur les performances de l'exécution parallèle.

L'étape suivante regroupe les différentes tâches (éléments d'exécution) en partitions tout en tenant compte des contraintes de précédence entre les tâches.

7.2.2 Etape 2 : regroupement des tâches en partitions

Les techniques permettant de partitionner un graphe de précédence tout en minimisant une fonction objectif reposent sur l'utilisation d'un **ordonnanceur**.

Afin de résoudre ce problème d'ordonnancement, nous avons intégré dans notre prototype l'algorithme de liste DSC présenté en section 4.4. DSC provient de la bibliothèque PYRROS [123][1]. Ainsi, nous avons dû concevoir un traducteur nécessaire à la transformation de la structure de données INDISS-RT en une structure compatible avec celle nécessaire à DSC. Elle se présente sous la forme d'un tableau d'entiers représentant le graphe d'exécution selon ses tâches, elles-mêmes définies par ses relations de succession. Une tâche est alors représentées par le n-tuple :

$$(\text{Id}, \text{Exec}, \text{Id}_1, \text{Comm}_1, \text{Id}_2, \text{Comm}_2, \ldots, \text{Id}_n, \text{Comm}_n)$$

où

- *Id* : identifiant de la tâche,
- *Exec* : temps estimé d'exécution pour la tâche *Id*,
- Id_k : identifiant de la tâche du $k^{\text{ème}}$ successeur de la tâche *Id*,
- $Comm_k$: temps estimé de communication des données produites par la tâche *Id* et consommées par la tâche Id_k.

Afin de réaliser cette transformation, la structure de données INDISS-RT nous offre une représentation abstraite du schéma de procédé où pour chaque opération unitaire nous connaissons celles en amont et en aval. Les flux coupés sont également recensés. Ces deux informations permettent de construire la représentation requise en entré de l'algorithme DSC.

Nous verrons en partie III, l'utilisation d'un autre ordonnanceur ETF [60].

Le fait de partitionner le graphe d'exécution permet dans une dernière étape de déployer les p partitions obtenues sur p entités physiques de calcul.

7.2.3 Etape 3 : déploiement d'un ensemble de partitions logiques

Le déploiement est considéré comme un pré-traitement à l'exécution distribuée de la simulation, le schéma que nous avons utilisé est de type *one-to-all* où l'entité de traitement qui héberge l'ordonnanceur diffuse, à tous les nœuds de la grappe de calcul, les informations nécessaires à la création des flots d'exécution distribués. Le coordinateur de l'application (en l'occurrence le logiciel INDISS) détient la connaissance des paramètres et des propriétés du schéma de procédé à simuler. Les informations utiles à chaque partition sont alors sauvegardées et envoyées à chaque ressource de calcul à la fin de la validation du schéma de procédé. La validation, définie dans le standard CAPE-OPEN, s'assure que le schéma de procédé est correctement configuré (*e.g.* chaque connexion est connectée à une opération unitaire en amont et à une autre en aval). Elle a lieu avant le début de l'exécution de la simulation.

[1] Téléchargeable à l'adresse http://www.cs.ucsb.edu/~tyang/papers/PYRROScode.html.

Sur chaque entité de calcul distante, la description du calcul à effectuer est reçue et le flot d'exécution associé est créé. Chaque entité de calcul peut accéder à chacun des composants métier impliqués dans la simulation. Ces derniers sont instanciés localement, en fonction des besoins.

Cette étape calcule également, au préalable, les communications nécessaires à l'exécution. Ces communications sont déduites des partitions générées par l'ordonnanceur et de la représentation complète du schéma de procédé (*i.e.*, comprenant les recyclages).

7.2.4 Composants métier distribués

Les partitions déployées sur chaque ressource de l'architecture parallèle ne sont que des descriptions logiques permettant d'instancier les composants métier. Afin d'exécuter les tâches d'une partition, les trois classes de composants suivantes sont nécessaires.

1. Opération unitaire
2. Serveur thermodynamique
3. *Material Object*

Chaque opération unitaire du schéma de procédé appartient à une et une seule partition formant alors des ensembles disjoints pour qualifier le graphe complet. Conceptuellement, le standard CAPE-OPEN associe le serveur de calcul thermodynamique au PME définissant alors une entité d'exécution centralisée. Or, ce dernier ne possède pas d'état et peut ainsi être distribuée sur chaque ressource de calcul. Seuls les *Material Object*, représentant des conteneurs de données, posent problème.

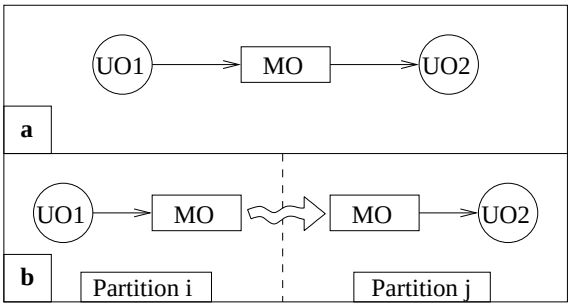


FIG. 7.2 – Comparaison entre *Material Object* partagé et dupliqué. Un *Material Object* dont les opérations unitaires en entrée et en sortie sont affectées à la même partition est naturellement partagé (a). Dans le cas contraire (b), il est dupliqué et appartient aux deux partitions.

Un problème de localisation est levé lorsqu'un *Material Object* est dupliqué entre deux opérations unitaires, chacune localisée sur un processus différent. La figure 7.2 présente une comparaison schématique entre *Material Object* local et distribué. Connaissant le flot de données, deux techniques peuvent être implantées pour traiter les communications :

– La première technique consiste à placer le *Material Object* sur l'un des processus. Les deux opérations unitaires référencent le même objet et les communications sont donc laissées à la charge du *middleware*. De nombreuses communications peuvent être générées sur les variables atomiques composant le *Material Object*.
– La seconde technique consiste à dupliquer le *Material Object* sur les deux processus. Un protocole de cohérence de cache doit être utilisé afin de garantir la consistance des valeurs stockées dans le *Material Object*. Ceci est assuré par le moteur exécutif qui s'appuie sur les contraintes définies par le graphe de flot de données. Une fois les valeurs mises à jour, celles-ci sont communiquées au *Material Object* destinataire et la tâche qui en dépend sera exécutable. Cette technique est détaillée plus en avant en section 7.3.3.

La section suivante présente les moteurs exécutifs distribués implantés.

7.3 Coordinateur Central et Local

Ce découpage en étapes donne alors naissance à deux entités conceptuelles, présentées sur la figure 7.3, ayant chacune la charge d'un flot de contrôle particulier. La première entité est le **coordinateur central** de l'exécution qui est associée à la résolution par étape définie dans INDISS-RT. Elle regroupe les invocations à effectuer sur l'ensemble du graphe abstrait du schéma de procédé.

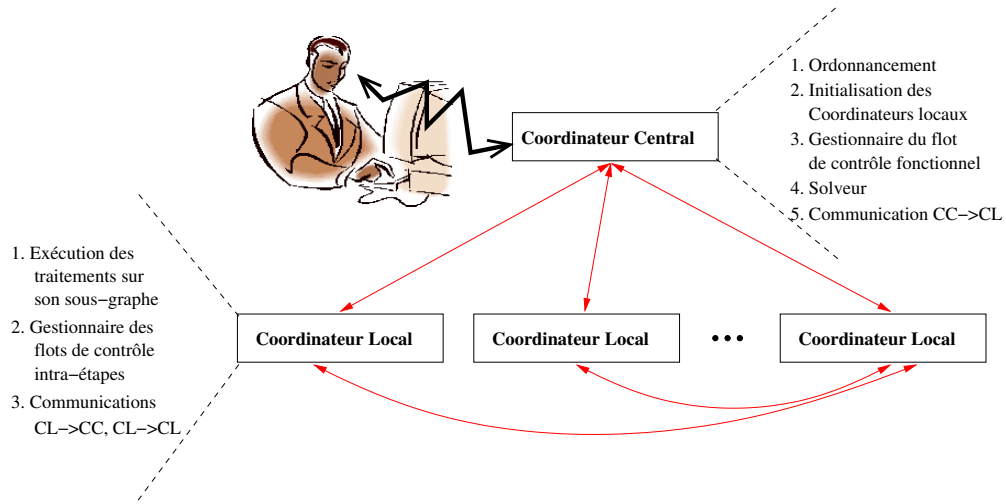


FIG. 7.3 – Vue fonctionnelle des entités intervenant dans une simulation distribuée.

La résolution de chaque étape nécessite l'appel ordonné des méthodes CAPE-OPEN sur les opérations unitaires en respectant les contraintes de précédence. Chaque processeur distant gère son propre flot de contrôle sur l'ensemble des opérations unitaires qui le compose. Nous nommons cette entité **coordinateur local**.

7.3.1 Choix d'implantation

Afin de conserver une cohérence dans le choix des outils de programmation utilisés dans le reste de INDISS-RT, coordinateurs central et local sont implantés selon le modèle à composants CORBA. La figure 7.4 présente le diagramme de composants selon la norme CORBA3 [84]. D'après cette figure, sources et puits représentent des méthodes non-bloquantes définissant une

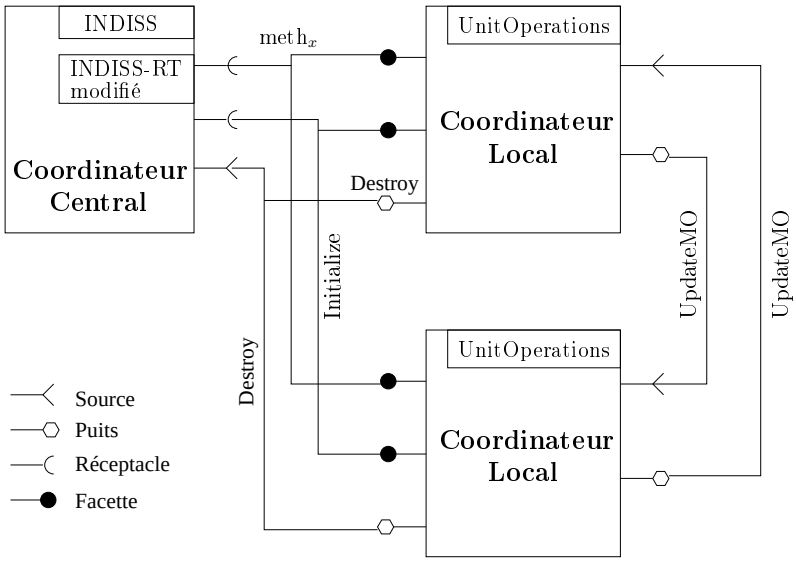


FIG. 7.4 – Organisation fonctionnelle des entités prenant part à une simulation distribuée.

propriété événementielle du composant qui ne nécessitent pas de valeurs de retour. En revanche, les réceptacles / facettes représentent des appels de méthodes bloquantes exploitant des paramètres de retour. L'implantation utilisée dans le cadre de cette thèse est OmniORB [85] 4.0.4.

7.3.2 Coordinateur central

La résolution dynamique repose sur l'exécution séquentielle des étapes fonctionnelles d'INDISS-RT. Les traitements associés réalisent un ensemble d'invocations de méthodes CAPE-OPEN sur les opérations unitaires du schéma de procédé. Les étapes INDISS-RT considérées sont définies dans le chapitre précédent à la figure 6.4 pages 104.

Les opérations unitaires étant réparties sur un ensemble de processeurs, le schéma de procédé est un graphe distribué G_{dist} :

$$G_{\text{dist}} = \{G_i\} \text{ pour } i = 0 \ \ldots \ n-1$$

G_i est le sous-graphe (défini par l'ordonnanceur) sur le processeur i. Le traitement f associé à une étape sur le graphe distribué G_{dist} pour n processeurs est défini comme suit :

$$f(G_{\text{dist}}) = f(G_0) \cup f(G_1) \cup \ldots \cup f(G_{n-1})$$

La notion de graphe distribué repose sur l'interception et la transmission des appels de méthodes exécutives définies dans INDISS-RT (*FirstCompute, NetworkCompute* et *LastCompute*) à destination de chaque processeur p_i. L'implantation d'un tel concept ne modifie que très peu le moteur exécutif INDISS-RT.

En plus de ces propriétés, le coordinateur central est doté de fonctionnalités nécessaires aux pré-traitements de l'exécution. Par exemple, les méthodes `Schedule` et `Distribute` permettent respectivement d'ordonnancer le graphe et de distribuer les partitions générées.

Le solveur, indispensable à l'étape *NetworkCompute*, est un code séquentiel et centralisé car il nécessite la contribution de chaque opération unitaire. Il est alors naturel de le placer sur le coordinateur central.

La conception de ce coordinateur central s'est orientée sur la spécialisation du PME INDISS. En effet, il présente les mêmes fonctionnalités que ce dernier (*e.g.* conception du schéma de procédé, configuration du serveur de calculs thermodynamiques et des opérations unitaires ...) tout en apportant les fonctions nécessaires aux pré-traitements liés à l'exécution distribuée (ordonnancement / déploiement).

Le tableau 7.2 présente les paramètres d'entrée et de sortie pour chaque invocation de méthodes initiée par le coordinateur central.

Méthodes	**Paramètres d'entrée**	**Paramètres de sortie**
FirstCompute	$\emptyset$	Compressibilités initiales
NetworkCompute	Pressions	Débits, Compressibilités et Dérivées
LastCompute	$\emptyset$	*Material Object*

TAB. 7.2 – Paramètres d'entrée et de sortie de chaque invocation de méthodes initiée par le coordinateur central.

Le flot de données exposé par ces méthodes correspondent majoritairement aux dépendances de données liées au solveur (contraintes de précédence 3a, 4a, 4c de la section 6.4, page 107). Les paramètres de sortie de *LastCompute*, pour le coordinateur central, n'ont pas d'intérêt exécutif. Ils servent uniquement à des fins de visualisation (contrainte énoncée en section 6.4.2, page 108). Le coordinateur central est également l'entité responsable des interactions avec l'utilisateur.

Ce coordinateur central nous permet, d'une part, d'exécuter les calculs qui sont centralisés (ordonnancement, solveur) et, d'autre part, de faire le lien avec l'environnement de simulation INDISS-RT (visualisation). Outre les propriétés de pré-traitements indispensables à la configuration de l'exécution distribuée, le schéma d'exécution suit le schéma itératif naturellement séquentiel présenté dans la section 6.2. L'obtention de performances repose sur la distribution des calculs et leur gestion par les coordinateurs locaux.

7.3.3 Coordinateur local

Chaque coordinateur local CL_i est garant de la cohérence de l'exécution intra-étapes du sous-graphe G_i qui lui est attribué. Les méthodes exécutives sont invoquées sur le coordinateur central qui les retransmet aux coordinateurs locaux. En section 6.2, nous avons présenté les étapes *FirstCompute* et *LastCompute* qui appellent respectivement les méthodes `StartTimeStep` et `Calculate` sur chacune des opérations unitaires selon une séquence spécifique liée à la coupe des recyclages. La super-étape *NetworkCompute* nécessite un schéma d'exécution itératif où la méthode CAPE-OPEN `ResolveFlowPressure` doit, dans un premier temps, être appelée sur toutes les opérations unitaires de type arc, et dans un second temps, sur toutes celles de type nœud. D'un point de vue exécutif, un coordinateur local peut être considéré comme une opération unitaire CAPE-OPEN composite car il offre les mêmes fonctionnalités exécutives sur un ensemble de composants élémentaires opération unitaire.

Toutefois, le coordinateur local est un moteur exécutif devant traiter les synchronisations nécessaires au respect des contraintes de précédence et de dépendance. La gestion de la cohérence de l'exécution suit une technique d'activation des tâches prêtes. Une tâche est dite prête si les données nécessaires à son exécution ont été produites. L'état d'une tâche est ainsi déduit du graphe de précédence représentant l'exécution de l'application. Une tâche correspond à une opération unitaire sur laquelle la méthode CAPE-OPEN, assimilée à l'étape INDISS-RT en cours de réalisation, est invoquée.

Chaque coordinateur local conserve une liste de tâches prêtes (*i.e.*, opérations unitaires pour lesquelles toutes les dépendances de données sont respectées) qui évolue dynamiquement en fonction des exécutions et des réceptions de données. L'exécution des étapes *FirstCompute* et *LastCompute* suit un algorithme simple présenté en figure 7.5.

Début du code

```
// EnsNonPrêt : ensemble des tâches non prêtes
// EnsPrêt    : ensemble des tâches prêtes

Tant que EnsPrêt != vide  ET EnsNonPrêt != vide
Faire
  // Récupérer la première opération unitaire de la liste lstUOPrêt
  pUO = Get_from_PriorityList();

  // Exécuter sur l'opération unitaire la méthode CAPE-OPEN StartTimeStep ou Calculate
  // selon l'étape en cours
  pUO.execute()

  // Mettre à jour les Material Object de sortie à l'opération unitaire
  UpdateMO(pUO)
Fait
```

Fin du code

FIG. 7.5 – Calcul des étapes *FirstCompute* et *LastCompute*.

L'extraction de l'opération unitaire de la liste est une opération bloquante : si la liste est vide alors qu'il reste des tâches à exécuter, le processus se bloque sur une attente passive. Lorsque la

réception d'un *Material Object* entraîne l'insertion d'une opération unitaire dans la liste de tâches prêtes, le flot de contrôle principal se réveille pour l'exécuter.

L'activation des tâches qui dépendent de données produites par des processus distants nécessite l'initiation de communications inter-processus. Pour cela, le coordinateur local gère le flot de données en mettant à jour les *Material Object* de sortie. Le traitement associé consiste à envoyer la donnée et à faire passer à *prêt*, le cas échéant, la tâche possédant en entrée le *Material Object* dupliqué sur le coordinateur local distant. Si le *Material Object* n'est pas dupliqué et donc local, la liste des tâches prêtes à l'exécution est modifiée en fonction des dépendances de données. Le traitement associé est défini en figure 7.6.

Début du code

```
Pour chaque Material Object mis à jour
Faire
  Si il est dupliqué
  Alors
    Envoyer le Material Object au site distant consommateur.
  Sinon
    Récupérer l'opération unitaire connectée en sortie du Material Object
    Si tous ses Material Object en entrée sont produits
    Alors
      Insérer l'opération unitaire dans l'ensemble EnsPrêt des tâches prêtes
    FinSi
  FinSi
Fait
```

Fin du code

FIG. 7.6 – Traitements associés lors de la mise à jour des connexions de sorties en réponse à l'exécution des méthodes `StartTimeStep` ou `Calculate` sur une opération unitaire.

La résolution distribuée des étapes du *NetworkCompute* met en jeu principalement un schéma d'exécution Coordinateur Central ⇔ Coordinateurs Locaux. Toutefois, il existe une contrainte de dépendance entre les étapes *ArcNC* et *NœudNC* où les débits calculés sont répercutés entre les *Material Object* dupliqués. Ainsi, la réalisation de l'étape *NœudNC* est soumise à la réception de tous les débits produits à distance. Cette dépendance marque alors une barrière de synchronisation entre les étapes *ArcNC* et *NœudNC*.

La section suivante présente les limitations soulevées par l'approche retenue ainsi que celles imposées par l'environnement de simulation INDISS utilisé dans ce travail.

7.4 Limitations

Les limitations à une exécution distribuée efficace sont liées à trois facteurs : 1/ la gestion des anti-dépendances ; 2/ l'environnement de simulation INDISS ; 3/ les composants métier disponibles.

La première limitation provient de l'interprétation des contraintes de précédence sur les étapes de la résolution dynamique d'INDISS-RT. Initialement conservé pour des raisons de simplicité, ce schéma d'exécution nous permet également de traiter simplement les anti-dépendances soulevées

lors de simulations itératives. Ceci a motivé la mise en place d'un second prototype (qui sera développé dans la partie III) où le recouvrement entre les étapes *LastCompute* et *FirstCompute* entre deux pas de temps est possible.

La simulation de procédé requiert généralement de nombreux composants métier dont certains sont au standard CAPE-OPEN alors que d'autres sont propriétaires à l'environnement de simulation. Cette remarque nous a conduit à implanter notre moteur exécutif au dessus de INDISS-RT. Deux raisons majeures ont motivé ce choix :

- les traitements de gestion du graphe ainsi que d'instanciation des composants dans l'environnement exécutif sont implantés. INDISS-RT est alors un conteneur de composants ;
- la distribution du moteur exécutif INDISS-RT permet théoriquement de distribuer les composants propriétaires.

Cette dernière considération n'est toutefois pas exacte. En effet, les composants métier INDISS sont fortement couplés à l'interface graphique et non au moteur exécutif pour des raisons de propriétés intellectuelles. Ainsi, seuls les composants conformes au standard CAPE-OPEN peuvent être répartis sur les différentes ressources de calcul de la grappe.

Enfin, la dernière difficulté (et de taille) repose sur le système d'exploitation nécessaire à l'exécution des composants métier : Microsoft Windows. Bien que largement utilisé industriellement, la conception de grappes de calcul repose majoritairement (et presque exclusivement) sur des système de type UNIX. Pour cela, le standard CAPE-OPEN propose les spécifications d'interfaces pour les *middleware* DCOM et CORBA. Alors que CORBA est multi plate-forme, la technologie DCOM est propriétaire à Microsoft et ne peut être employée, en pratique, qu'avec le système d'exploitation Windows. Or, les industriels n'implantent que l'interface DCOM à CAPE-OPEN. En théorie, les codes encapsulés sont indépendants d'un environnement cible (un langage de programmation est une norme). Toutefois, tous utilisent des instructions spécifiques à l'API Windows. Notre environnement, bien que portable sur Linux, est, de ce fait, complètement lié au système d'exploitation Microsoft Windows à cause des codes métiers et du moteur exécutif INDISS-RT.

7.5 Bilan

La représentation de l'exécution d'une simulation CAPE-OPEN sous la forme d'un graphe de tâches est une opération complexe. D'une part, elle nécessite l'étude des codes métier afin de détecter les dépendances de données entre les tâches de calcul. En effet, les méthodes CAPE-OPEN sont invoquées sans aucun paramètre. D'autre part, l'approche par composants introduit des contraintes supplémentaires non triviales à résoudre :

- placement : l'élément d'exécution est l'invocation de méthodes sur un composant. L'ensemble formé (représentant un graphe de tâches) est transmis à l'ordonnanceur qui calcule un placement quasi-optimal. Or, un composant est une unité de déploiement imposant le site d'exécution sur l'ensemble des méthodes qu'il offre. L'affectation des sites par un ordonnanceur doit tenir compte des contraintes de localité liées à la nature des composants logiciels. Il n'existe pas, à notre connaissance, d'ordonnanceur capable de traiter cette problématique ;

– précédence : les différentes invocations de méthodes sur un même composant peuvent communiquer à travers des paramètres implicites étant liés à l'état du composant (ses attributs). Il est alors indispensable d'inclure des contraintes de précédence marquant la sérialisation des invocations de méthodes sur un composant.

Pour ces raisons, notre environnement construit une représentation restreinte de l'exécution. Le graphe associé correspond aux schémas d'exécution des étapes *FirstCompute* et *LastCompute* : le parallélisme exposé est alors structurel. La partie III expose une technique de construction de graphe où un parallélisme inter pas de temps (de type *pipelining*) est considéré.

L'exécution distribuée d'une simulation des procédés nous a amené à déterminer deux entités conceptuelles. La première, le Coordinateur Central, est directement induite par la résolution en étapes proposée par INDISS-RT. Le flot de contrôle associé à chacune de ces étapes repose sur la définition de graphe distribué. La réalisation de chaque étape est sous-traitée aux coordinateurs locaux répartis sur différentes machines.

Un Coordinateur Local gère le flot de contrôle et de données intra-étapes. L'exécution d'une étape nécessite la collaboration des processus participant à l'exécution. Ils prennent en charge l'envoi des *Material Object* dupliqués entre les producteurs et les consommateurs. La réception associée met à jour, le cas échéant lorsque toutes les dépendances de données sont satisfaites, l'ensemble des tâches prêtes.

La conservation du modèle de résolution en trois étapes séquentielles inter-dépendantes apporte une solution simple aux anti-dépendances mais introduit des barrières de synchronisation limitant *a priori* les performances globales de l'exécution. Dans la partie III, le gain potentiel d'une approche sans barrière de synchronisation est quantifié.

Le chapitre suivant mesure les gains, sur une grappe de calcul, de notre extension à INDISS-RT pour les simulations de type statique et dynamique sur un cas test métier.

Performances sur une grappe de calcul
8

8.1 Introduction

Afin de valider notre approche distribuée, nous utilisons un cas test métier de simulation d'extraction de pétrole en eau profonde. Sur un tel procédé, la quasi-totalité du temps de calcul est passée dans l'exécution des opérations unitaires *pipeline* où 80 à 90% de charge de calcul consiste à résoudre des problèmes thermodynamiques. Sur les architectures actuelles, la simulation d'un cas test comprenant 12 *pipeline* permet tout juste d'atteindre le temps réel. Cette question relative au temps de calcul est fortement problématique dans le cas où quelques mois, voir même des années de production, doivent être simulés alors que le pas de temps de simulation doit rester petit afin de modéliser correctement les comportements transitoires. Une approche distribuée permet de réduire sensiblement le temps de calcul. Ce chapitre propose de quantifier le gain atteignable au moyen de l'exploitation d'une grappe de PC.

La section 8.2 présente les problématiques du cas test TINA. Cette simulation repose principalement sur l'interconnexion de *pipeline* permettant de conduire le pétrole brut du réservoir aux installations de surface où il est traité (séparation Gaz/Huile/Eau). La section 8.3 expose le modèle d'exécution du *pipeline*.

Notre approche fut initialement appliquée à la simulation statique où le schéma de procédé ne présente pas de recyclages. Son étude fut présentée en [94] et fait l'objet de la section 8.4.

Dans le cas d'un calcul dynamique, un cas métier comprenant deux *gas-lift* (et donc deux recyclages) est simulé. Le *gas-lift* est introduit lorsque le système ne peut plus naturellement produire : la limite d'éruptivité naturelle est atteinte. Les performances obtenues ont été présentées dans [91, 92] et sont reprises dans la section 8.5.

8.2 Application de test : TINA

Pour valider le prototype basé sur les techniques vues au chapitre 7, un ensemble de cas tests métier, issu du projet de recherche TINA, est considéré. Il repose principalement sur l'interconnexion de plusieurs *pipeline* permettant de simuler l'écoulement poly-phasique du pétrole sous contraintes.

TINA est un projet de recherche entre l'Institut Français du Pétrole et Total dont le but est de simuler la production de réseaux *offshore* ou *onshore* de *pipeline* du réservoir aux installations de surface. L'étude des écoulements poly-phasiques est un point clé de la simulation. La figure 8.1 représente un tel réseau de production.

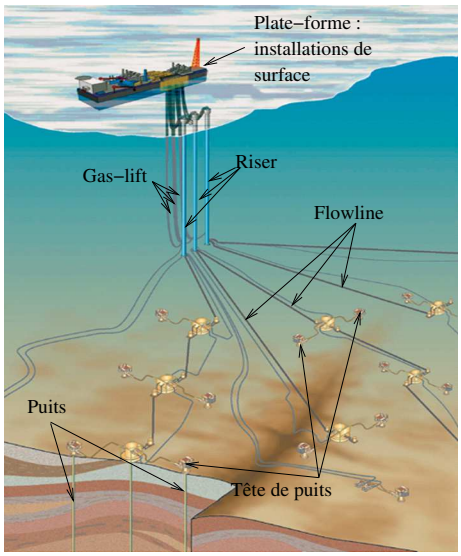


FIG. 8.1 – Représentation d'un réseau de production *offshore*.

Une tête de puits est un ensemble de raccords, vannes, buses, ...installés à la sortie d'un puits de production. *Flowline* et *riser* sont des *pipeline* permettant de transporter respectivement la charge de la tête de puits au *riser* et du *riser* aux installations de surface. Généralement, en début de vie, les puits sont naturellement éruptifs, c'est-à-dire que la seule différence de pression entre le réservoir et la tête de puits permet d'extraire le brut du réservoir. Au bout d'un certain temps de production, la pression dans le réservoir diminue (induit par les pertes de charge dans le système dues à l'augmentation de la quantité d'eau dans le réservoir) et il devient nécessaire d'injecter du gaz, par exemple, en pied de *riser* afin d'alléger le poids de la colonne. Cette technique est appelée « *gaz-lift* ». Généralement, ce gaz provient de la charge (le brut) qui est obtenue au moyen d'un séparateur gaz-liquide (le *scrubber*) situé sur la plate-forme en surface (figure 8.1).

Simuler statiquement, un tel schéma de procédé permet une certaine connaissance du système telle que l'évolution de la production en fonction du « *water-cut* » (pourcentage de la concentration d'eau dans le brut), le besoin de « *gas-lift* », *etc*.

Dans sa résolution dynamique, le procédé permet de valider les procédures opératoires : arrêt, redémarrage, mise en sécurité tels que le temps de refroidissement, les risques de formation d'hydrates (sous l'action du froid associé à l'eau, le gaz présent dans le pétrole forme de la glace (hydrate) susceptible de boucher les conduites), le démarrage avec *gas-lift*, la fermeture d'un puits, ... Il permet également de prédire et de gérer les instabilités de production (*severe-slugging*

ou *terrain-slugging* : bouchon de liquide alternant avec des passages gazeux, le tout associé à d'importantes variations de pression).

Ce cas test repose principalement sur l'exécution d'un ensemble de *pipeline* qui régit fortement le temps d'exécution global de la simulation. La section suivante présente le modèle d'exécution de ce module.

8.3 Modèle d'exécution du pipeline

Un pipe est une conduite permettant de transporter le pétrole d'un point à un autre. Afin de modéliser l'écoulement de la charge, une technique généralement utilisée repose sur une discrétisation spatiale du modèle. Ainsi, la résolution du pipe pour les méthodes CAPE-OPEN `StartTime-Step` et `Calculate` consiste à adopter une approche modulaire séquentielle où chaque maille est interconnectée au moyen d'un *Material Object* interne. Le nombre de mailles modélisant le *pipeline* influe directement sur son temps d'exécution que se soit pour les simulations statique ou dynamique. Plus le nombre de mailles est grand, meilleure sera la précision du résultat. La simulation d'une maille nécessite de nombreux calculs thermodynamiques et hydrodynamiques qui sont coûteux à réaliser si bien que plus il y a de mailles, plus de calculs thermodynamiques sont nécessaires et plus le temps de calcul de l'opération est important.

Dans le cas d'une simulation dynamique, l'exécution de la seconde méthode *NetworkCompute* nécessite que chaque opération unitaire requiert le calcul de sa compressibilité ou de son débit (selon le type de l'opération unitaire) en fonction des pressions fixées par le solveur. Généralement, ces valeurs peuvent être obtenues de manière analytique ne nécessitant alors que peu de calculs. Toutefois, ce mode de calcul n'est pas possible avec l'opération unitaire *pipeline* de type arc. Un solveur interne au *pipeline* est requis afin de calculer le débit où chaque itération utilise une approche modulaire séquentielle.

L'une des premières versions du pipe utilisait cette méthode en interne à chaque appel de méthode CAPE-OPEN *ResolveFlowPressure*. Cependant, elle était totalement inefficace. Ainsi, une version optimisée a été implantée. Elle consiste à factoriser ce traitement durant la première étape *FirstCompute via* une technique de linéarisation de la résolution des équations différentielles permettant d'approximer le débit en fonction des pressions en entrée/sortie. Cette technique est particulièrement robuste à condition que le solveur global propose une pression comprise dans l'intervalle calculé, ce qui est généralement le cas dans la majorité des problèmes de simulation. Notons que ce calcul n'est pas réalisé à chaque *FirstCompute* si, d'un pas de temps à l'autre, l'intervalle est toujours valide. Dans le cas contraire, le temps de calcul de l'étape est plus de 1000 fois supérieur. Ceci sera approfondi dans la section 8.5 où l'étude de l'exécution d'une simulation dynamique est proposée.

8.4 Simulations statiques

Cette section a pour but de présenter les performances obtenues lors de l'exécution d'un schéma de procédé simulant un réseau de production en eaux profondes. Dans un premier temps, nous exposons les deux cas tests métier utilisés. Puis en 8.5.2, il sera fait état de la configuration

du procédé ainsi que de la méthodologie de prise de temps mise en place. Enfin, nous conclurons cette partie statique à travers la présentation des performances obtenues.

8.4.1 Schéma de procédé et évaluation du parallélisme

Afin de valider notre approche sur un calcul statique, deux simulations ont été utilisées. La première simulation, présentée en figure 8.2 (a), modélise la production d'un réseau composé de trois puits. Le pétrole est conduit à travers plusieurs *flowline* et remonté aux installations de surface au moyen de deux *riser*. Le schéma de procédé est décomposé en trois parties :

1. une partie complètement parallèle composée des trois branches en amont (partie 1 sur la figure 8.2 (a)) ;
2. une partie complètement séquentielle (partie 2) ;
3. une partie parallèle, composée des deux branches en aval (partie 3).

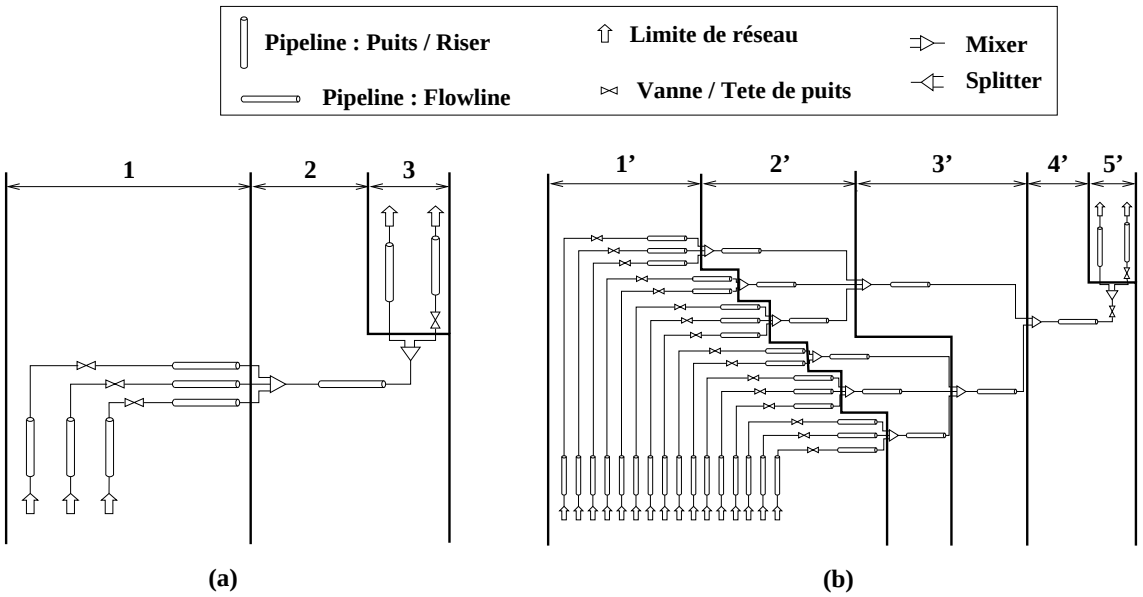


FIG. 8.2 – Cas test métier TINA de simulation d'un réseau de production *offshore* présentant 3 puits (a) et 16 puits (b).

Selon cette distinction, le schéma de communication expose toutes les caractéristiques propres à une exécution distribuée :

– une phase de réduction entre les parties 1 et 2 (cas test (a)) et $1'$ et $4'$ (cas test (b)) ;
– une phase de diffusion entre les parties 2 et 3 (cas test (a)) et $4'$ et $5'$ (cas test (b)).

D'après la topologie du schéma de procédé et de par la méthode de résolution employée de type modulaire séquentielle, le parallélisme maximal est atteignable sur trois processeurs (un pro-

cesseur affecté à chaque branche parallèle) pour le cas test (a).

Dans le but d'utiliser un plus grand nombre de machines, ce premier cas test fut étendu afin d'exposer un plus grand parallélisme. La figure 8.2 (b) présente ce schéma de procédé. Il consiste à simuler un réseau de production composé de seize puits où deux *riser* sont utilisés pour remonter le pétrole aux installations de surface. Le parallélisme maximal est obtenu à travers l'utilisation de 16 processeurs. Le schéma est décomposé en 5 parties. Les 4 premières suivent un schéma de réduction où la transition d'une partie à une autre divise le parallélisme par un facteur compris entre 2 et 3. Ainsi, la partie $1'$ expose un parallélisme de 16, la seconde $2'$ de 6, la troisième $3'$ de 2 et la quatrième $4'$ est séquentielle. A l'instar du cas test précédent, la partie $5'$ définit un schéma exposant un parallélisme maximal de 2. Il est à prévoir que l'efficacité lors de l'utilisation de 16 machines soit assez médiocre due à l'inutilité de certains processeurs après chaque phase de réduction.

8.4.2 Configuration de la simulation et protocole expérimental

Les deux cas tests utilisés afin de valider notre approche exposent respectivement un parallélisme maximal de 3 et de 16. Pour le premier cas test, deux configurations sont définies afin d'apprécier le comportement de notre environnement d'exécution pour des temps d'exécution « faible ». Elles diffèrent par le nombre de mailles et de constituants de la charge. Evoqué en section 8.3, le nombre de mailles influence directement de temps de résolution (le temps de résolution est linéaire en fonction du nombre de mailles). Le nombre de constituants conditionne le temps de calcul de chaque opération unitaire. Les calculs thermodynamiques sont fonction de ce nombre. Par exemple, le temps de calcul d'un flash est proportionnel au carré des constituants considérés. De plus, le nombre de valeurs à virgule flottante contenu par chaque *Material Object* (qui représente le volume de données à communiquer) est également fonction de ce nombre de constituants.

Ainsi, nous appelons configuration *légère*, celle dont la charge est composée de trois constituants (soit des *Material Object* définis par 67 doubles) et les opérations unitaires *pipeline* sont constituées de 5 mailles. Une configuration *lourde* définit une charge de 14 constituants (soit 144 doubles pour les *Material Object*) et les opérations unitaires *pipeline* sont discrétisées par 50 mailles. Ces informations sont résumées dans le tableau 8.1.

	Nombre de constituants de la charge	**Nombre de mailles des pipes**
Configuration légère	3	5
Configuration lourde	14	50

TAB. 8.1 – Propriétés des deux configurations employées pour le cas test 3 puits.

Plusieurs simulations sur une architecture monoprocesseur, utilisant l'environnement de si-

mulation INDISS originel, ont été réalisées au préalable afin d'obtenir les temps d'exécution des méthodes CAPE-OPEN `Calculate` de chaque opération unitaire. Leur somme moyennée sur ces expériences représente le temps d'exécution séquentielle noté T_s.

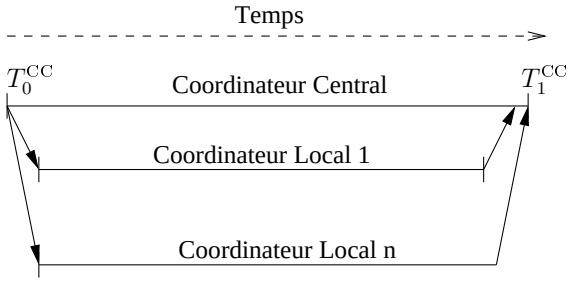


FIG. 8.3 – Méthodologie de prise de temps pour le cas statique.

La méthodologie de prise de temps est présentée en figure 8.3. La date T_0^{CC} est prise juste avant l'appel à la méthode `Calculate` sur chaque coordinateur central. La date T_1^{CC} est relevée après la mise à jour sur le coordinateur central des *Material Object* de tous les coordinateurs locaux. Ainsi, la différence $T_p = T_1^{CC} - T_0^{CC}$ représente le temps d'exécution distribuée sur p processeurs.

Les valeurs T_p, utilisées dans la section 8.4.4, sont issues des moyennes calculées à partir de 10 exécutions menées dans des conditions identiques.

Les métriques accélération S_p et efficacité e_p, vues dans la section 3.5 page 69, sont également utilisées afin de quantifier les performances brutes de l'exécution distribuée.

8.4.3 Environnement d'évaluation

La contrainte sur le système d'exploitation a rendu la conception de la plate-forme expérimentale complexe. En effet, la quasi-totalité des grappes de PC utilise un système de type Unix. Après quelques mois de recherche visant à trouver soit des composants métier CAPE-OPEN CORBA, soit un moyen d'exécuter des codes DCOM sous Linux, la seule solution acceptable fut de concevoir une plate-forme expérimentale basée sur le système d'exploitation Microsoft Windows.

Pour ce faire, 16 machines du *I-Cluster 1* (projet HP, INRIA, ID-IMAG) de Grenoble ont été réutilisées. Les caractéristiques de ces machines sont les suivantes :

- processeur : Intel Pentium III cadencé à 733 MHz ;
- mémoire : 256 MB ;
- réseau : FastEthernet à 100 Mbits/s.
- Système d'exploitation : Windows 2000.

8.4.4 Performances

Dans un premier temps, nous avons comparé les valeurs théoriques et expérimentales pour les métriques efficacité et accélération. Les expériences tenant compte de la distinction configuration

lourde-légère ne furent menées que pour le cas test 3 puits alors que le cas test 16 puits n'utilise que la configuration lourde. Pour ces trois expériences, un modèle de coût simple est utilisé comme estimation des temps proposés à l'ordonnanceur. Les coûts des tâches et des communications ont été respectivement affectés à 10 UT (unité de temps) et 1 UT. L'utilisation d'un tel modèle de coût reste cohérent avec le calcul de la politique globale d'exécution (placement) où un nombre maximal de processeur est utilisé. Le modèle de coût associé à l'expérience présentée en figure 8.7 tient compte des temps mesurés de chaque opération unitaire en utilisant l'environnement de simulation INDISS.

	Accélération S_p		**Efficacité** e_p		T_p	T_s
	S_p^e	S_p^m	e_p^e	e_p^m	(ms)	(ms)
Cas Test 3 **puits** Configuration légère	2,22	2,02	73,9%	67,3%	256	517
Cas Test 3 **puits** Configuration lourde	2,26	2,25	75,4%	74,9%	1914	4 302
Cas Test 16 **puits** Configuration lourde	10,14	9,70	63,4%	60,6%	1928	18 695

TAB. 8.2 – Comparaison des métriques accélération et efficacité estimées (S_p^e et e_p^e) et mesurées (S_p^m et e_p^m) pour les deux cas tests sur un nombre p de processeurs (parallélisme maximal). Les temps séquentiels T_s et parallèles mesurés sont également reportés.

Le temps parallèle estimé T_p^e tient compte du placement des opérations unitaires sur chaque partitions P_i sur un modèle sans communication :

$$T_p^e = \max(T_{P_i}), \text{ pour } i \in [0..p]$$

Les résultats pour les métriques accélération et efficacité sont présentés sur le tableau 8.2. Le calcul de ces métriques est défini comme suit :

- $S_p^e = T_s/T_p^e$
- $e_p^e = S_p^e/p$
- $S_p^m = T_s/T_p$
- $e_p^m = S_p^m/p$

Les figures 8.4, 8.5 et 8.6 présentent les temps T_s, T_p^e et T_p pour un nombre p de processeurs correspondant au parallélisme maximal respectivement pour les cas test 3 puits sur une configuration légère et sur une configuration lourde et 16 puits sur une configuration lourde.

Les résultats présentés montrent une légère différence entre les temps estimé et mesuré. Nous ne pouvons déterminer exactement quel est le surcoût engendré par notre environnement étant donné que les prises de temps ne sont pas suffisamment précises (écart type d'une dizaine de

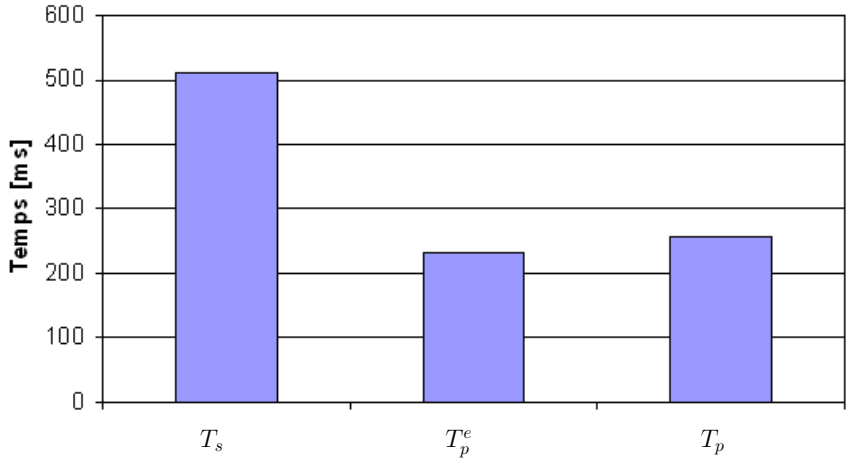


FIG. 8.4 – Temps séquentiel (T_s), parallèle estimé (T_p^e) et mesuré (T_p) sur 3 processeurs pour le cas test 3 puits (configuration légère).

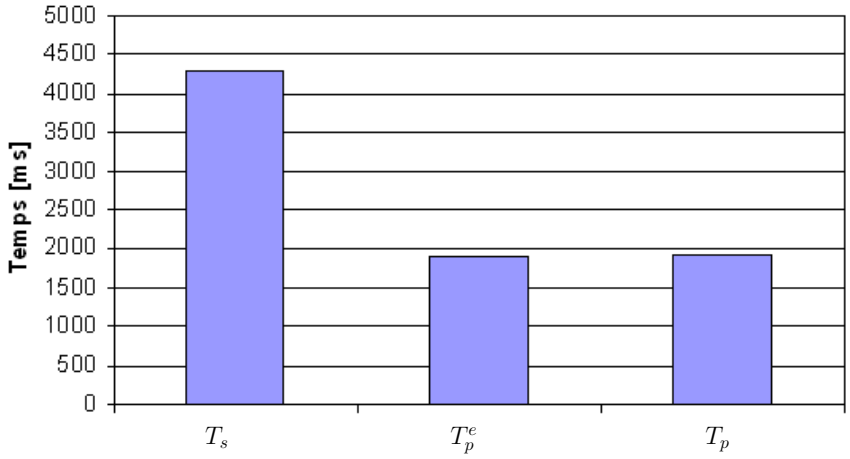


FIG. 8.5 – Temps séquentiel (T_s), parallèle estimé (T_p^e) et mesuré (T_p) sur 3 processeurs pour le cas test 3 puits (configuration lourde).

millisecondes ce qui correspond aux pénalités induites par les communications). Cet écart type observé est prévisible de par le faible nombre d'expérimentations (moyenne de 10 expériences dans des conditions opératoires identiques). Cependant, une approche distribuée d'exécution offre un gain non négligeable d'autant plus important que le grain de l'application augmente (*i.e.* temps de calcul fortement supérieur au temps de communication).

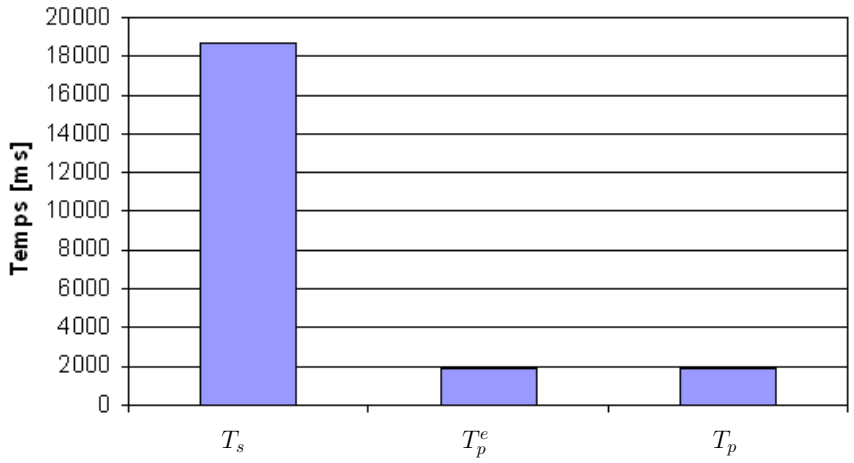


FIG. 8.6 – Temps séquentiel (T_s), parallèle estimé (T_p^e) et mesuré (T_p) sur 16 processeurs pour le cas test 16 puits (configuration lourde).

La figure 8.7 présente une étude de l'accélération et de l'efficacité en fonction du nombre de processeurs utilisés sur le cas test 16 puits. Pour ces expériences, les valeurs affectées aux tâches et aux communications (nécessaires à l'algorithme d'ordonnancement DSC) sont calculées à partir de la moyenne de 15 exécutions précédentes.

Comme présumé, alors que l'accélération croît avec le nombre de processeurs, l'efficacité quant à elle décroît. Ceci est dû, comme précisé auparavant, aux différentes étapes de réduction où le nombre de processeurs en activité est réduit à chaque fois de plus de la moitié.

L'exécution de simulation statiques représente la base à l'évaluation de notre approche distribuée. Cependant, les exigences de performances concernent majoritairement les simulations dynamiques itératives. La section suivante présente l'évaluation de notre environnement pour résoudre ces simulations.

8.5 Simulations dynamiques

Cette section présente les résultats de l'exécution distribuée du cas test TINA en mode dynamique. A l'instar de la section précédente, nous proposons de détailler le schéma de procédé ainsi que les sources de parallélisme. Puis, il sera fait état du comportement de l'exécution du schéma de procédé global dicté par le modèle d'exécution du *pipeline* présenté précédemment en 8.3. Enfin, nous présentons les performances obtenues *via* l'utilisation de notre environnement d'exécution.

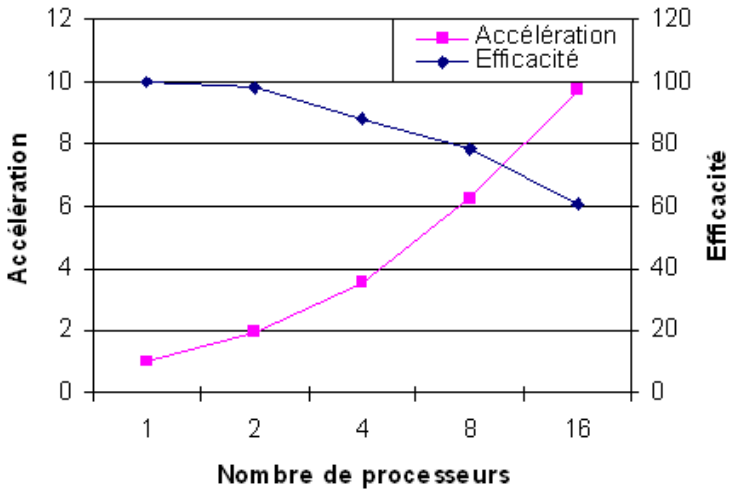


FIG. 8.7 – Evaluation de l'accélération et de l'efficacité en fonction du nombre de processeurs.

8.5.1 Schéma de procédé et évaluation du parallélisme

Le cas test est présenté sur la figure 8.8. Il est composé de dix puits d'extraction et comporte deux *riser*. Un double recyclage est mis en place pour le *gas-lift* en amont de chaque riser. Un algorithme métier est alors utilisé pour déterminer le ou les flux à couper. Seule une coupe est nécessaire à rendre le schéma de procédé acyclique. Elle intervient en amont de la vanne 8 comme indiquée sur la figure 8.8.

Le parallélisme exposé par le cas test est de type réduction. Les 10 puits peuvent être exécutés en parallèle. La réduction intervient au niveau des *flowline 1*. Dès cet instant le parallélisme est réduit à deux tâches pouvant s'exécuter en parallèle et reste ainsi jusqu'à la fin de l'exécution (*flowline 2* et *riser*). Au début de l'exécution, il existe une onzième branche parallèle correspondant aux installations de surface. Ces opérations unitaires sont des composants propriétaires INDISS et ne peuvent être localisées que sur le PME INDISS.

8.5.2 Configuration de la simulation et protocole expérimental

Le schéma de procédé présenté ci-avant est symétrique sur les installations sous-marines (le réseau de *pipeline*) présentant également les mêmes propriétés de configuration. La table 8.3 expose les différentes configurations des composants *pipeline*.

La charge est composée de 15 constituants qui représentent 151 valeurs de type *double* définissant chaque *Material Object*.

La figure 8.9 expose la méthodologie de prise de temps. Pour chaque exécution d'un pas de temps, quatre dates sont relevées sur le Coordinateur Central symbolisant le début d'exécution

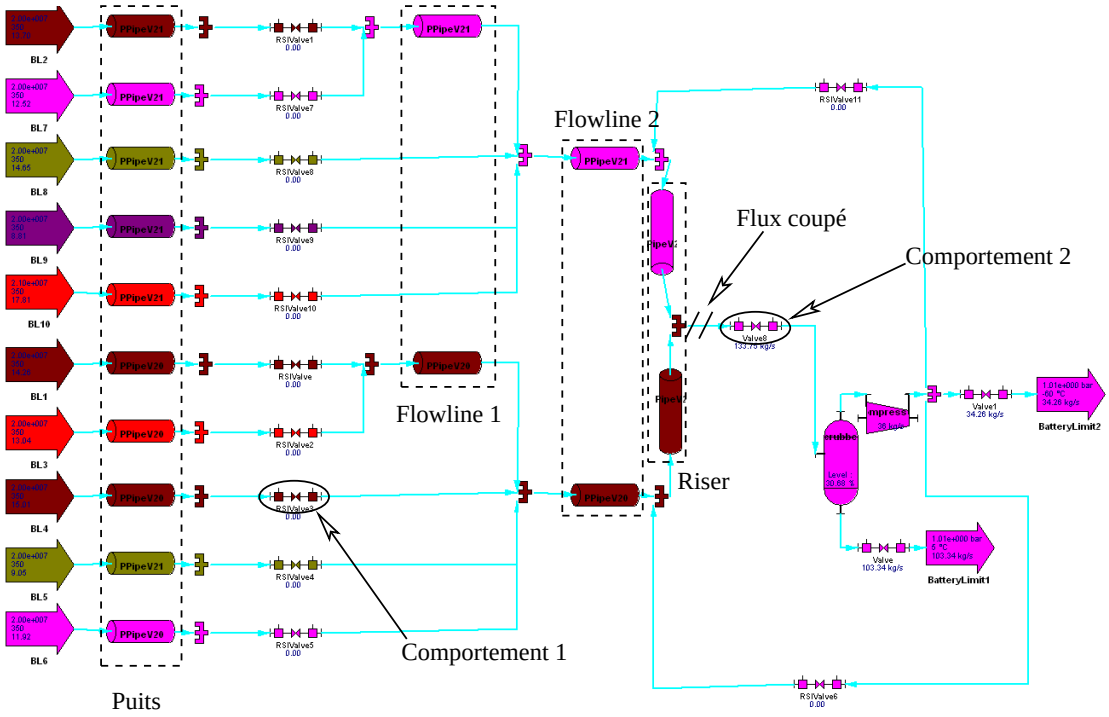


FIG. 8.8 – Schéma de procédé du cas test dynamique.

	Nombre de mailles
puits 1	6
puits 2	15
puits 3	21
puits 4	16
puits 5	15
flowline 1	6
flowline 2	6
riser	34

TAB. 8.3 – Nombre de mailles de chaque module *pipeline* composant le schéma de procédé de la figure 8.8.

d'une étape INDISS-RT. La différence de deux dates consécutives marque le temps de complétion parallèle d'une étape. Comme pour les résultats précédents, les temps utilisés dans la suite de ce chapitre correspondent aux moyennes de 10 expériences.

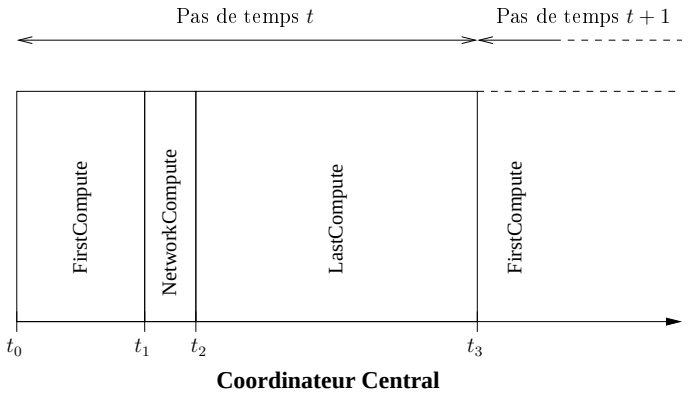


FIG. 8.9 – Méthodologie de prise de temps pour le cas dynamique.

8.5.3 Comportement de l'exécution

L'évolution de la courbe du temps en fonction du pas de temps courant suit un modèle précis et déterministe. La figure 8.10 présente l'évolution du temps de calcul par pas de temps. L'exécution du premier pas de temps est extrêmement coûteuse de par le modèle mathématique du composant *pipeline* basé sur une thermodynamique compositionnelle et d'une hydrodynamique complète. Il entre alors dans une période de calcul « instable » pour atteindre un régime stationnaire à l'itération 13 (sur la figure 8.10) selon notre configuration. Ce régime ne requiert que peu de temps de calcul durant l'étape *FirstCompute* où la linéarisation calculée aux pas de temps précédents sont toujours valide pour le pas de temps courant. Si aucune perturbation n'est injectée, il conserve ainsi ce comportement stationnaire jusqu'à la fin de la simulation.

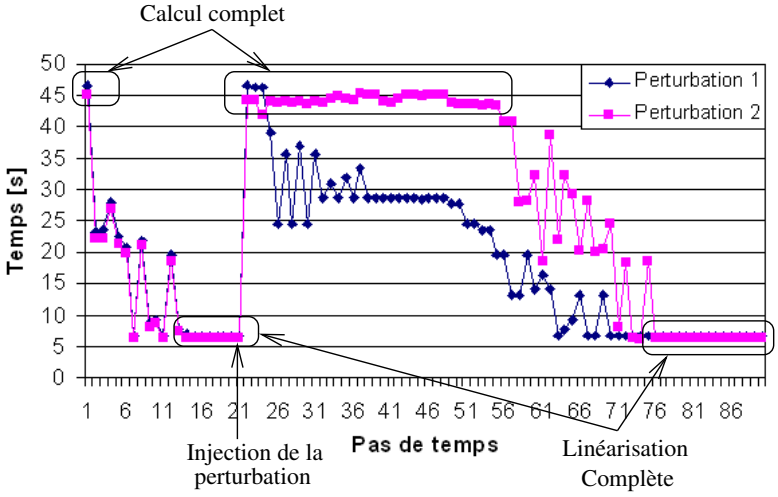


FIG. 8.10 – Comportement de l'exécution dynamique selon la perturbation injectée.

L'injection d'une perturbation a pour effet de déséquilibrer la résolution globale du réseau, ce qui se répercute sur le modèle d'exécution des composants *pipeline*. Les bornes de l'intervalle de linéarisation ne sont plus valides d'un pas de temps à l'autre provoquant alors une hausse sensible du temps global d'exécution. A son apogée, ce temps est près de 8 fois supérieur à celui du régime stationnaire et correspond au calcul complet des équations différentielles par chaque opération unitaire *pipeline*. La figure 8.10 présente la répercution de deux perturbations sur le temps global de résolution d'un pas de temps. Les perturbations 1 et 2 correspondent respectivement à la fermeture des vannes identifiées par les comportements 1 et 2 de la figure 8.8. Le comportement 1 réduit l'ouverture de vanne de 8% à 5% alors que le comportement 2 équivaut à la réduire de 9% à 8%. L'amplitude et la période définissant le régime déséquilibré varient selon la perturbation injectée.

Les expériences menées simulent le comportement 1 et les résultats obtenus sont présentés dans la section suivante.

8.5.4 Performances

Les expériences menées dans cette section sont réalisées sur la plate-forme présentée en section 8.4.3 pages 8.4.3 avec l'ordonnanceur DSC où :

- le graphe ordonnancé est le schéma de procédé après la coupe des recyclages ;
- le poids des tâches est de 10 UT ;
- le poids des arcs est de 1 UT ;
- 5 processeurs ont été utilisés.

La figure 8.11 présente une comparaison des temps d'exécution entre l'exécution monoprocesseur et distribuée sur cinq processeurs. Afin d'apprécier le gain obtenu *via* l'utilisation de notre environnement d'exécution, la figure expose également la courbe des temps minimaux atteignables sur l'architecture matérielle selon l'ordonnancement spécifié. Cette borne inférieure correspond au maximum, pour toutes les partitions générées par l'ordonnanceur, de la somme des temps unitaires de chaque opération unitaire.

Les résultats obtenus montrent que notre environnement d'exécution distribué génère un faible surcoût d'exécution : les courbes « 5 processeurs » et « Borne inférieure » se superposent quasiment. Dans la phase stationnaire, une exécution distribuée réduit sensiblement le temps d'exécution pour chaque pas de temps permettant d'obtenir un temps d'exécution par pas de temps de $3,1$ secondes comparé à $6,71$ secondes pour une exécution monoprocesseur. Le gain de temps le plus probant intervient lorsque le modèle d'exécution est perturbé où en moyenne notre approche distribuée permet de réduire le temps d'exécution par pas de temps à $16,9$ secondes au lieu d'environ 30 secondes.

La figure 8.12 propose d'étudier le comportement de la simulation en se basant sur les trois étapes de résolution INDISS. Pour cela, la métrique accélération est retenue comme élément de comparaison nous permettant également d'analyser le gain de chaque étape selon le placement défini. Dans un premier temps, nous étudions le comportement global de l'exécution.

Dans la phase stationnaire, on observe une accélération globale constante de l'ordre de $2,2$. Lors de l'injection de la perturbation (ainsi que pour le premier pas de temps), l'accélération atteint

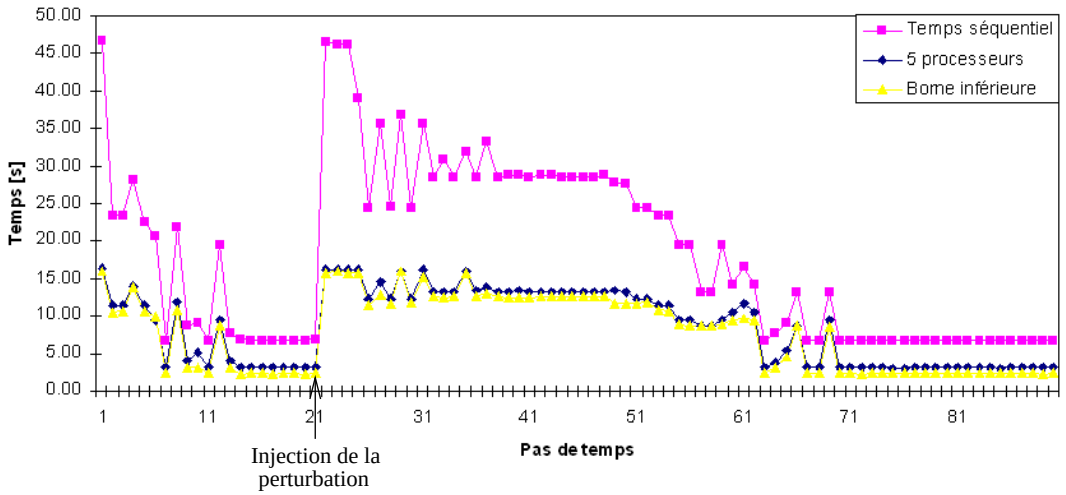


FIG. 8.11 – Comparaison des temps d'exécution selon le pas de temps exécuté.

la valeur de $2,9$. Dans sa plus faible valeur, l'accélération est de $1,4$. Ces différences de valeurs proviennent essentiellement du fait que la perturbation n'est pas absorbée de manière homogène sur le composant *pipeline*. En effet, à un pas de temps t (durant la phase d'instabilité du calcul), il est possible que le modèle de résolution d'un unique *pipeline* nécessite le calcul complet entraînant ainsi un fort déséquilibre dans la charge de calcul. Le maximum est quant à lui atteint lorsque chacun des *pipeline* requiert ce calcul.

Les accélérations de la seconde (*NetworkCompute*) et de la dernière (*LastCompute*) étapes restent constantes durant toute l'exécution : ces deux étapes ne sont pas (ou peu) soumises aux perturbations. Lors de l'exécution de l'étape *NetworkCompute*, la version distribuée est près de deux fois moins rapide que la version monoprocesseur. Ceci s'explique par un ratio calcul/communication en deçà de 1 : le temps de communication est supérieur au temps de calcul. Cependant, il est bon de noter que la perte est négligeable compte tenu de la durée de cette étape (de l'ordre de quelques dizaines de millisecondes par rapport à des dizaines de secondes). La dernière étape est $2,3$ fois plus rapide en distribué. On remarque qu'elle majore le temps d'exécution durant les périodes stationnaires de la simulation (accélération globale quasiment identique à l'accélération *LastCompute*).

La première étape (*FirstCompute*) est la plus problématique. En effet, lors de l'injection de perturbations, c'est elle qui majore le temps d'exécution alors que dans les phases d'exécution stationnaire son temps d'exécution est négligeable. Comme cité précédemment, le modèle de résolution du *pipeline* explique cette variation. Lors de perturbations, l'exécution de la méthode CAPE-OPEN `StartTimeStep` du *pipeline* peut requérir jusqu'à 6 secondes alors qu'elle est de l'ordre de quelques centaines de microsecondes en régime stationnaire. Lorsque ces temps deviennent importants, notre environnement d'exécution permet d'obtenir une accélération non négligeable (près de 3).

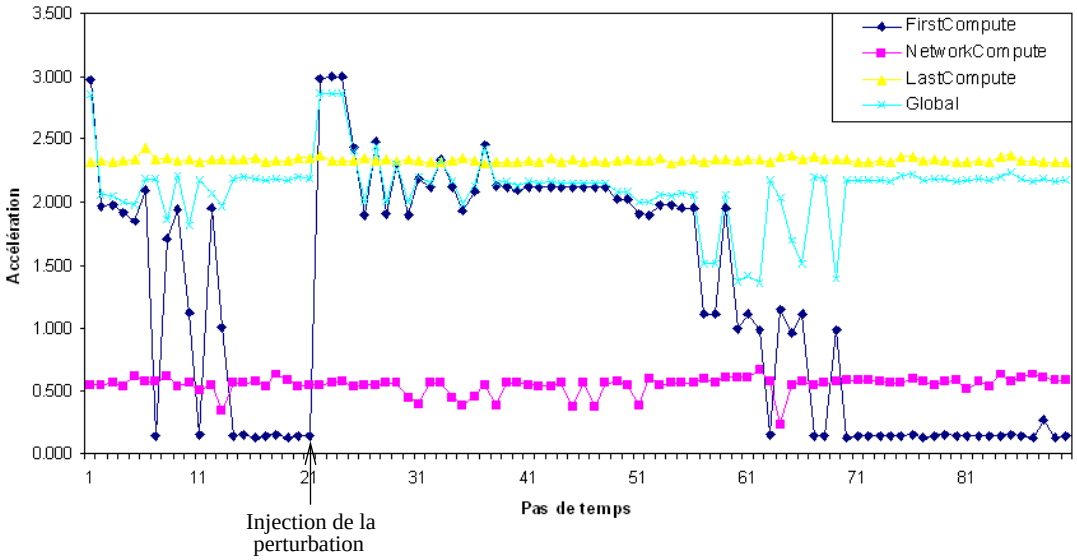


FIG. 8.12 – Etude de l'accélération de chaque étape INDISS-RT.

8.6 Bilan

La simulation de procédé peut tirer bénéfice d'une exécution distribuée dans le but de réduire son temps d'exécution global. Toutefois, notre approche orientée *middleware* reste limitée par la méthode de résolution (modulaire séquentielle).

Les expériences menées dans le cadre des simulations statiques, bien que restreintes aux schémas de procédé ne présentant pas de recyclages, montrent un gain non négligeable (accélération de $9,70$ sur 16 processeurs). Cependant, il convient de constater que son obtention est étroitement liée à la topologie du schéma de procédé et du degré de parallélisme qu'il expose.

L'approche de résolution dynamique, adoptée dans INDISS-RT, est également largement dominée par un calcul modulaire séquentiel. Malgré ceci, un rendement est observable sur le cas test TINA permettant de réduire sensiblement (accélération de près de 3 sur 5 processeurs) le temps d'exécution global de la simulation.

Le solveur reste le goulot d'étranglement de l'exécution distribuée. En effet, l'étape *Network-Compute* nécessite la contribution d'un solveur séquentiel centralisé et représente une barrière de synchronisation forte. Même si la recherche vise actuellement à concevoir des solveurs parallèles, à un moment de la résolution, ils nécessitent une coordination centralisée. Ceci est d'autant plus vrai dans le domaine du CAPE où les équations sont fortement couplées entre elles à travers les flux de matières.

Nous avons consciemment développé notre environnement d'exécution, non pas dans une recherche d'optimalité mais, dans le respect des concepts fonctionnels implantés dans INDISS-RT. Ainsi, nous nous sommes *contentés* de tester uniquement un parallélisme structurel alors qu'un parallélisme inter pas de temps peut offrir un autre placement intéressant à étudier. De plus, les

barrières de synchronisation entre pas de temps limitent les performances globales de l'exécution.

Ces choix furent clairement spécifiés dès le début de thèse. Le développement de TINA ainsi que d'INDISS CAPE-OPEN sont des projets de recherche en cours de conception. Ainsi, notre environnement d'exécution devait être le moins intrusif afin de bénéficier des améliorations / corrections de bogues inhérentes à tout projet de recherche.

Le système d'exploitation Microsoft Windows indispensable à l'exécution des composants métier fut également un frein non négligeable à nos travaux. Le fait que la quasi totalité des grappes de calcul utilise des systèmes de type UNIX, nous a contraint à nous rabattre sur une « *vieille* » architecture (mais qui nous a rendu bien des services) afin de tester notre approche. Toutefois, les expériences menées mettent en évidence la faisabilité d'une approche distribuée pour les simulations CAPE et montre qu'un gain non négligeable peut être obtenu.

Afin de lever les restrictions et de mener une étude plus poussée sur les performances atteignables lors d'une exécution distribuée, nous nous intéressons à concevoir un environnement de simulation du comportement de l'exécution des simulations de procédés. Le cœur de cet environnement repose sur le moteur exécutif KAAPI. Pour cela, nous avons dû l'étendre afin de supporter différentes techniques d'ordonnancement de type statique.

III

Environnement parallèle de simulation itérative

Environnement d'exécution KAAPI et son extension 9

9.1 Introduction

Dans le chapitre 7, les limitations de notre prototype ont été évoquées. La plus pénalisante est la gestion de plusieurs données (ou versions) d'une même variable pour supprimer les anti-dépendances. Ajoutons à cela les limitations introduites par le *middleware* omniORB. L'utilisation de l'attribut *oneway* définissant un appel de méthode non-bloquante ne permet pas le recouvrement des communications par du calcul (seule l'exécution de la méthode est recouverte). La section 9.2 présente l'environnement KAAPI tel qu'il est implémenté au début de cette thèse.

KAAPI offre un support pour résoudre ces limitations, néanmoins, tous les prérequis nécessaires à la réalisation de nos objectifs ne sont pas implantés. En particulier, seule une technique d'ordonnancement par vol de travail est prévue afin de traiter le caractère distribué d'une exécution. Efficace dans le cas des applications de type série-parallèle à grain fin, cette technique est entachée par un fort surcoût à l'exécution pour les autres types d'applications telles que les simulations itératives. C'est pourquoi, nous avons étendu KAAPI afin de permettre une exécution selon une technique d'ordonnancement statique. La section 9.3 expose les extensions qui ont été nécessaires.

9.2 KAAPI : un environnement d'exécution distribué et parallèle

KAAPI [66] possède une API de très haut niveau, nommé Athapascan. Son modèle de programmation simple et son efficacité à l'exécution en font le point de départ idéal pour définir notre environnement du comportement d'une application CAPE-OPEN.

9.2.1 Modèle de programmation et d'exécution

KAAPI est un environnement d'exécution qui, à partir de la représentation de l'application sous la forme d'un graphe de flot de données, offre un support exécutif sur des architectures de calcul parallèle. La construction du graphe DFG repose sur l'API Athapascan [41, 35].

9.2.1.1 Athapascan

Athapascan[1] fournit une API de haut niveau à l'utilisation de l'environnement KAAPI. La figure 9.1 rappelle sur un exemple simple, les trois concepts liés à la programmation d'une application selon le modèle défini par Athapascan. Les mots clés de l'API sont :

- `Shared` : définition d'une variable en mémoire virtuellement partagée ;
- `Fork` : déclaration d'une tâche parallèle ;
- `Shared_X` : mode d'accès sur les variables partagées.

Début du code

```
struct T1{
  void operator()(a1::Shared_r<int> a, a1::Shared_r<int> b, a1::Shared_w<int> c){
    ...
  }
}
struct T2{
  void operator()(a1::Shared_r<int> a, a1::Shared_w<int> b){
    ...
  }
}
struct T3{
  void operator()(a1::Shared_r<int> a, a1::Shared_r<int> b){
    ...
  }
}
struct T4{
  void operator()(a1::Shared_r<int> a){
    ...
  }
}
void main(){
  a1::Shared<int> d1, d2, d3, d4, d5;
  ...
  a1::Fork<T1>() (d1, d2, d3);
  a1::Fork<T2>() (d4, d5);
  a1::Fork<T3>() (d3, d5);
  a1::Fork<T4>() (d5);
  ...
}
```

Fin du code

FIG. 9.1 – Exemple simple d'une application utilisant l'API Athapascan.

L'API de programmation Athapascan permet de décrire un enchaînement de tâches avec dépendances sur des données qui accèdent à une mémoire globale. Cette description est interprétée afin de construire un graphe de flot de données. La représentation d'une application sous cette forme est particulièrement bien adaptée à la définition des synchronisations et des dépendances de données comme présentée en chapitre 4.3.3 page 78. Basé sur cette représentation, KAAPI propose un environnement d'exécution efficace pour les architectures matérielles parallèles et distribuées : l'API Athapascan et le moteur exécutif KAAPI possèdent un modèle de coût algorithmique prouvé en théorie [42] et validé en pratique [62].

[1] Cette interface est présentée en chapitre 3.4.2 page 67

9.2.1.2 Construction du graphe de flot données

Le modèle d'exécution de KAAPI permet de détecter automatiquement les synchronisations et le parallélisme de l'application tout en gérant une exécution distribuée efficace. Pour cela, la représentation abstraite de l'application sous la forme d'un graphe de flot de données est utilisée.

Ce graphe est construit dynamiquement à l'exécution du programme. Au début de l'exécution d'un programme, le graphe est initialisé avec la tâche principale initiale (le « *main* » du programme), qui démarre sur un seul processeur. Chaque instruction de l'API Athapascan est alors interprétée, conduisant à la modification du graphe DFG représentant l'application. L'instruction `Shared` insère un nouveau nœud donnée alors que `Fork` crée un nouveau nœud tâche. En fonction des modes d'accès des tâches sur les données partagées, un arc est inséré entre les nœuds tâches et données. La figure 9.2 représente l'état du graphe de flot de données à chaque interprétation d'une instruction Athapascan du programme 9.1.

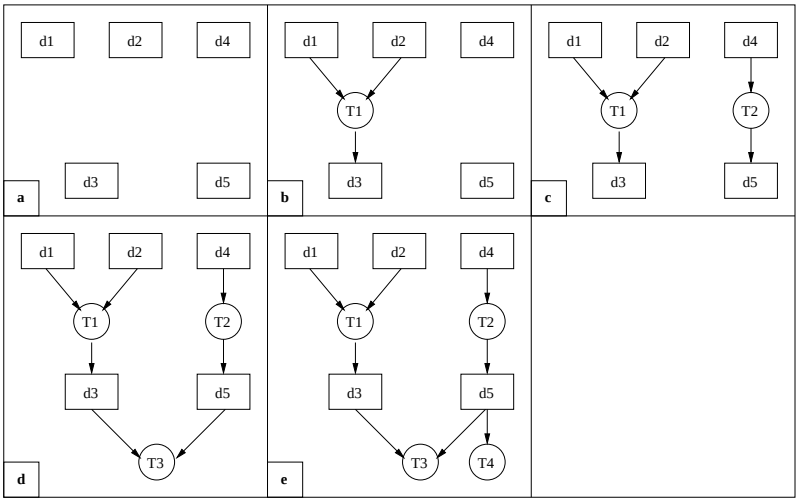


FIG. 9.2 – Construction dynamique du graphe de flot de données.

Ce graphe de flot de données définit le travail à exécuter. La réalisation de ces tâches est découplée de la construction : à tout moment un *thread* de calcul peut exécuter l'une d'entre elles alors que le graphe n'est pas totalement généré. En particulier, la sémantique d'une tâche d'une application récursive (série-parallèle) peut modifier le graphe en en créant (*fork*) d'autres.

L'ordre dans lequel sont insérées les tâches représente un ordre valide d'exécution [41] : c'est l'ordre séquentiel d'exécution défini par l'utilisateur et il est nommé RFO (pour *ReFerence Order*). Cette séquence de tâches est la pierre angulaire à la représentation interne du graphe DFG et par conséquence à l'exécution distribuée de l'application.

9.2.1.3 Exécution suivant le flot de données

Pour exécuter les tâches, les dépendances de données exposées par le graphe doivent être respectées. Pour cela, à tout moment, KAAPI doit être capable de déterminer les tâches « *prêtes* »

à être exécutées. La définition d'une tâche prête repose sur la production effective de ses données d'entrée (paramètres en lecture ou modification). Pour cela, un état est affecté à chaque donnée :

- Produit : tous les nœuds tâches prédécesseurs au nœud donnée ont terminé leur exécution ;
- Non produit : au moins l'un des nœuds tâches prédécesseurs au nœud donnée n'a pas terminé son exécution.

Considérons, pour l'instant, qu'il existe un mécanisme permettant d'exécuter les tâches en concurrence. La détection des tâches prêtes à l'exécution repose sur l'état des nœuds données. Ainsi, une tâche dont toutes les données d'entrée sont produites est prête à être exécutée. D'après cette définition, les données en écriture (en sortie) ne participent pas au calcul de l'état « *prêt* » d'une tâche. De plus, KAAPI fait l'hypothèse que l'ordre de création RFO est un ordre valide d'exécution : les premières tâches sont ainsi considérées comme prêtes.

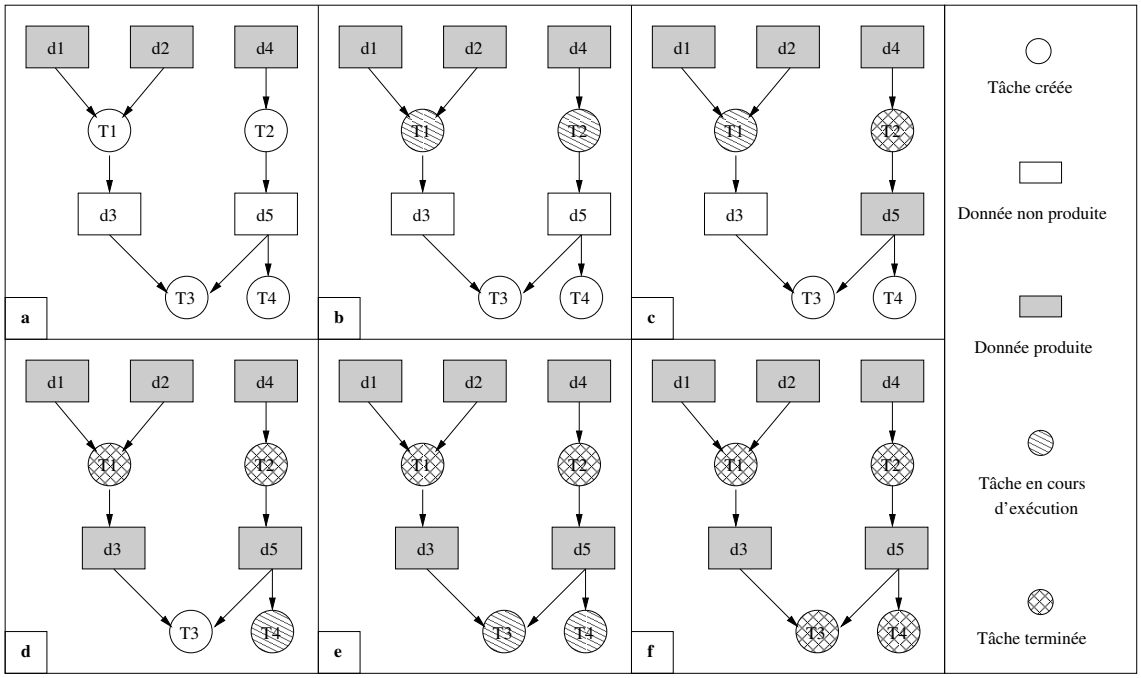


FIG. 9.3 – Exécution du graphe DFG selon les contraintes de flots.

La figure 9.3 illustre l'évaluation de l'application de la figure 9.1 suivant la représentation DFG : a) initialement, les données $d1$, $d2$ et $d4$ sont produites, b) Les tâches $T1$ et $T2$ sont calculées comme prêtes et exécutées, c) la terminaison de la tâche $T2$ produit la donnée $d5$ qui passe son état à *Produit*, d) la tâche $T4$ est calculée comme prête et exécutée ; parallèlement, la tâche $T1$ se termine et produit la donnée $d3$, e) La tâche $T3$ est exécutée ; f) toutes les tâches sont terminées, le calcul est terminé.

En pratique, cette représentation ne se trouve que sur une seule entité de calcul, nommée *thread*, qui exécute les tâches dans l'ordre RFO. L'ordonnancement dynamique permet de distri-

buer la charge de calcul en extrayant les tâches prêtes par les *threads* inactifs.

9.2.2 Ordonnancement dynamique

Au début de cette thèse, KAAPI ne proposait qu'un module d'ordonnancement basé sur la technique de vol de travail [42, 16]. Au démarrage de l'exécution, tous les *threads* sont dans un mode de calcul mais seul le *thread* principal possède la description du calcul. L'implémentation dans KAAPI de l'heuristique par vol de travail transforme[2] un *thread* de calcul en un *thread* voleur dans deux situations :

- le *thread* n'a plus de tâche à exécuter ;
- la tâche à exécuter suivant l'ordre RFO n'est pas dans un état *Créé* ou *Terminé*.

Ainsi, une double condition permet l'exécution des tâches. La première est basée sur la définition d'une tâche prête (tous ses paramètres d'entrée ont été produits) et la seconde nécessite qu'elle soit dans un état propre à l'exécution ou alors qu'elle soit déjà exécutée. L'ensemble des états possibles affectables à une tâche est donc le suivant :

- Créée : la tâche est créée. Le moteur exécutif calcule, en fonction des états des données en entrée, si celle-ci est prête.
- Exécution : la tâche est en cours d'exécution.
- Terminée : l'exécution de la tâche est terminée.
- Volée : la tâche est volée.

Lorsque le flot d'exécution rencontre une tâche non prête (*i.e.*, dans l'état *Volée*), le *thread* se suspend et entame une procédure de vol. Avant de chercher à voler une tâche, cette procédure tente à réveiller un *thread* suspendu et potentiellement exécutable. Si tel est le cas, il est réveillé et son exécution reprend, sinon un vol de tâche est initié. La définition du moteur exécutif KAAPI impose un invariant sur le nombre de *threads* actifs à un instant t sur un nœud de l'architecture matérielle. Ce nombre est fixé comme paramètre en démarrage de l'application. Par exemple, si ce nombre est fixé à 2, il existe à tout instant deux *threads* actifs qui, soit exécutent des tâches, soit cherchent à voler du travail.

Deux politiques de vol, dont la réalisation est identique, sont implantées. Dans un premier temps, un *thread* voleur tente de voler localement une tâche prête. Ainsi, un parcours de son propre graphe est entrepris afin de trouver une tâche dont toutes les dépendances de données sont respectées. S'il n'en trouve pas, une requête est initiée à destination d'un processeur distant.

A la réception d'une requête de vol, la première tâche prête selon l'ordre de référence est volée et marquée comme telle. Au retour du vol, la tâche passe dans l'état *Terminée* et le flot d'exécution suspendu peut reprendre l'exécution où il l'avait laissé. Bien entendu, les tâches *Terminée* (celles volées et terminées) ne sont pas ré-exécutées.

La figure 9.4 illustre le comportement d'un *thread* de calcul où le nombre de *threads* par nœud est fixé à 1 :

[2] Cette transformation est détaillée ci-après

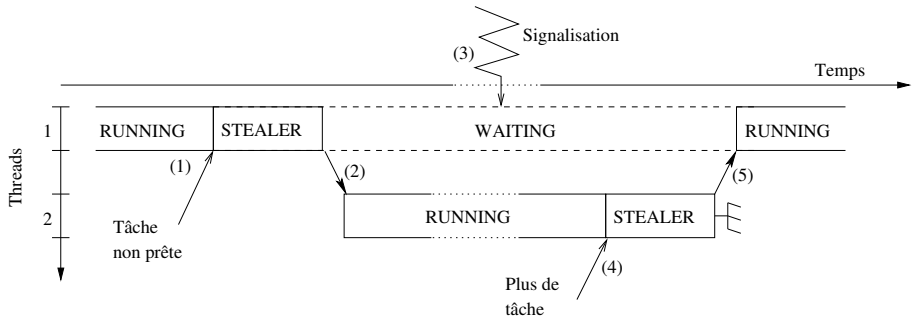


FIG. 9.4 – Modèle d'exécution et vol de travail. Lorsque le flot d'exécution rencontre une tâche non prête, un *thread* voleur est instancié afin d'exploiter les ressources de calcul.

1. lorsque le *thread* principal A rencontre une tâche non prête $T_{\text{Volée}}$, il crée un *thread* voleur 2 qui cherche à voler localement ou à distance une tâche prête ;

2. si la requête de vol aboutit, le *thread* voleur 2 crée un *thread* de calcul secondaire 3 en charge d'exécuter la tâche volée, puis se termine ;

3. lors de l'exécution de la tâche volée, la signalisation de fin de calcul intervient changeant l'état de $T_{\text{Volée}}$ à *Terminée*, débloquant alors l'exécution du *thread* de calcul principal 1 lors de la prochaine requête de vol ;

4. selon notre hypothèse, il ne peut exister qu'un seul *thread* de calcul actif à tout instant. Ainsi, le *thread* 1 reste suspendu tant que le *thread* secondaire 3 n'a pas terminé son exécution. Lorsqu'il n'a plus de tâches à exécuter, il passe en mode voleur. Ce mode tente de réveiller un *thread* suspendu et exécutable.

5. en l'occurrence, le *thread* 1 répond à cette condition. Il est alors réactivé et poursuit ainsi son exécution.

Un ordonnancement par vol de travail offre de bonnes garanties de performances et propose une qualité de répartition de charge intéressante sur des architectures à mémoire partagée [16]. Toutefois, sur une architecture à mémoire distribuée, cette technique est entachée par le surcoût dû aux requêtes de vol qui peuvent générer de nombreuses communications. Pour être efficace, le nombre de vol à l'exécution doit être faible. C'est pourquoi, son champ d'utilisation est souvent restreint aux applications de type série-parallèle. En effet, de par leur schéma d'exécution sous la forme d'une arborescence, [16] montre que le nombre de vols est très faible devant le nombre de tâches exécutées. Dans ce cas, une tâche génère tout le travail développé par sa descendance. Plus la tâche volée est haute dans l'arborescence, plus le travail récupéré est important, ce qui limite d'autant plus les situations de famine qui génèrent des requêtes de vol.

9.2.3 Support exécutif

Dans le modèle de graphe de flot de données présenté, les variables de l'application sont en assignation unique (une seule valeur est affectée à la variable utilisateur). Le support exécutif apporte une solution efficace au problème d'allocation/libération mémoire des différentes valeurs affectées pour une même variable. Cette propriété pourra directement être utilisée pour le problème que nous avons soulevé en section 6.5, page 108.

9.2.3.1 Représentation interne du graphe DFG et gestion des anti-dépendances

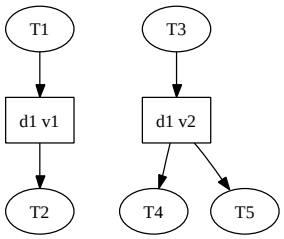
KAAPI exécute une application selon les contraintes exposées par le graphe de flot de données d'une application. Le graphe de flot de données représente les données et non les variables définies par l'application. Toutefois, un concepteur d'applications spécifie des variables sur lesquelles les tâches effectueront leur traitement et il est libre de les réutiliser. Cette approche de programmation est commune dans la conception d'applications mais extrêmement problématique dans le cas d'application distribuée : des anti-dépendances sont alors créées.

Début du code

```
void main(){
  a1::Shared<int> d1;
  ...
  a1::Fork<T1>() (d1); //Accès en écriture à la donnée d1
  a1::Fork<T2>() (d1); //Accès en lecture à la donnée d1
  a1::Fork<T3>() (d1); //Accès en écriture à la donnée d1
  a1::Fork<T4>() (d1); //Accès en lecture à la donnée d1
  a1::Fork<T5>() (d1); //Accès en lecture à la donnée d1
  ...
}
```

Fin du code

(a)



(b)

FIG. 9.5 – Anti-dépendances et représentation DFG. La représentation formelle du graphe DFG (b) de l'application (a) nécessite d'identifier les versions des variables.

Le repliement des données sur les variables utilisateur pose un problème de performance critique. Considérons le programme Athapascan présenté en figure 9.5 (a) et sa représentation DFG en (b), la variable $d1$ est décomposée en deux données $d1\ v1$ et $d1\ v2$ (ou deux versions). Une version représente la valeur affectée à une variable ainsi que sa durée de vie. Elle est créée dès qu'une tâche accède à la variable en écriture (ou modification) et sa durée de vie est égale à la portée de l'ensemble des tâches la consommant. Soit sur l'exemple présenté, les ensembles composés des tâches $\{T1, T2\}$ et $\{T3, T4, T5\}$ sont totalement concurrents alors qu'ils utilisent la même variable $d1$. Dans la suite de ce manuscrit, nous considérons que les termes « donnée » et version sont équivalents, une « *variable* » désigne un objet qui possède plusieurs données au cours du temps.

La représentation KAAPI introduit une nouvelle structure afin de formaliser les accès des tâches aux variables. En fonction de leur mode (lecture, écriture, modification), la détection des

différentes versions (et donc des données au sens DFG) est possible. La représentation de l'application de la figure 9.5 (a) dans KAAPI est définie en figure 9.6.

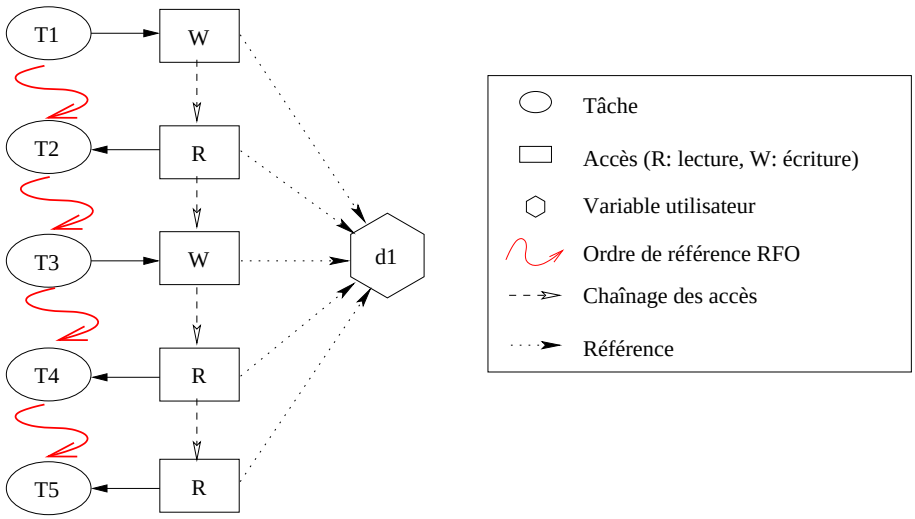


FIG. 9.6 – Représentation interne dans KAAPI du graphe DFG de l'application présentée en figure 9.5 (a).

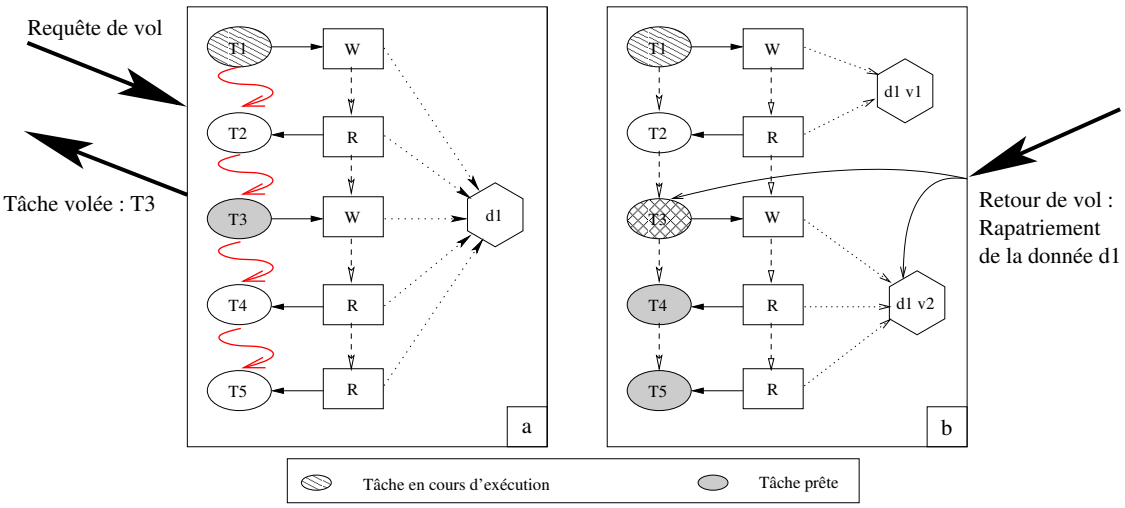


FIG. 9.7 – Gestion des variables utilisateurs et des données du graphe DFG lors du vol de travail. Une requête de vol intervient au début de l'exécution et vole la seule tâche prête $T3$ (a). Au retour du vol, une nouvelle version est allouée afin de ne pas violer la cohérence de l'exécution de $T2$ (b). La version $d1v1$ sera détruite à la fin de l'exécution de $T2$.

Cette représentation ne permet pas *a priori* de lever les ambiguïtés sur les anti-dépendances.

Notons que l'exécution suivant l'ordre de création des tâches (RFO) est un ordre valide avec la sémantique attendue sur les données lues. Lors de l'opération de vol, cette représentation est modifiée. Reprenons alors la représentation KAAPI du graphe DFG de l'application de la figure 9.5 (a). La figure 9.7 illustre la modification de la représentation interne KAAPI permettant d'obtenir une exécution concurrente. Une requête de vol intervient lors de l'exécution de la tâche $T1$ (a). Au regard des contraintes de dépendance, seule la tâche $T3$ est prête. Le retour de vol se produit alors que la tâche $T1$ est toujours en cours d'exécution (b) (l'exécution de la tâche $T3$ est terminée sur le site distant et les données produites sont retournées au *thread* volé). Le traitement associé au signal de fin de vol vérifie, d'une part s'il est nécessaire de créer une nouvelle version de la variable et, d'autre part, affecte l'état de la tâche $T3$ à *Terminée*. Tant que la tâche $T2$ n'est pas terminée, il est indispensable d'allouer une nouvelle donnée (ce qui est le cas dans l'exemple présenté). La fin d'exécution de la tâche $T2$ entraînera la destruction de la version $d1$ $v1$ qui n'a plus lieu d'être étant donné qu'elle n'est plus référencée. Remarquons également qu'à présent les tâches $T4$ et $T5$ sont prêtes.

A travers cette technique de gestion des versions des variables, KAAPI traite d'une manière élégante et optimisée la gestion de l'espace mémoire tout en levant les anti-dépendances pouvant être exposées par une application.

A présent, descendons encore dans la hiérarchie KAAPI et observons les techniques utilisées afin de traiter les communications.

9.2.3.2 Messages actifs

Toutes les communications nécessitant l'intervention du réseau se basent sur des messages actifs proches des définitions énoncées dans [76, 86].

Les communications définies par les messages actifs dans KAAPI sont de type *one-side*, c'est-à-dire que seul le code utilisateur de l'émetteur explicite l'envoi des messages. Du côté du récepteur, la couche de communication de KAAPI se charge de recevoir le message et d'appeler une méthode implantée par l'utilisateur. Cette méthode est libre d'effectuer le contrôle adéquat (par exemple, la mise à jour d'une condition pouvant libérer l'exécution d'un *thread*) et le cas échéant d'extraire les données dans l'espace mémoire utilisateur.

Sur l'émetteur, le processus qui réalise l'appel n'est pas, *a priori*, bloqué : une fois le message émis au système, le processus continue son exécution en concurrence avec la communication du message. Durant l'envoi, le processus émetteur doit garantir l'intégrité des données du message jusqu'à la fin de l'émission qui est signalée par la couche de communication à l'application au moyen d'une méthode *callback*.

L'utilisation de l'API Athapascan masque ces méthodes au développeur d'application. Nous prenons le temps de détailler cette technique car notre extension emploie ce support de communication pour le recouvrement des communications par du calcul. La couche de communication de KAAPI a fait l'objet d'un sujet de thèse [73]. Entre autre, il est montré que le recouvrement des communications est bien mieux traité que dans les autres *middleware* tels que omniORB (CORBA) ou MPICH (MPI).

La figure 9.8 définit la manière dont est réalisé l'envoi d'un message actif :

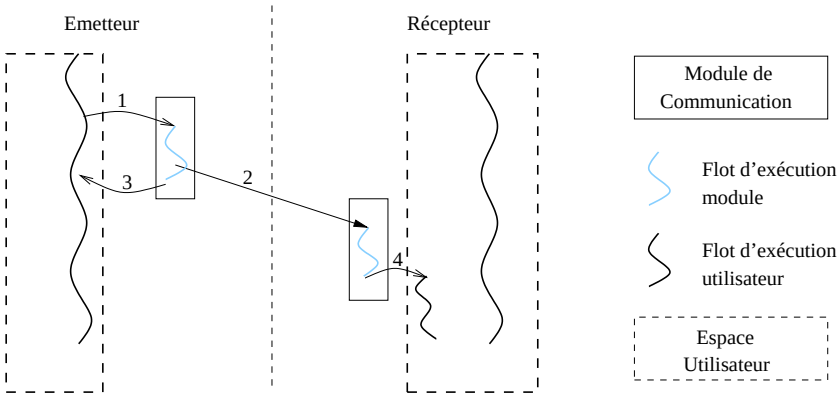


FIG. 9.8 – *Message actif* : l'émetteur effectue l'appel au service de communication (1). Le message et les données sont transmis (2). Le processus émetteur est informé, au moyen d'un méthode *callback*, que la communication est terminée (3). Côté récepteur, le module de communication traite le message et appelle une méthode en espace mémoire utilisateur (4) en concurrence avec le flot d'exécution principal du processus récepteur.

- Le processus émetteur appelle le module de communication afin de réaliser l'envoi du message. Cet appel est non-bloquant, le flot d'exécution utilisateur continue son exécution (1).
- Le module de communication s'occupe de transmettre le message au processus concerné (2).
- Dès que les données en zone mémoire utilisateur peuvent être réutilisées (soit elles ont été bufferisées, soit elles ont été copiées dans le tampon d'émission), une méthode *callback* (3) est appelée : cette méthode doit être implantée par le concepteur de l'application.
- A la réception du message, le module de communication détecte son arrivée et appelle une méthode utilisateur (4) chargée d'extraire les données dans la zone mémoire de l'application. L'exécution de cette méthode se fait en concurrence avec le flot d'exécution principal du processus récepteur.

9.3 Extension pour les simulations itératives

Dans le chapitre précédent, nous avons résolu le problème des anti-dépendances *via* l'insertion de barrières de synchronisation qui permettent de garantir l'assignation unique des variables. Cependant, cette solution bride le parallélisme de l'application.

Nous venons de présenter la manière dont KAAPI permet de lever ces fausses dépendances tout en optimisant la réutilisation de l'espace mémoire. Toutefois, l'algorithme d'ordonnancement implémenté reste limité aux applications récursives (ou séries-parallèles). Dans notre cas d'application où des simulations itératives sont traitées, l'écriture récursive de l'application n'est pas naturelle. Exécutée par le moteur KAAPI qui implante une technique de placement par vol de travail, notre application introduirait de trop nombreux vols, rendant l'exécution distribuée inefficace. Une technique d'ordonnancement statique est plus appropriée pour l'obtention de performances sur une architecture parallèle.

9.3.1 Intégration d'ordonnanceurs statiques

Le schéma d'exécution (présenté en section 4.2 page 74) suivant une approche statique est caractérisé par cinq traitements successifs permettant de définir les informations nécessaires à une exécution distribuée. La figure 9.9 présente, sous la forme d'un graphe, les dépendances entre ces traitements.

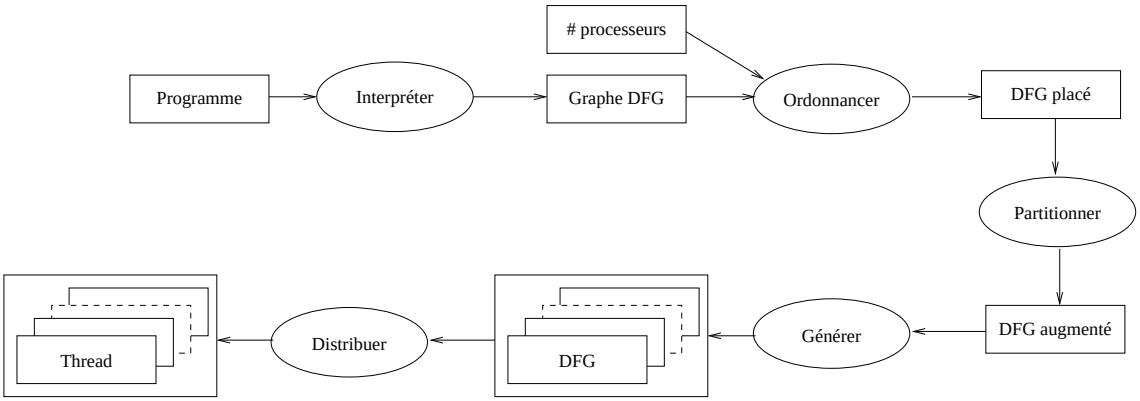


FIG. 9.9 – Graphe des traitements nécessaires au démarrage d'une exécution distribuée selon une technique d'ordonnancement statique.

Ce pré-traitement a pour objectif de générer autant de flots d'exécution que de processeurs physiques. Dans l'approche choisie, ces flots d'exécutions doivent être autonomes dans le sens où aucun processus centralisé ne prend part à l'exécution. Ceci se traduit principalement par l'introduction de tâches de communication où émetteurs et récepteurs sont identifiés.

La suite de cette section présente les fonctionnalités de chacun de ces traitements.

9.3.1.1 Construction de la représentation abstraite

KAAPI réalise la construction du graphe dynamiquement, c'est-à-dire que l'exécution des tâches peut débuter alors le graphe n'est pas complètement défini. Afin d'avoir une connaissance complète du graphe avant exécution, une unique tâche est insérée dans le graphe dont son exécution insère les tâches de l'application sans pour autant les exécuter. L'ensemble des tâches et des précédences ainsi construit sera donné en entrée des algorithmes d'ordonnancement.

9.3.1.2 Ordonnancement

L'étape d'ordonnancement, telle que nous l'avons définie dans KAAPI, affecte un site logique d'exécution à chaque tâche du graphe construit à l'étape précédente. L'ensemble des tâches affecté à un même site forme une partition et chaque numéro de partition représente alors un site logique. Par la suite, nous emploierons le terme numéro de partition afin de désigner ce site logique, évitant toute confusion avec le site physique d'exécution.

Le calcul des partitions repose sur l'utilisation d'ordonnanceurs qui nécessitent la transformation de la structure DFG KAAPI en une représentation adéquate à leur utilisation.

A l'heure actuelle, nous avons intégré trois algorithmes d'ordonnancement à KAAPI :
- DSC requiert un graphe de précédence où, pour chaque tâche, la liste des successeurs directs est exigée. La structure de données est présentée en section 7.2.2, page 116.
- ETF[3] nécessite également un graphe de précédence qui, à l'inverse de DSC, se présente sous la forme d'une liste de prédécesseurs directs. La structure de données en entrée de ETF définit, dans un premier temps, les tâches identifiées par un entier i pour lesquelles un coût d'exécution est associé. Dans un second temps, les relations de précédences sont définies sous la forme de triplets ($i_{\text{Préd}}$, i, Coût) qui impliquent que la tâche $i_{\text{Préd}}$ précède l'exécution de la tâche i avec un coût de communication de « *Coût* ».
- METIS[4] est basé sur un graphe de dépendance. La structure de données demandée est de type CSR (*Compressed Storage Format*) [68].

La représentation DFG offre toutes les informations nécessaires à la construction des graphes de précédence (noté *PG*) et de dépendance (noté *DG*). Nous avons vu en section 4.3, page 75, qu'il est possible de passer d'une représentation DFG à PG ou DG.

Une fois cette structure construite, l'algorithme d'ordonnancement approprié est invoqué. L'algorithme retourne, pour chaque tâche, un numéro de partition (*i.e.* le site logique). Pour compléter cette étape, il suffit d'affecter, à chaque tâche du graphe DFG KAAPI, le numéro de partition calculé par l'ordonnanceur.

9.3.1.3 Partitionnement

L'étape de partitionnement identifie les schémas de type « 1 *écrivain* / n *lecteurs* » sur les variables du graphe DFG. En pratique, ce traitement est équivalent à scanner la liste des accès à une donnée et à identifier toutes les versions d'une variable. Cette identification permet d'insérer, dans le graphe DFG, des tâches spécifiques liées aux communications des données entre les différentes partitions. La figure 9.10 représente le graphe de flot de données augmenté (b) du graphe original proposé en (a).

Le traitement nécessaire à l'insertion de ces tâches est appliqué sur chaque version de chaque variable. Les partitions différentes de la partition du producteur sont alors identifiées. Si l'ensemble calculé E_{diff} n'est pas vide, une tâche de diffusion et n tâches de réception sont insérées dans le graphe où n est le cardinal de E_{diff}. Ainsi, deux tâches spécifiques liées au caractère distribué et statique de l'exécution sont définies :
- *BroadcastTask* : cette tâche prend la version produite en lecture et sa réalisation diffuse la donnée à l'ensemble des sites la consommant (la relation de correspondance site physique/partition fait l'objet de l'étape de distribution). Elle est placée dans la partition ayant produit la donnée et est insérée, dans l'ordre RFO, juste après la tâche qui la produit.
- *ReceiveTask* : à chaque tâche *BroadcastTask* est associée un ensemble de tâches de réception *ReceiveTask*. Une tâche de réception est insérée dans chaque partition qui lit la version, juste avant la tâche nécessitant la donnée en lecture. L'accès à la donnée est de type écriture.

[3] Algorithme développé au sein du laboratoire LIG par J. Pecero de l'équipe MOAIS
[4] Téléchargeable à l'adresse http://www.cs.umn.edu/~metis

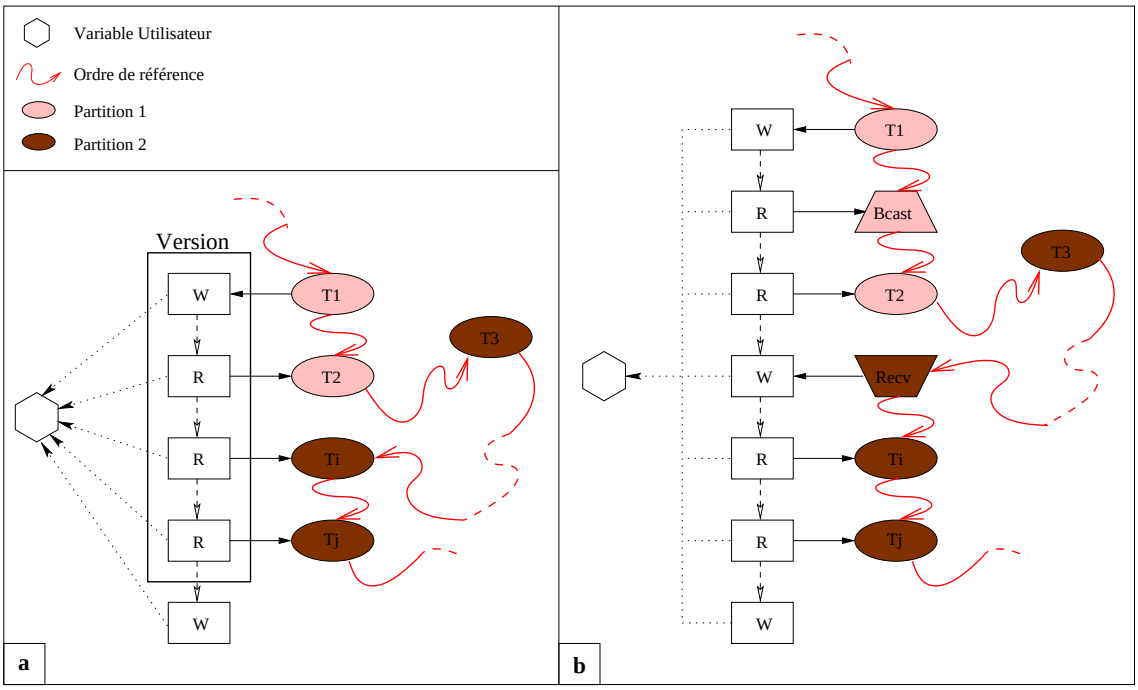


FIG. 9.10 – Partitionnement d'un graphe de flot de données. Le graphe initial placé (a) est augmenté des tâches de communication (b). Le graphe mise à jour est le graphe initial complet.

Il existe une relation d'association entre tâches d'envoi et de réception qui permet d'identifier, de manière unique, la communication à travers un *tag*.

A la fin de cette étape, le graphe DFG est alors augmenté des tâches de communication (*BroadcastTask* et *ReceiveTask*). Nous verrons dans la section suivante 9.3.2 que cette étape ajoute également une tâche de contrôle liée à l'utilisation de la couche de communication non-bloquante par messages actifs.

Le traitement présenté est la pierre angulaire à la définition des flots distribués d'exécution. Bien que chaque tâche soit affectée à une partition particulière, il n'existe encore qu'une seule représentation complète des sous-graphes DFG. L'étape suivante permet de générer les k partitions sous la forme de k sous-graphes.

9.3.1.4 Génération

La figure 9.11 présente les deux sous-graphes construits après l'étape de génération à partir du graphe DFG initial de la figure 9.2 où les tâches $T1$ et $T3$ appartiennent à la partition 1 et les tâches $T2$ et $T4$ représentent la partition 2.

La dernière étape vise alors à déployer ces graphes sur les divers processeurs de l'architecture

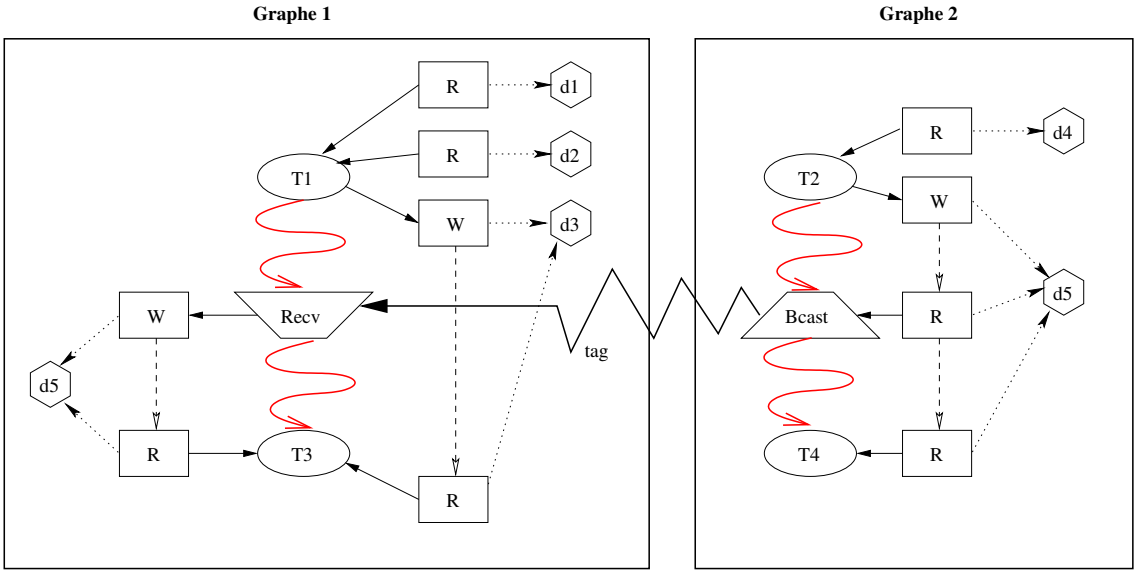


FIG. 9.11 – Graphes DFG après l'étape de génération pour l'application de la figure 9.1 selon la représentation interne KAAPI. Le *tag* permet d'identifier la communication.

matérielle.

9.3.1.5 Distribution

Idéalement, les k graphes générés à l'étape précédente sont déployés sur p processeurs où $k = p$. En pratique, le nombre k de partitions peut être supérieur au nombre disponible de processeurs. Ce cas se présente lorsque l'utilisateur a intentionnellement défini un nombre de partitions plus grand que le nombre de processeurs physiques, ou lorsque des contraintes de placement ont été spécifiées sur certaines tâches.

La distribution suit une technique d'affectation cyclique où les partitions $0,\ p,\ 2p,\ \ldots$ sont attribuées au premier processeur, les partitions $1,\ p+1,\ 2p+1,\ \ldots$ au second et ainsi de suite. La question de définir une politique de distribution plus « *intelligente* » est soulevée. Par exemple, si l'architecture matérielle est hybride (multi-ordinateurs de multi-processeurs), le choix d'affecter les partitions par affinité de communication (celles qui communiquent beaucoup) sur le même multi-processeurs peut être envisagé. Toutefois, cette politique d'affectation nécessite un descriptif de l'architecture matérielle complète, information qui n'est pas actuellement disponible au sein de l'environnement KAAPI.

A réception des partitions, une méthode est activée réalisant un traitement local sur le sous-graphe obtenu. Il permet, entre autre, de construire une table de relation entre les *tags*, identifiant les communications, et les tâches à activer en réponse à celle-ci. Ce traitement étant réalisé, le graphe DFG causalement connecté à l'exécution, est attaché à un *thread* de calcul.

La représentation augmentée construite tout au long des cinq traitements caractérise le mouvement des données entre les *threads* d'exécution. Toutefois, la réalisation d'une exécution concurrente est étroitement liée à la gestion des synchronisations. Par exemple, une tâche ne peut s'exécuter tant que les données consommées ne sont pas disponibles localement. La section suivante présente les techniques implémentées permettant de manier les synchronisations liées, d'une part, à l'ajout des tâches de communication et, d'autre part, au caractère centralisé de la gestion du cycle de vie l'exécution distribuée.

9.3.2 Support exécutif

Selon le modèle d'exécution défini dans KAAPI, la gestion des synchronisations est majoritairement traitée à travers les contraintes de flot où une tâche est prête dès que tous ses paramètres en entrée ont été produits. Toutefois, avant l'exécution d'une tâche, le moteur vérifie si elle est dans un état propre à l'exécution : c'est-à-dire, dans l'état *Créé*. Si elle est déjà terminée, la tâche est ignorée et la prochaine, dans l'ordre RFO, est traitée. Si elle est dans un autre état, le *thread* se suspend, et la recherche d'une tâche prête est initiée.

Basée sur cette méthode d'exécution, nous avons alors inclu un cinquième état, appelé « *Attente* » associable aux tâches dont le diagramme d'états est présenté en figure 9.12.

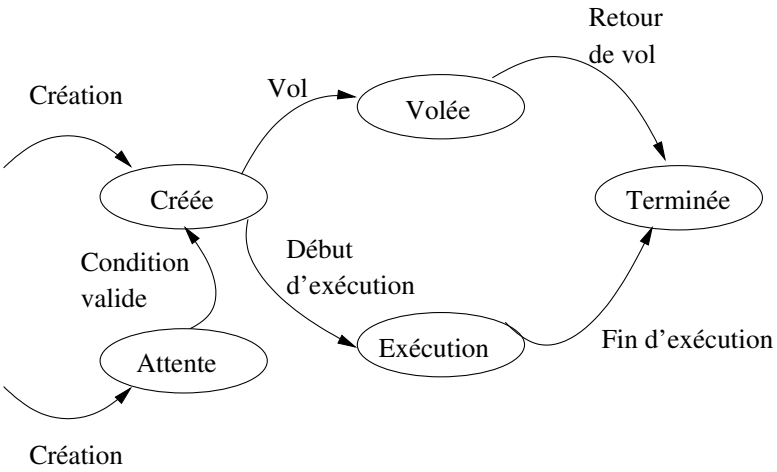


FIG. 9.12 – Diagramme d'états complet d'une tâche KAAPI.

L'état *Attente* est similaire à l'état *Volée*. En effet, la tentative d'exécution d'une tâche dans l'état *Volée* provoque le blocage du flot d'exécution sur une attente conditionnelle qui est validée lors du retour de vol. Dans le cas d'une exécution statique, cet état est utilisé afin de bloquer le flot d'exécution sur la réception d'une communication. La tâche *ReceiveTask* est alors créée dans un tel état. La condition de transition à l'état *Créée* repose sur la réception de la donnée. Une seconde tâche utilise également cet état : *WaitTask*. Elle est insérée dans le graphe du *thread* émetteur afin de manier le caractère non bloquant de l'émission d'une donnée.

A travers l'introduction de ce nouvel état, définissons plus en détails la gestion des synchronisations. Celles-ci se présentent sous deux formes. La première gère le flot de contrôle de l'exécution distribuée et la seconde caractérise le besoin de synchronisation lié à l'échange de données lors de la réception.

9.3.2.1 Contrôle global de l'exécution distribuée

La gestion de l'exécution distribuée d'une application selon une technique d'ordonnancement statique est majoritairement traitée par les *threads* distribués : l'exécution est décentralisée. Toutefois, une entité globale de contrôle pilote l'exécution distribuée.

L'introduction de ce contrôle est utilisé également dans un but de synchronisation du lancement de l'exécution. Cette synchronisation assure que chacun des *threads* participant à l'exécution est créé et peut traiter les requêtes qui lui sont destinées. Cela permet d'éviter les situations où un processus exécute une tâche de diffusion dont le destinataire est un processus n'ayant pas encore reçu la description de son flot d'exécution.

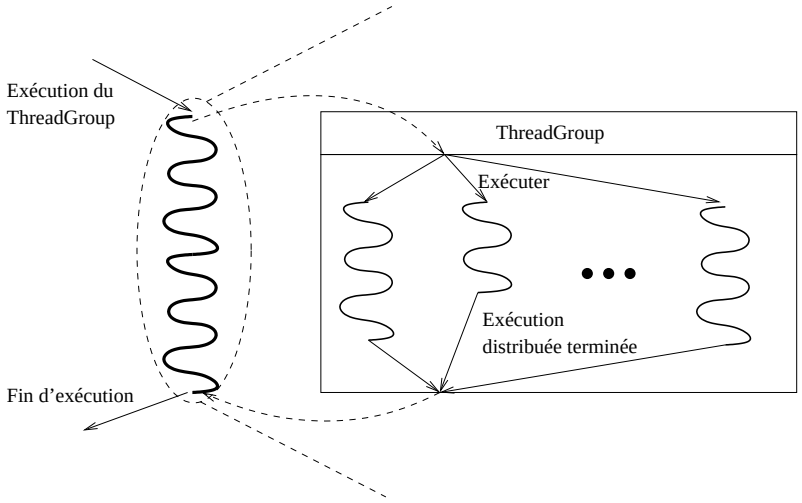


FIG. 9.13 – L'objet distribué *ThreadGroup*. Toutes les opérations nécessitant l'intervention de l'ensemble des *threads* font appel au *ThreadGroup*. Sa fonction première est de contrôler l'exécution distribuée résultant du partitionnement de la tâche initiale.

Ainsi, le début et la fin de l'exécution sont soumis à un contrôle. La gestion de ce contrôle est définie par l'entité *ThreadGroup*. Cet objet permet de piloter l'ensemble des *threads* participant à l'exécution et est créé suite à l'ordonnancement d'un graphe. La fin de l'exécution de chaque *thread* distant informe le *ThreadGroup* de la fin d'exécution. La figure 9.13 représente schématiquement le fonctionnement de cet objet. L'exécution d'un *ThreadGroup* est considérée comme une opération de calcul bloquante qui, en réalité, délègue la totalité du travail à l'ensemble des *threads* distribués.

9.3.2.2 Synchronisation et envoi de messages

La réalisation des communications, bien que définie au moyen de tâches, nécessite la gestion d'une certaine forme de synchronisations. En effet, la tâche de réception (*ReceiveTask*) ne rentre pas dans la définition de l'exécution selon les contraintes de flot de données. Une telle tâche prend un accès en écriture sur la donnée devant être reçue. Les tâches de réception sont créées dans un état *Attente*. La réception de la donnée passe l'état de la tâche à *Créée*. Le traitement associé à l'extraction de la donnée du réseau détecte s'il est possible de réutiliser la zone mémoire de la dernière version de la variable. Dans le cas contraire, une nouvelle version (et donc une nouvelle allocation mémoire) est nécessaire.

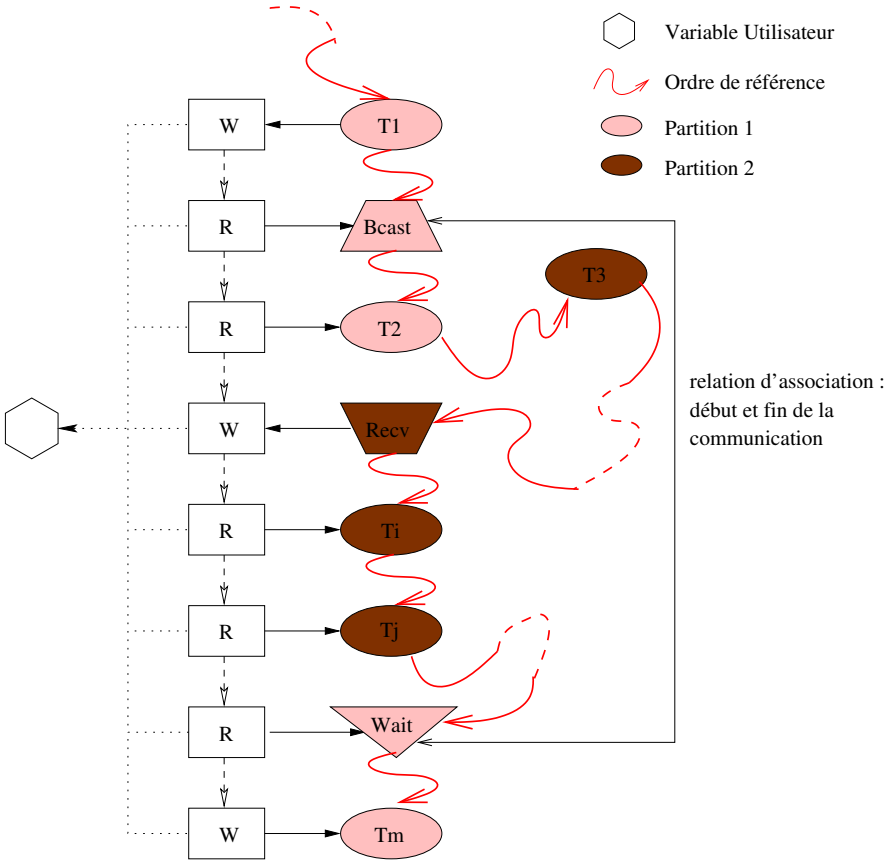


FIG. 9.14 – Insertion des tâches de contrôle garantissant la cohérence du flot d'exécution lors de l'émission des données. La tâche *WaitTask* est insérée juste avant la prochaine version et est initialisée dans un état *Attente*.

Une tâche d'envoi ne nécessite pas la modification de l'état par défaut (*Créée*). Elle accède à la donnée à envoyer en lecture et les contraintes de flot de données suffisent à garantir que la bonne donnée sera transmise. Cette tâche est calculée comme prête en suivant l'évolution des contraintes

de flot de données. Toutefois, l'envoi de données se base sur la couche de transport réseau KAAPI définissant une communication non-bloquante. Avant d'affecter de nouveau la variable, il est nécessaire de s'assurer que la communication est terminée. Une nouvelle tâche de contrôle est insérée dans le graphe de flot de données : *WaitTask*. Elle permet de bloquer le flot d'exécution tant que la donnée n'est pas effectivement réutilisable. Elle est insérée dans la même partition que la tâche d'émission *BroadcastTask* juste avant la prochaine écriture (avant la prochaine version). Le graphe augmenté de ces tâches est présenté en figure 9.14 (a). L'état de la tâche est initialisé à *Attente* et sera passé à *Créée* lors de l'appel de la méthode *callback* associée à l'émission, comme décrit dans la section 9.2.3.2.

Jusqu'à présent, nous avons défini les étapes permettant de construire des *threads* d'exécution distribués et la façon dont ceux-ci se coordonnent pour la réalisation d'une application. L'exécution d'applications itératives s'appuie sur ce support exécutif et permet de ré-exploiter les cinq étapes présentées en section 9.3.1.

9.3.3 Applications itératives

Nous considérons comme applications itératives, les programmes qui exposent le même schéma d'exécution (le même graphe DFG) d'une itération à l'autre. L'étude des heuristiques d'ordonnancement pour ces applications (on parle d'ordonnancement cyclique) est complexe. L'approche actuellement considérée est pragmatique dans le sens où l'ordonnancement est réalisé en déroulant la boucle sur quelques itérations. Le diagramme d'exécution qui en découle forme le graphe de tâches qui sera soumis à un ordonnanceur statique tel que ceux intégrés dans KAAPI. De ce fait, notre approche s'oriente sur la définition d'un environnement d'exécution efficace.

Dans ce qui suit, nous supposons que l'ensemble des *threads* de calcul sont distribués et prêts à être exécutés. Le graphe DFG initial distribué correspond au développement d'une ou plusieurs itérations de l'application. Par la suite, nous nommons *étape*, un ensemble d'itérations de l'application répété par KAAPI.

A partir de ces conditions, les deux problèmes dus à l'exécution totale (*i.e.* toutes les itérations) de l'application sont les suivants :

1. L'exécution d'un graphe DFG, dans l'environnement KAAPI, « *consomme* » les tâches le composant. Ainsi, d'une étape à l'autre, il est nécessaire de reconstruire le graphe DFG.
2. Les données produites à la fin de l'étape i doivent être redistribuées pour l'étape $i + 1$ selon les contraintes de flot et le placement des tâches.

La solution à 1. consiste à sauvegarder, avant toute exécution de tâches applicatives, les invariants de chaque représentation du calcul à réaliser par chaque entité distante. D'une étape à l'autre, seule l'adresse mémoire des données peut évoluer (de par la gestion des anti-dépendances) si bien que l'ensemble, composé de l'état des tâches et celui des accès, forme cette sauvegarde. La fin de l'exécution d'une étape entraîne la restauration en mémoire du graphe sauvegardé. Toutefois, il ne peut débuter l'exécution de l'étape suivante tant que tous les autres *threads* n'ont pas terminé l'étape courante. Cette barrière simplifie le processus de réception lié à la tâche *ReceiveTask*. Une solution envisageable à cette barrière est basée sur l'utilisation d'un tampon de réception permettant de bufferiser les données des futures étapes.

La solution à 2. nécessite un pré-traitement à la sauvegarde du graphe. En effet, il est nécessaire de modifier l'état de certaines tâches de calcul afin d'ajouter le contrôle nécessaire aux redistributions des données.

Début du code

```
void main(){
  a1::Shared<int> d1;
  a1::Shared<int> d2;
  ...
  a1::Fork<T1>() (d1);      //Accès en lecture sur la donnée d1
  a1::Fork<T2>() (d1);      //Accès en écriture sur la donnée d1
  a1::Fork<T3>() (d1, d2); //Accès en lecture sur les données d1 et d2
  a1::Fork<T4>() (d1, d2); //Accès en lecture sur la donnée d1 et en modification
                           //sur la donnée d2
  ...
}
```

Fin du code

FIG. 9.15 – Exemple de programme Athapascan utilisé pour présenter le calcul des redistributions.

La redistribution des données, d'une étape à la suivante, nécessite l'identification du schéma de communication qui doit être mis en place. Il est préalablement calculé lors de l'étape de partition (section 9.3.1.3). Pour cela, le calcul associé détecte les partitions qui ont des tâches dont les premiers accès aux variables sont en lecture. En outre, il passe en revue les partitions qui détiennent la dernière version de chaque variable. Pour illustrer ceci, considérons le graphe de flot de données de la figure 9.16 issu du programme de la figure 9.15.

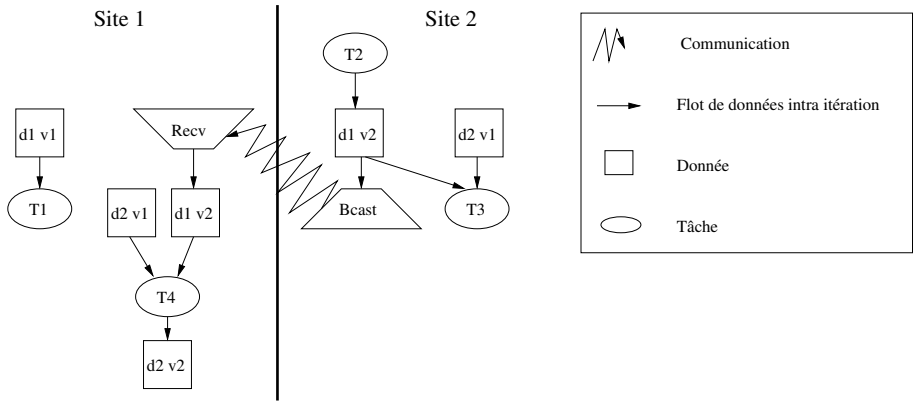


FIG. 9.16 – Représentation du graphe DFG par site d'exécution de l'application de la figure 9.15.

D'après le flot de données, la variable $d1$ n'est lue que par le site 1 à travers la tâche $T1$. Les sites qui détiennent la dernière version $d1v2$ sont formés de l'ensemble $\{1, 2\}$. Le site 2 est le producteur de cette version (tâche $T2$). Néanmoins, le site 1 détient également une copie de cette dernière ($d1$ $v2$) de par le dépendance de données entre $T2$ et $T4$. Quant à la variable $d2$, les sites lecteurs forment l'ensemble $\{1, 2\}$ (tâches $T3$ et $T4$) alors que seul le site 1 détient la dernière

version. Les sites *lecteurs* et *propriétaires* de la dernière version de chaque donnée sont reportés sur la figure 9.17 à droite.

	$d1$	$d2$
Lecteurs	1	1, 2
Propriétaires	1, 2	1

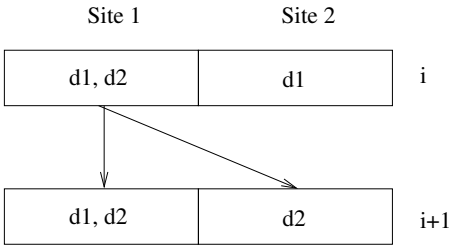


FIG. 9.17 – Ensemble des sites lecteurs et propriétaires des variables $d1$ et $d2$ du programme de la figure 9.15. Le schéma de redistribution est présenté à droite.

La différence entre les lecteurs et les propriétaires permet de détecter les données qui doivent être redistribuées. Sur l'exemple, le site 2 doit recevoir la dernière version de $d2$ ($d2$ $v2$) qui sera transmise par l'un des sites propriétaires (ici, seule le site 1 détient cette version). Le site 1 possède déjà les dernières versions des variables $d1$ et $d2$ (figure 9.17 à droite).

En pratique, la réalisation des envois nécessaires est prise en charge par le *ThreadGroup* qui ordonne à chacun des *threads* unitaires la réalisation des communications dont il a la charge. Chaque sous-graphe de flot de données est modifié afin de prendre en compte le contrôle nécessaire aux synchronisations. Les tâches de calcul, qui requièrent une donnée transmise lors de l'étape de redistribution, sont créées dans l'état *Attente* et la méthode de signalisation associée aux messages actifs passe l'état à *Prête*. En outre, l'exécution des tâches prenant une variable en écriture, alors que cette dernière prend part au schéma de redistribution, doivent s'assurer que la donnée est envoyée. Si la communication n'est pas terminée, une nouvelle version est créée.

La réalisation de cette redistribution revient à insérer des tâches d'envoi (*BroadcastTask*) et de réception (*ReceiveTask*) ainsi que de contrôle (*WaitTask*). La figure 9.18 présente la formalisation de cette méthode. Une technique de recouvrement des communications par du calcul est implémentée. Les tâches potentiellement prêtes et qui ne participent pas à la redistribution peuvent débuter leur exécution sans pour autant attendre la fin globale de l'étape de redistribution des données.

Le schéma de redistribution présenté ne prend en considération que les mouvements de données entre deux étapes. Toutefois, il existe un autre schéma qui doit être considéré : la redistribution initiale. L'étape de distribution des partitions ne diffuse pas les données avec la description de chaque sous-graphe DFG. Les données restent *en place* sur le site qui les a produites. Ainsi, l'exécution de la première étape nécessite une première distribution des données qui suit la même technique.

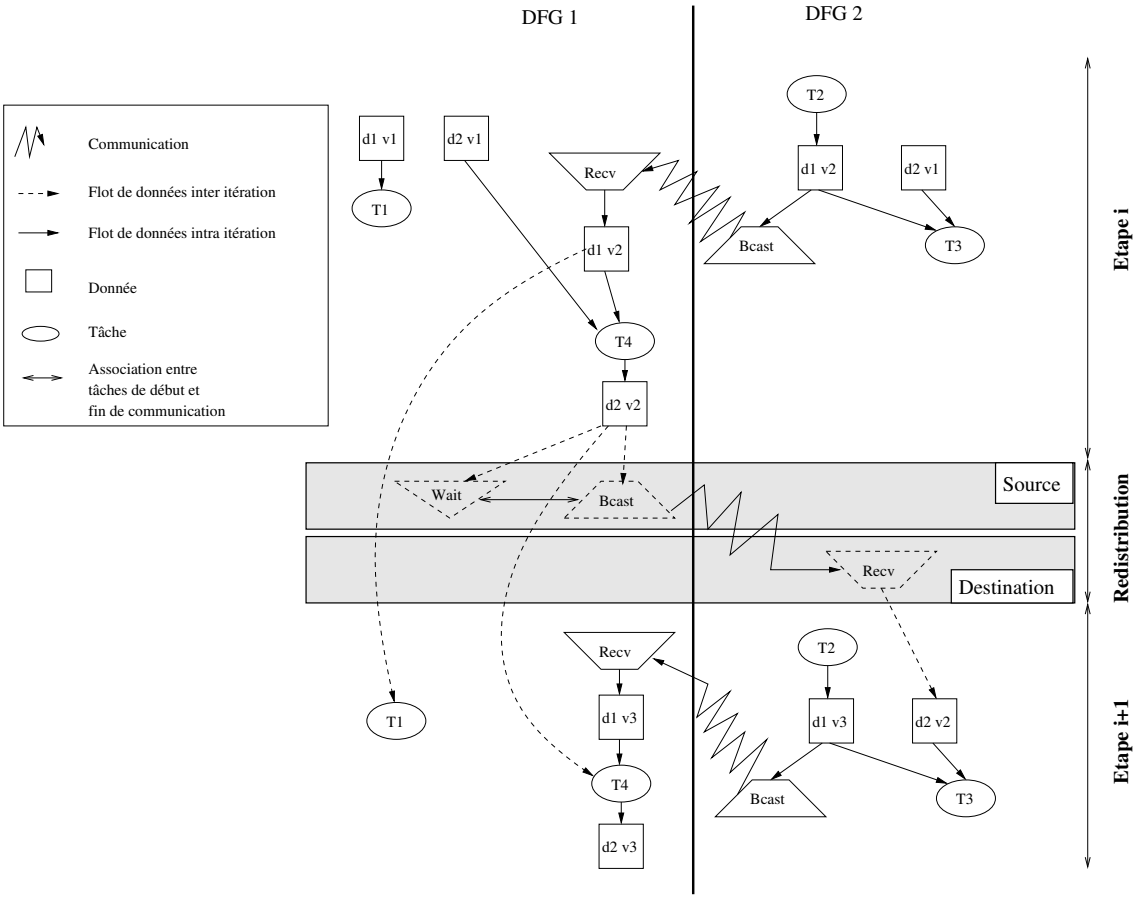


FIG. 9.18 – La redistribution revient à insérer des tâches de communication (*BroadcastTask* et *ReceiveTask*) et de contrôle (*WaitTask*) entre chaque étape.

9.4 Bilan

L'environnement de base KAAPI offre un modèle d'exécution selon les contraintes de flot d'un graphe DFG. Ce modèle d'exécution est prouvé performant [98]. Afin de traiter le caractère distribué de l'exécution, une technique d'ordonnancement dynamique par vol de travail est implémentée. Offrant de bonnes garanties de performance sur les applications de type série-parallèle, elle devient inefficace pour les autres types d'application.

Notre approche est orientée sur la spécification d'une couche « *ordonnanceur statique* » au-dessus du moteur exécutif KAAPI. Une technique statique d'exécution offre de meilleures performances pour les applications itératives. En effet, compte tenu de la connaissance complète du graphe DFG au moment du calcul de l'ordonnancement, il est possible de générer les communications strictement nécessaires à l'exécution. Toutefois, l'introduction du caractère statique nécessite une légère modification du moteur exécutif. En effet, un état *Attente* est ajouté à l'ensemble possible affectable aux tâches. Il permet de bloquer le flot d'exécution principal sur cette tâche et

de définir des synchronisations dues à des événements quelconques (en l'occurrence, dans notre cas, il s'agit des notifications de réception ou de fin d'émission d'une donnée). Cependant, cette synchronisation n'entraîne pas nécessairement une inactivité du processeur. Le *thread* passe alors en mode voleur et tente d'exécuter une tâche prête parmi les tâches qui lui restent à réaliser.

De cette manière, l'exécution d'applications itératives ordonnancées par des algorithmes dits *statiques* est possible. La partie précédente (du chapitre 7, page 122) soulève le problème de l'inclusion de barrières de synchronisation garantissant l'assignation unique des variables. Bien que l'environnement d'exécution KAAPI permette de lever le problème d'assignation unique par la création de versions de variables, l'implémentation actuelle nécessite toutefois une telle synchronisation après plusieurs itérations de l'application. La restauration du graphe d'une étape à l'autre impose cette barrière. Néanmoins, les itérations applicatives peuvent être développées offrant alors un parallélisme potentiel plus important. Dans notre approche, une étape représente une à plusieurs itérations de l'application. Le graphe DFG représente le développement d'un ensemble d'itérations. Ainsi, le gain de performance par le recouvrement des itérations peut être mesuré. Il convient de noter que cette implémentation peut être améliorée ce qui permettrait en tout état de cause de s'affranchir des barrières de synchronisation entre les étapes. Par exemple, il serait envisageable de définir un tampon de réception permettant de bufferiser les données futures produites à l'étape suivante sans pour autant influencer l'exécution de l'étape courante.

Notre approche traite principalement du placement des tâches. Un ordonnanceur, tel que ETF ou DSC, calcule également les dates de début de chaque tâche de l'application. En fonction de celles-ci, il est possible de construire un ordre d'exécution total (qui définit la politique locale d'exécution) pour chaque partition du graphe DFG. Par exemple, si deux tâches sont prêtes à un instant t, l'ordre total définit quel est la « *meilleure* » tâche à exécuter. Cette propriété qualitative repose sur les heuristiques développées par les algorithmes d'ordonnancement, où, par exemple, pour DSC, il s'agit de la tâche appartenant à la séquence dominante. Cet ordre peut avoir un impact sur les performances de l'exécution distribuée. En l'état actuel d'avancement de notre extension, cet ordre n'est pas modifié comparé à l'ordre RFO. Les expériences de la partie suivante utilisent, toutefois, la politique locale calculée par l'ordonnanceur grâce à un traitement préalable.

L'intérêt de l'approche choisie est de disposer, à tout instant, d'une représentation abstraite de l'exécution sous la forme d'un graphe de flot de données. Cette information est capitale pour des algorithmes coordonnés et optimisés de tolérance aux pannes tels que présentés dans [12, 63]

Le chapitre suivant présente un ensemble de tests qui évalue notre extension ainsi que les performances à l'exécution de la simulation dynamique du procédé TINA.

Expérimentations avec les simulations dynamiques de procédés 10

10.1 Introduction

Dans le chapitre 8, nous avons évalué notre premier prototype sur des applications métier du génie de procédés. Les expérimentations nous ont permis de quantifier les gains atteignables sur une architecture de type grappe de calcul. Toutefois, nous avons pointé du doigt certaines optimisations qui peuvent être réalisées afin de s'approcher au plus de la borne inférieure théorique compte tenu du placement proposé par l'ordonnanceur.

Dans un premier temps, nous proposons d'évaluer notre extension. Pour cela, le temps de réalisation des étapes d'ordonnancement, de partitionnement, de génération et de distribution est relevé. Nous corrélons les résultats observés au moyen des complexités des algorithmes utilisés.

Puis, nous étudions deux axes permettant de minimiser le temps d'exécution d'une simulation des procédés :

1. Plusieurs facteurs sont étudiées afin d'évaluer les performances à l'exécution compte tenu du placement des tâches. Pour cela, deux ordonnanceurs, ETF et DSC (ETF et DSC sont présentés dans le chapitre 4 page 82), sont utilisés. La section 10.7 étudie l'impact du modèle de coût utilisé comme estimations en entrée des ordonnanceurs, sur les performances à l'exécution. La section 10.8 compare les deux ordonnanceurs et la section 10.10 étudie le parallélisme de type *pipeline* décrit dans le chapitre 6, page 108.
2. Une optimisation du modèle d'exécution. Le recouvrement entre les pas de temps (*i.e.* sans les barrières de synchronisation) est étudié dans la section 10.9.

Avant cela, la section suivante présente la définition du simulateur du comportement d'une application du génie des procédés avec l'API de programmation KAAPI.

10.2 Définition du simulateur

La simulation du comportement de l'exécution des applications du génie des procédés sous KAAPI requiert une sémantique particulière liée à celle définie par Athapascan. En effet, le standard CAPE-OPEN prône une approche par composants alors que l'API Athapascan développe un modèle de programmation procédural à base de tâches.

La transcription d'un modèle de programmation par composants en un modèle procédural s'attache principalement à définir une dépendance supplémentaire entre les différentes invocations de méthodes sur un même composant par rapport au flot de données de l'application. Ainsi, chaque méthode invoquée sur un composant se traduit, suivant l'API Athapascan, en un appel de méthode dont les paramètres sont ceux intiaux augmentés du paramètre représentant le composant. Formellement, la tâche accède à ce paramètre en modification (toute méthode peut modifier l'état du composant), c'est-à-dire qu'aucune concurrence d'invocations n'est autorisée.

Cette section décrit chacune des étapes INDISS-RT (*FirstCompute*, *NetworkCompute* et *LastCompute*) selon le modèle de programmation Athapascan. L'écriture d'une telle application consiste à décrire les contraintes de dépendance de données issues du schéma d'exécution présentées en section 6.4, page 105.

10.2.1 *FirstCompute*

La figure 10.1 présente les deux types de tâches créées représentant l'étape *FirstCompute*. Ils caractérisent les deux types principaux d'opérations unitaires qui participent à la résolution

Début du code

```
struct UO_ARC_FC{
  public:
    void operator () (a1::Shared_rw<UO> UO,
                      const std::vector<a1::Shared_r<MO> >& rMO,
                      const std::vector<a1::Shared_w<MO> >& wMO){
      //code
      ...
    }
};

struct UO_NODE_FC{
  public:
    void operator () (a1::Shared_rw<UO> UO,
                      const std::vector<a1::Shared_r<MO> >& rMO,
                      const std::vector<a1::Shared_w<MO> >& wMO,
                      a1::Shared_w<double> Comp){
      //code
      ...
    }
};
```

Fin du code

FIG. 10.1 – Définition des deux types de tâches (selon le type d'opérations unitaires : arc ou nœud) composant l'étape *FirstCompute* avec le modèle de programmation Athapascan. Les paramètres ainsi que leur mode d'accès sont définis dans le texte.

dynamique définit dans le standard CAPE-OPEN. Ainsi, on distingue la tâche `UO_ARC_FC` de la tâche `UO_NODE_FC` représentant respectivement les opérations unitaires de type arc et nœud (*c.f.*, chapitre 2, page 43).

La première (`UO_ARC_FC`) expose uniquement les dépendances de données sur les *Material Object*. Ceux connectés en amont définissent un mode d'accès en lecture et ceux en aval caractérisent un accès en écriture. Ces données sont représentées sous la forme d'un vecteur, c'est-à-dire un tableau de *Material Object*.

La seconde tâche (`UO_NODE_FC`) requiert les mêmes paramètres formels auxquels il est nécessaire d'ajouter la contrainte de dépendance avec le solveur qui nécessite les compressibilités initiales calculées par les opérations unitaires de ce type. Ces données sont donc produites (accès en écriture) par la tâche et sont consommées lors de l'étape suivante par le solveur.

10.2.2 *NetworkCompute*

Le chapitre 6.2, page 101 décrit l'étape *NetworkCompute* comme une super-étape marquant la séparation des traitement entre solveur, opérations unitaires de type arc et de type nœud. Le schéma d'exécution, tel que nous l'avons développé, suit les mêmes contraintes.

La figure 10.2 illustre la définition formelle de chacune des tâches unitaires composant l'étape *NetworkCompute*. La tâche `SOLVER` représente l'entité de synchronisation associée aux traite-

Début du code

```
struct SOLVER{
  public:
    void operator () (a1::Shared_w<Solver> Solveur,
                      const std::vector<a1::Shared_r<double> >& Val,
                      const std::vector<a1::Shared_w<double> >& Pressure){
      //code
      ...
    }
};

struct ARC_NC{
  public:
    void operator () (a1::Shared_rw<UO> UO,
                      const std::vector<a1::Shared_r<MO> >& MO,
                      a1::Shared_w<double> Flow,
                      const std::vector<a1::Shared_r<double> >& Pressure){
      //code
      ...
    }
};

struct NODE_NC{
  public:
    void operator () (a1::Shared_rw<UO> UO,
                      const std::vector<a1::Shared_r<MO> >& MO,
                      const std::vector<a1::Shared_r<double> >& Flow,
                      a1::Shared_w<double> Comp,
                      const std::vector<a1::Shared_r<double> >& Pressure){
      //code
      ...
    }
};
```

Fin du code

FIG. 10.2 – Définition des tâches composant l'étape *NetworkCompute* suivant le modèle de programmation Athapascan.

ments du solveur qui nécessite toutes les contributions (débits, compressibilités et dérivées) des opérations unitaires. Les compressibilités initiales, calculées à l'étape précédente, sont utilisées en tant que paramètres d'entrée en lecture (le vecteur de *double* `Val`). En contrepartie, il propose

le jeu de pressions en chaque point du réseaux (vecteur `Pressure`) et accède en écriture à cet ensemble. C'est alors que la sous-étape *ArcNC* peut débuter. Les tâches (`ARC_NC`) qui composent cette sous-étape sont caractérisées par les paramètres suivants (outre la donnée *Composant* `UO`) :

- les deux pressions (le vecteur `Pressure`) proposées en entrée/sortie par le solveur (paramètre en lecture) ;
- ses deux *Material Object* (vecteur `MO`) connectés en entrée/sortie (paramètre en lecture) ;
- le débit (`Flow` en écriture) calculé à partir des quatre paramètres en lecture.

L'exécution de ces tâches ne peut débuter tant que les pressions n'ont pas été produites par la tâche `SOLVER` à cause des dépendances de flot de données sur les variables `Pressure`.

L'exécution des tâches `NODE_NC` composant la sous-étape *NodeNC* est soumise à la production des pressions (par la tâche `SOLVER`) ainsi que des débits des entrées et des sorties produites par les tâches `ARC_NC`. En fonction de ses *Material Object* connectés, la tâche `NODE_NC` calcule sa compressibilité.

Les compressibilités et les débits calculés par ces deux derniers types de tâches sont les paramètres d'entrée à la tâche `SOLVER` (paramètre `Val` sur la figure 10.2) qui calcule si le critère de convergence est atteint. Dans le cas contraire, les tâches `ARC_NC` et `NODE_NC` consommeront les valeurs de sortie `Pressure` de la tâche `SOLVER`. Dans le cas contraire, elles ne seront pas utilisés : seule la dépendance sur la donnéee `Solveur` en modification caractérise la synchronisation entre l'étape *NetworkCompute* et *LastCompute*.

10.2.3 *LastCompute*

Les tâches de la dernière étape *LastCompute* sont implantées suivant le modèle présenté en figure 10.1. Le schéma d'exécution de cette étape s'apparente à celui de l'étape *FirstCompute* à l'exception que seuls les *Material Object* sont produits. Ainsi, un seul type de tâche est spécifié (nommée `UO_LC`) représentant l'ensemble des opérations unitaires. La dépendance avec l'étape précédente (*NetworkCompute*) est réalisée par le biais de la donnée `Solveur`. La dépendance avec l'étape suivante (*FirstCompute* au pas de temps $t + 1$) s'effectue au moyen de la dépendance sur la donnée `UO` définissant la sérialisation des invocations de méthodes sur un composant opération unitaire `UO`.

Les figures 10.1, 10.2 et 10.3 définissent totalement l'ensemble des tâches qui participent à l'exécution. Il est à présent nécessaire de les créer selon un ordre séquentiel valide tel que définit par la sémantique Athapascan.

10.2.4 Schéma d'exécution principal

Le schéma d'exécution pour notre simulateur suit l'approche suivante :

1. construction du graphe de flot de données entre les tâches (invocations de méthodes) et les différents paramètres, dont le paramètre représentant le composant ;
2. calcul d'un ordonnancement adapté par différents algorithmes décrits ci-dessous ;
3. utilisation du support exécutif proposé dans notre extension.

Début du code

```
struct UO_LC{
  public:
    void operator () (a1::Shared_r<Solver> Solveur,
                      a1::Shared_rw<UO> UO,
                      const std::vector<a1::Shared_r<MO> >& rMO,
                      const std::vector<a1::Shared_w<MO> >& wMO){
      //code
      ...
    }
};
```

Fin du code

FIG. 10.3 – Définition des tâches qui composent l'étape *LastCompute* suivant le modèle de programmation Athapascan.

Le calcul de l'ordonnancement, spécialisé pour notre simulateur, prend en compte les deux points non traités actuellement par notre extension de KAAPI.

- gestion des contraintes de placement afin de donner un même site d'exécution aux différentes tâches qui s'appliquent sur un même composant ;
- gestion d'un ordre local (sur un même site) d'exécution des tâches en utilisant la politique locale calculée par les ordonnanceurs DSC ou ETF.

Nous avons utilisé différents schémas de calcul de l'ordonnancement qui suivent le schéma algorithmique suivant :

1. construction d'un graphe « *représentatif* » extrait du graphe de flot de données initial ;
2. calcul de l'ordonnancement de ce graphe « *représentatif* » par l'algorithme ETF ou DSC ;
3. calcul des sites et de l'ordre d'exécution local des tâches sur le graphe de flot de données.

En sortie, les tâches sont distribuées sur les différents processeurs grâce au développement de notre support exécutif. Le calcul de l'ordre local d'exécution est décrit dans la section 10.2.4.2. La section suivante expose les deux représentations utilisées pour gérer les contraintes de placement liées à un même composant.

10.2.4.1 Placement du graphe de l'application

Dans les expériences suivantes, nous avons utilisé, deux types de graphe "représentatif". Le premier est le graphe de précédence entre composants, appelé graphe « *topologique* », qui replie les différents nœuds tâche du graphe de flot de données sur un même nœud représentant le composant ; de cette manière nous intégrons les contraintes de placement. Le second graphe « *représentatif* », nommé graphe « *pipeline* », est un graphe de flot de données dans lequel nous avons supprimé les tâches liées à l'étape *NetworkCompute* et imposé des poids infinis sur les données « *Composant* » pour forcer le placement, sur un même site, des tâches s'y référant.

Les expériences présentées en section 10.6, 10.7, 10.7, 10.9 utilise un graphe « *représentatif* » composé des tâches issues du schéma d'exécution de l'étape *FirstCompute* ou *LastCompute* (qui

suit le même schéma) : ce graphe est alors le schéma de procédés ordonnés par les dépendances de données suivant une approche modulaire séquentielle. Ce graphe est nommé « *topologique* ».

La section 10.10 évalue les performances du placement sur un graphe développé exposant un parallélisme de type *pipeline*. Pour cela, le graphe ordonnancé est composé des tâches des étapes *LastCompute* et *FirstCompute* avec les relations de dépendances exposées dans le chapitre 6, page 105. Les tâches `SOLVEUR`, `ARC_NC` et `NODE_NC` composant l'étape *NetworkCompute* ne sont pas soumises à l'ordonnanceur. Deux raisons justifient cela. La première repose sur le fait que le schéma d'exécution de cette étape expose un parallélisme quasi-parfait : les sous-étapes *ArcNC* et *NœudNC* sont séquentielles mais les tâches qui les composent sont concurrentes. Ainsi, seule une problématique d'équilibrage de charge a un sens. Toutefois, le temps de calcul de ces tâches est négligeable comparé aux temps des tâches des deux autres étapes. C'est alors que leur ordonnancement n'a que peu d'impact sur le temps global de simulation et seul le respect des contraintes de placement (les méthodes sur un même composant sont affectées au même site) doit être traité. Ce graphe est dit « *pipeline* ».

La section 10.5 définit le placement des tâches à l'identique à celui utilisé dans notre première approche. En théorie, le placement devrait être identique au placement proposé par DSC sur un graphe « *topologique* ». Toutefois, la technique contraignant les sites d'exécution des composants propriétaires diffère entre les deux approches. Alors que notre ancien prototype fusionnait une partition avec celle composée des composants contraints, l'approche dans KAAPI génère une $p + 1^{\text{ème}}$ partition qui sera placé selon la stratégie « *round robin* » énoncé dans le chapitre 9 page 156.

10.2.4.2 Création de l'ordre RFO

A la fin du traitement précédent, chaque composant est affecté à un site particulier qui est reporté sur l'ensemble des tâches (des méthodes) de ce même composant. La construction de l'ordre RFO est également basée sur les informations calculées à partir du graphe restreint « *représentatif* ».

Dans le premier cas (graphe « *topologique* »), l'ordre de création des tâches est fixé à l'identique pour les étapes *FirstCompute* et *LastCompute*. Pour l'étape *NetworkCompute*, compte tenu de la barrière de synchronisation en début et fin ainsi que de la faible dépendance au sein d'une même sous-étape (*ArcNC* et *NœudNC*), l'ordre de création n'a que peu d'importance. Il est toutefois nécessaire de respecter la séquence des sous-étapes : *Solveur* $\rightarrow$ *ArcNC* $\rightarrow$ *NœudNC* $\rightarrow$ *Solveur*.

Dans le second cas (graphe « *pipeline*), l'ordre de création des tâches pour les étapes *FirstCompute* et *LastCompute* est calculé par l'ordonnanceur. La méthode précédente est employée pour ordonner les tâches de l'étape *NetworkCompute*.

Avant de quantifier la charge de calcul générée par notre extension ainsi que les performances obtenues, la section suivante présente la méthodologie de prise de temps ainsi que l'architecture sur laquelle nous avons mené les expériences.

10.3 Plate-forme d'évaluation & Protocole expérimental

Les tests ont été menés sur la grappe calcul *Paravent* de la plate-forme Grid5000. Cette grappe est constituée des éléments suivant :

- Unités : 99 nœuds.
- Modèle : HP ProLiant DL145G2.
- CPU : AMD Opteron 246, 2 GHz / 1MB x 2.
- Mémoire : 2 GB.
- Réseau : Gigabit Ethernet.
- Système d'exploitation : Linux.

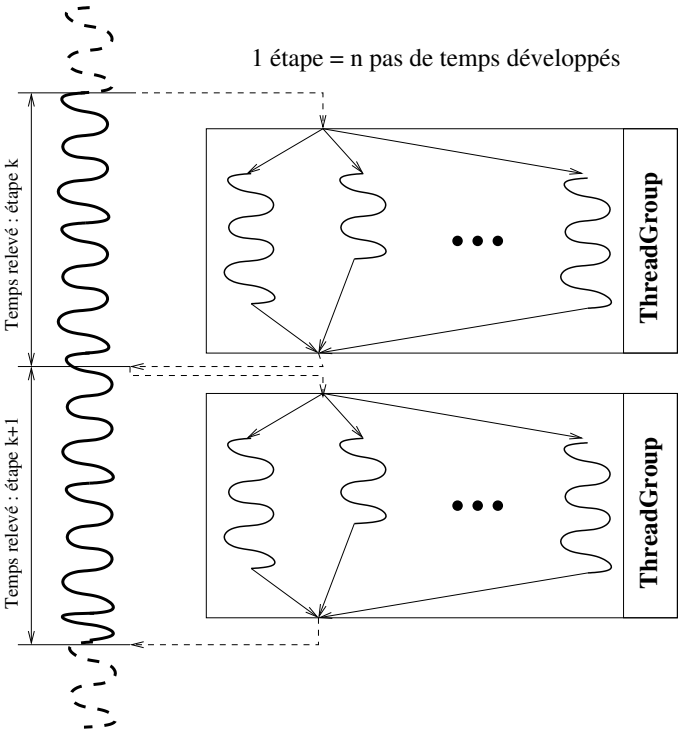


FIG. 10.4 – Méthodologie de prise de temps pour l'étude des performances à l'exécution.

Les résultats présentés sont issus de la moyenne de 50 exécutions dans des conditions identiques. Les temps sont relevés à la fin de l'exécution du *ThreadGroup* qui représente l'exécution distribuée d'un ensemble d'itérations de l'application itérative. Ainsi, les temps mesurés sur le *thread* principal englobent les délais dus aux communications de contrôle de l'exécution (démarrage, fin). La figure 10.4 illustre ces points de collecte.

L'évaluation de notre extension à KAAPI suit la même méthodologie. Les dates de début et de fin sont respectivement prises avant et après l'appel à la méthode en charge de réaliser l'une

des opérations de la séquence des traitements sur l'entité d'exécution centrale. Les traitements mesurés (*c.f.*, figure 9.9, page 153) sont les suivants :

- ordonnancement ;
- partitionnement ;
- génération ;
- distribution.

La section suivante évalue le coût des différentes étapes liées à une exécution statique.

10.4 Coût des différentes étapes de l'ordonnancement

Les tableaux 10.1 présente le temps des trois sous-étapes permettant l'ordonnancement du graphe DFG de l'application :

1. Construction de la structure de données « *ordonnanceur* » à partir de la représentation DFG KAAPI.
2. Ordonnancement du graphe au moyen de l'ordonnanceur.
3. Post-traitement visant à affecter un site d'exécution à chaque tâche de l'application en tenant compte, le cas échéant, des contraintes de placement définies par le développeur.

	« *Topologique* »			Total	« *Pipeline* »			Total
	Pré	Ordonnanceur	Post		Pré	Ordonnanceur	Post	
ETF	4 ms	$<$ 1 ms	4 ms	8 ms	4 ms	$<$ 1 ms	4 ms	9 ms
DSC	4 ms	1 ms	4 ms	9 ms	4 ms	2 ms	5 ms	11 ms

TAB. 10.1 – Temps relevés (en millisecondes) selon les deux ordonnanceurs ETF et DSC pour les trois sous-traitements (Pré : construction de la structure de données en entrée de l'ordonnanceur ; Ordonnancement : réalisation du calcul de l'ordonnancement ; Post : affectation des sites à chaque tâche du graphe DFG KAAPI). Les deux types de graphes (« *topologique* » et « *pipeline* ») sont représentés.

On constate que les temps requis au pré et post-traitement sont plus coûteux que le coût lié au calcul de l'ordonnancement. Ceci est vrai compte tenu des tailles de graphes traités qui sont relativement faibles (54 tâches pour le graphe « *topologique* » et 108 pour le graphe « *pipeline* »). En effet, les ordonnanceurs DSC et ETF ont respectivement une complexité en $O(v^2p)$ et $O((v+e)\log v)$ où p est le nombre de processeurs, v est le nombre de tâches du graphe et e est le nombre d'arcs. Les algorithmes employés pour les pré et post-traitements ont tout deux une complexité en $O(v(\log v))$ multipliée par une constante c. Ainsi, pour des graphes dont le nombre de tâches est supérieur quelques milliers, la tendance s'inverse et le temps de calcul de l'ordonnanceur domine le temps total de cette étape.

La figure 10.5 présente un comparatif des temps requis pour mener à bien chaque étape (partitionnement, génération, distribution) sur 10 processeurs selon le nombre initial (*i.e.*, avant ajout

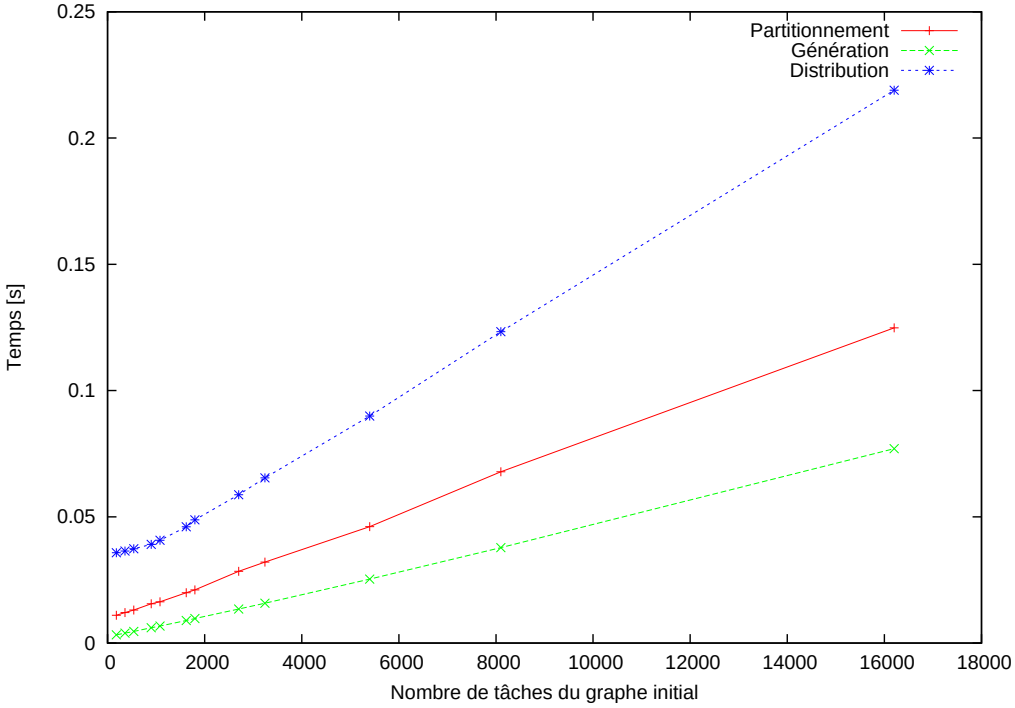


FIG. 10.5 – Comparatif des temps de chaque étape en pré-traitement à une exécution statique en vue d'une exécution sur 10 processeurs.

des tâches de communication et de contrôle) de tâches composant une étape KAAPI. Ceci revient à développer le graphe d'exécution itératif sur plusieurs pas de temps (*c.f.* figure 10.4 de la section précédente).

Les deux étapes « Partitionnement » et « Génération » sont linéaires en fonction du nombre de pas de temps développés. Ces deux étapes ont une complexité en $O(n + n_{\text{ajout}})$ où n est le nombre de tâches de l'application initiale (le graphe DFG de l'application) et n_{ajout} est le nombre de tâches ajoutées (tâches de communication et de contrôle) par notre approche.

La complexité de l'étape de distribution est difficilement qualifiable. En effet, elle dépend, d'une part, du volume de données à communiquer et, d'autre part, du traitement à réaliser à réception de ladite partition sur chaque ressource de calcul. Ainsi, le temps global de cette étape est composé d'un temps indivisible (le temps de communication de chaque sous-graphe généré à destination des processeurs) qui peut être en partie recouvert par du calcul. Typiquement, le traitement local lié à la partition i peut débuter alors que la partition $i + 1$ est en cours de transmission. Notons toutefois que le terme dominant est le traitement effectué à réception des partitions.

Nous constatons que pour des tailles de graphes conséquent, le temps de distribution est linéaire en fonction du nombre de tâches. En deçà d'un seuil (sur la figure, ce seuil est représenté par 600 tâches ce qui correspond en pratique, après génération des tâches de contrôle et de communication, à environ 2000 tâches), le temps de distribution est relativement constant. Ce comportement s'explique par un coût important indivisible de création des *threads* sur les entités de

calcul distantes.

Le tableau 10.2 résume les temps relevés en millisecondes pour les trois étapes présentées ci-avant (Partitionnement, Génération, Distribution) avec une étape KAAPI de 90 pas de temps. Les nombres de tâches totales ($n_{\text{all}} = n + n_{\text{ajout}}$) et maximales des partitions $\max(n_i)$ sont également représentés.

# Processeurs	Partitionnement	Génération	Distribution	n_{all}	$\max(n_i)$
2	100	55	201	34290	19172
4	126	82	357	59580	22772
6	124	82	254	60570	18722
8	125	75	235	55800	17552
10	125	77	219	58770	17282

TAB. 10.2 – Résumé des temps (en millisecondes) relevés pour les 3 étapes (Partitionnement, Génération et Distribution) avec une étape KAAPI de 90 pas de temps. Les nombres de tâches totales ($n_{\text{all}} = n + n_{\text{ajout}}$) et maximales ($\max(n_i)$) sont également représentés.

Pour un nombre de partitions créées entre 4 et 10, le temps des étapes de partitionnement et de génération sont sensiblement équivalents. Compte tenu des complexités énoncées ci-avant, ces temps se justifient au regard du nombre de tâches totales créées (n_{all}) de l'ordre de 60000. Pour 2 partitions, ce nombre est fortement inférieur ce qui se répercute sur le temps de réalisation des deux traitements.

L'étape de distribution est, quant à elle, soumise aux deux facteurs énoncés ci-avant. Le premier tient compte du volume de données à envoyer et est fonction du nombre de tâches totales (n_{all}) du graphe DFG augmenté. Le second facteur est fonction du traitement à effectuer à réception de chaque partition sur chaque entité distante de calcul. Il est également dépendant du taux de recouvrement des communications par du calcul. D'après cette description, l'étape de distribution est sensiblement moins coûteuse lorsque le nombre de processeurs augmente (l'ensemble des traitements locaux affectés à chaque partition est d'autant plus faible qu'il y a des processeurs). Pour un nombre de partition égal à 2, ce temps reste bas compte tenu du volume moindre de tâches du graphe augmenté. Ceci se répercute sur $\max(n_i)$. Pour 4 partitions, le temps est excessivement grand comparé aux autres pour deux raisons :

1. le nombre de tâches de la plus grosse partition est plus important que dans les autres cas ;
2. cette partition est la dernière à être communiquée.

Ainsi, afin de réduire le temps de distribution, une optimisation consisterait à envoyer les partitions par ordre décroissant de tâches ce qui maximise alors ce taux de recouvrement.

En conclusion, le traitement associé au démarrage de l'exécution d'une application reste sensiblement négligeable : pour des graphes de 60000 tâches, le temps total relevé est en deçà de la demi seconde. Toutefois, les traitements réalisés au cours de ces étapes peuvent être optimisés. Par exemple, l'usage intensive de la fonction de recherche dans des « *map* » STL [103] devra être revu

et utilisé avec parcimonie. Egalement, il sera bon dans un avenir proche de quantifier l'optimisation concernant le recouvrement communications par du calcul pour l'étape de distribution.

A présent, nous présentons les résultats à l'exécution obtenus en faisant varier différents critères. Dans un premier temps, nous proposons de valider cette approche de simulations du comportement des simulations de procédés en comparant les résultats obtenus avec KAAPI à ceux de notre prototype Windows. Pour cela, chaque tâche effectue un travail comparable en temps à celui évalué sur l'architecture matérielle présentée dans le chapitre 8, page 130, c'est-à-dire qu'une seconde sur l'architecture précédente représente une seconde sur la nouvelle architecture. Par la suite, nous avons divisé le grain des tâches par 4 afin de rendre compte d'une telle simulation sur une architecture moderne telle que celle présentée dans la section 10.3.

10.5 Validation du simulateur

Afin de proposer un environnement de simulation du comportement des applications CAPE, chaque tâche simule un travail correspondant aux temps mesurés lors de l'exécution de la simulation avec l'environnement INDISS. Pour cela, une boucle de calibration permet d'ajuster le nombre d'itérations nécessaire afin de consommer un temps t de calcul.

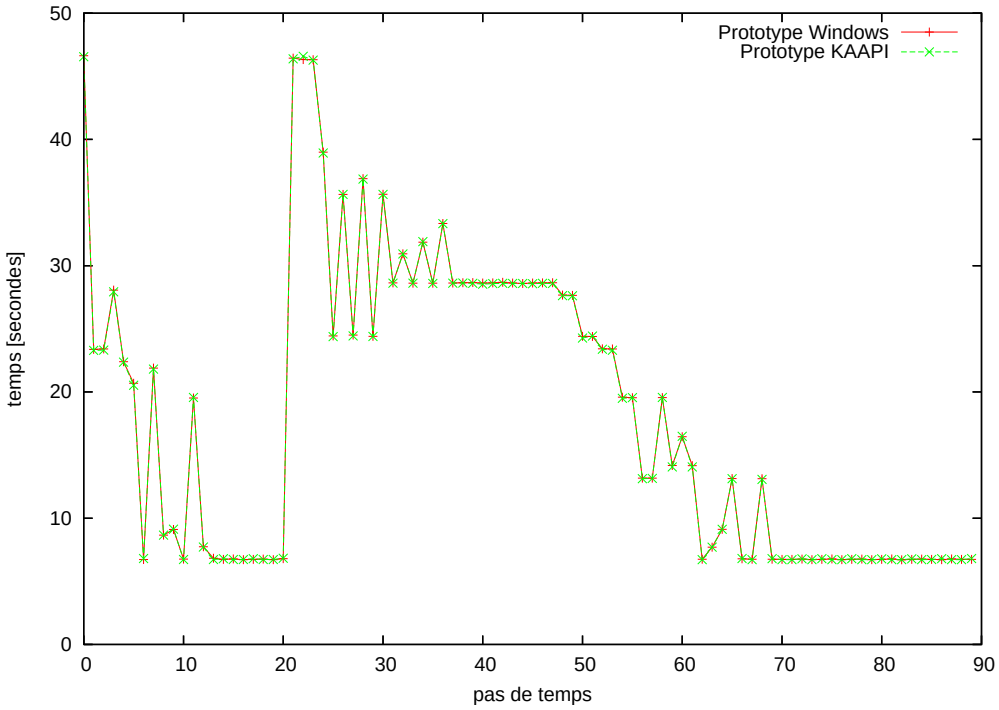


FIG. 10.6 – Comparatif du temps de simulation entre KAAPI et le prototype Windows en séquentiel.

La figure 10.6 compare les temps d'exécution séquentiels entre les environnements INDISS et KAAPI. On constate que les deux courbes correspondent parfaitement, ce qui nous permet d'affirmer que la boucle de calibration mise place nous permet de simuler au mieux les temps d'exécution de chaque tâche de l'application.

La figure 10.7 confronte l'environnement KAAPI avec notre premier prototype dans les mêmes conditions d'utilisation. Pour cela, le placement des composants est à l'identique et le nombre de processeurs utilisé est égal à 5. La figure 10.7 présente également la borne inférieure atteignable avec le placement proposé. Cette borne inférieure est calculée à partir de la somme des temps de chaque tâche de chaque partition comme présentée dans le chapitre 8, page 137.

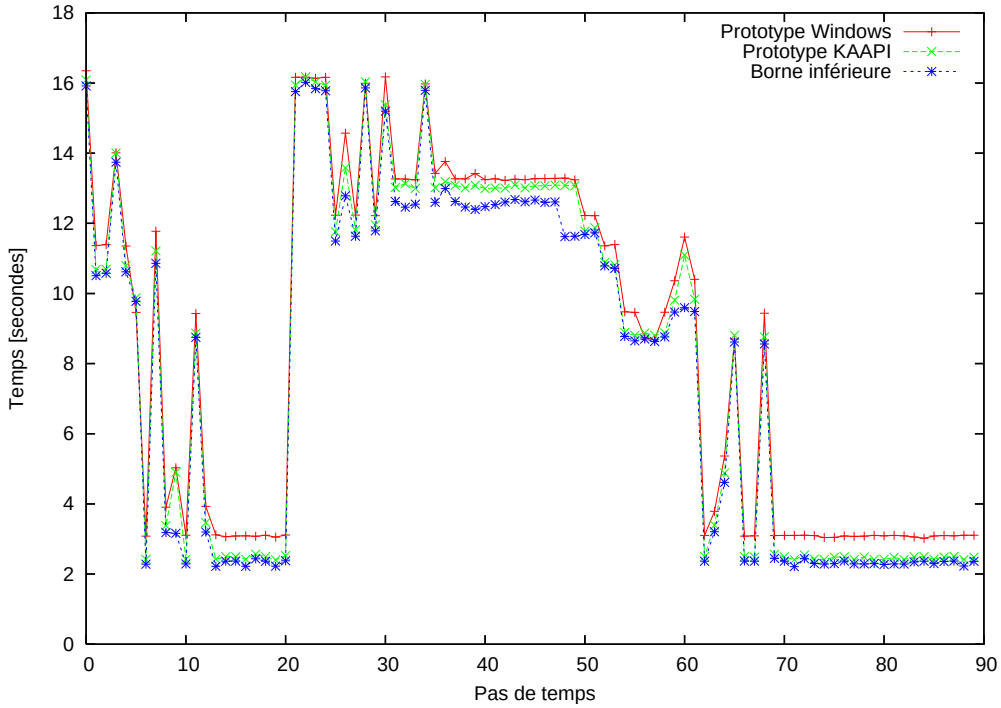


FIG. 10.7 – Comparatif du temps de simulation entre KAAPI et le prototype Windows sur 5 processeurs dont les tâches sont affectées aux même sites d'exécution (placement identique).

En général, le prototype KAAPI offre un temps d'exécution moindre comparé aux temps relevés précédemment. Un gain de 20% est observé sur la phase de calcul « *stationnaire* » (itérations 16 à 20 et 69 à 89). Sur la période « *perturbée* » de calcul (itérations 0 à 15 et 21 à 68), les performances sont soit sensiblement identiques à celles obtenues avec notre ancien prototype, soit proche de la borne inférieure.

Dans tous les cas de figure, l'environnement KAAPI, couplé à l'architecture matérielle employée, offre de meilleure performance. Nous avons intentionnellement mentionné le gain dû à l'architecture matérielle. En effet, même si le travail simulé est identique à celui réalisé avec l'environnement « *Windows* », les communications ne sont pas traitées à l'identique et ceci est

particulièrement dû au fait que les machines utilisées sont bi-processeurs. En effet, KAAPI traite les communications à travers un *thread* qui effectue l'envoi et la réception en concurrence du *thread* de calcul principal. Sur un architecture monoprocesseur, le *thread* de calcul est « *descheduled* » (il devient inactif afin de donner la main aux autres *threads*, en l'occurrence, le *thread* de communication) afin de réaliser la communication. Sur un architecture bi-processeurs, les *threads* de calcul et de communication sont chacun affectés à un processeur particulier. L'implémentation OmniORB de CORBA permettrait d'exploiter une telle propriété en partie. En effet, la réception active un *thread* en charge de traiter la communication : sur une architecture bi-processeurs, ce *thread* de réception peut être exécuté en concurrence du flot d'exécution principal. Au contraire, bien que l'emploi du mot clé *oneway* définit un appel de méthode asynchrone, l'émission reste bloquante, c'est-à-dire que le flot de contrôle ne reprend la main qu'une fois le message envoyé. Comme montré dans [73], La couche de communication de KAAPI permet également une émission non-bloquante. Ainsi, notre application tire profit de l'architecture bi-processeurs grâce au recouvrement des communications par du calcul offert par l'environnement KAAPI.

10.6 Influence du grain de l'application

Cette expérience met en avant la différence de gain (en terme d'accélération) suivant le travail simulé par chaque tâche de l'application. En effet, les courbes précédentes se basent sur les temps relevés sur un ensemble de machines dont la fréquence processeur est de 733 MHz. L'architecture utilisée (présentée ci-avant en section 10.3, page 171) est composée de processeurs à 2 GHz. Bien que le ratio entre ces deux fréquences est de l'ordre de 3, les technologies actuelles, que se soit en terme de cache ou de mémoire vive, sont bien plus performantes que celles proposées par notre ancienne architecture. Ainsi, diviser le temps d'exécution de chaque tâche par 4, n'est pas incohérent afin de rendre compte des performances à attendre sur des architectures modernes.

La figure 10.8 étudie l'influence du grain de l'application sur l'accélération par pas de temps dans les mêmes conditions que l'expérience menée précédemment. Durant la phase où le calcul est perturbé, l'accélération calculée pour les deux cas est très proche. Dans le phase de calcul « *stationnaire* », l'accélération engendrée par un grain fin est amoindrie d'environ de $0,1$ à $0,2$ point. Ceci s'explique au regard du temps des communications qui n'est pas réduit du facteur 4 (le même volume de données est communiqué). Bien que les communications peuvent être recouvertes par du calcul, il est nécessaire qu'il y ait du calcul disponible. Il arrive qu'un processeur soit en famine, par exemple, lors de phases de réduction ou de diffusion. Ce délai d'attente se répercute d'autant plus sur l'accélération que le temps de calcul est faible.

10.7 Modèle de coût

Cette expérience mesure l'impact du modèle de coût injecté en entrée des ordonnanceurs DSC et ETF sur lequel ils se basent pour réaliser le calcul du placement. Nous étudions trois modèle de coût. Les deux premiers considèrent le temps de communication à 500 microsecondes.

1. Modèle « *stationnaire* » : moyenne des temps relevés pour chaque tâche lors des phases de calcul stationnaire (itérations 16 à 20 et 69 à 89).

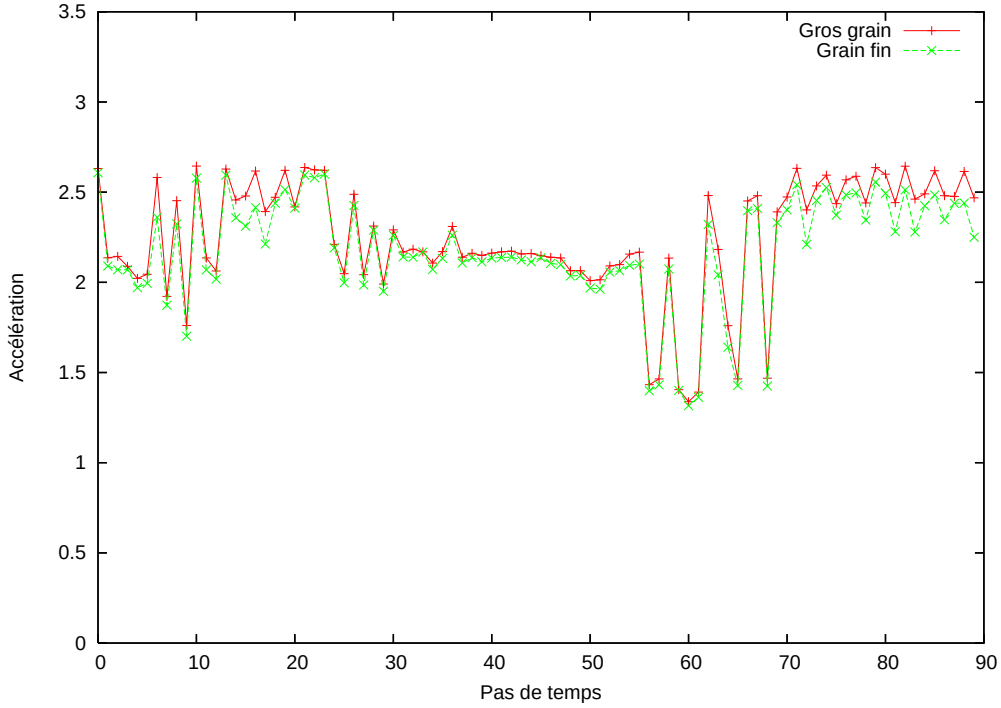


FIG. 10.8 – Influence du grain de la simulation sur l'accélération par pas de temps.

2. Modèle « *perturbé* » : moyenne des temps pour chaque tâche du sous-ensemble constant (itérations 40 à 50) du régime perturbé.
3. Modèle « *uniforme* » : chaque tâche est affectée d'un poids de 10 unités de temps et les communications à 1 unité de temps.

Les deux premiers modèles évaluent la qualité de l'ordonnancement calculé si l'on privilégie la phase de calcul perturbé ou celle stationnaire. Le troisième modèle, quant à lui, favorise le regroupement des séquences d'opérations unitaires suivant l'heuristique DSC. Il est à prévoir que cette troisième heuristique propose des résultats médiocres avec l'ordonnanceur ETF de par sa politique d'affectation des tâches aux processeurs (la première tâche prête est affectée au premier processeur libre qui minimise sa date de début). Sur un nombre de processeurs inférieure au nombre maximal, l'heuristique ETF, à l'inverse de celle développée par DSC, ne tente pas de minimiser le volume échangé de communications (ce qui implique que les séquences d'opérations unitaires ne sont pas regroupées). La figure 10.9 illustre cette caractéristique en comparant les placements des tâches calculés par ETF (en haut) et DSC (en bas) sur le schéma de procédé utilisé dans ce chapitre : on remarque un chaînage de tâches de même couleur sur le placement calculé par DSC.

Les figures 10.10 et 10.11 présentent les résultats pour les trois modèles sur le temps total de la simulation (90 itérations) en fonction du nombre de processeurs.

FIG. 10.9 – Différence sur le placement des tâches entre ETF (haut) et DSC (bas) sur 5 processeurs (A visionner en couleur pour une meilleure appréciation). Alors que DSC regroupe les séquences dominantes de tâches, ETF choisit la premier processeur libre qui permet d'exécuter une tâche au plus tôt.

L'ordonnancement calculé avec DSC (figure 10.10) à partir du modèle de coût « *uniforme* » offre, en moyenne, de meilleures performances par rapport au deux autres. Ces derniers engendrent un temps d'exécution sensiblement identique à l'exception du cas 2 processeurs où le modèle « *perturbé* » se comporte un peu mieux compte tenu du placement proposé.

En considérant l'ordonnanceur ETF (figure 10.11), le modèle « *uniforme* » se comporte moins bien que les deux autres. Comme précisé ci-dessus, ETF ne regroupe pas les tâches par séquence à l'inverse de DSC. Ainsi, les coûts étant complètement biaisés, le résultat de l'ordonnancement ne peut être que « *médiocre* ». Ceci étant, remarquons que cette politique « *uniforme* » offre sensiblement les mêmes performances que la meilleure des politiques avec DSC. En effet, la phase

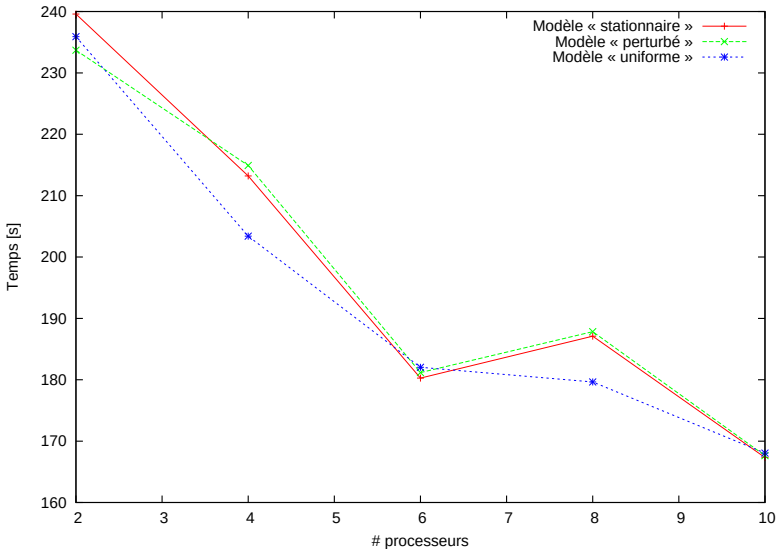


FIG. 10.10 – Courbes des temps en fonction du nombre de processeurs avec l'ordonnanceur DSC suivant trois modèles de coût.

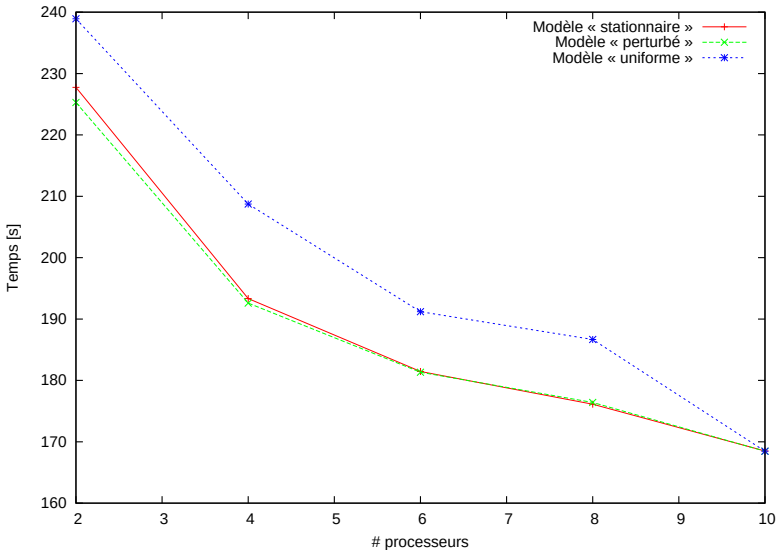


FIG. 10.11 – Courbes des temps en fonction du nombre de processeurs avec l'ordonnanceur ETF suivant trois modèles de coût.

de repliement des partitions générées pour un nombre illimité de processeurs sur un nombre limité, basée sur une technique d'équilibrage de charge, ne produit pas les meilleurs performances lorsque l'objectif à atteindre est la minimisation du temps global d'exécution. En revanche, sur

10 processeurs (qui représentent le parallélisme maximal exploitable par DSC), les performances sont identiques quelque soit l'ordonnanceur ou bien même le modèle de coût injecté. Dans ce cas, le placement proposé est identique.

Bien que proposant un placement légèrement différent, ETF calcule un ordonnancement offrant des performances sensiblement équivalentes à l'exécution pour les modèles « *perturbé* » et « *stationnaire* ». En pratique, les tâches qui consument un temps de calcul important dans le phase « *stationnaire* » (les opérations unitaires *pipeline*) reste les tâches dominant le calcul dans le phase « *perturbé* » provoquant alors un même déséquilibre de charge à l'exécution.

10.8 Comparatif ETF / DSC

Dans cette section, nous comparons les deux ordonnanceurs ETF et DSC dans leur configuration optimale, autrement dit, suivant le « *meilleur* » modèle de coût. D'après la section précédente, le modèle le plus performant est « *perturbé*» pour ETF et « *uniforme* » pour DSC.

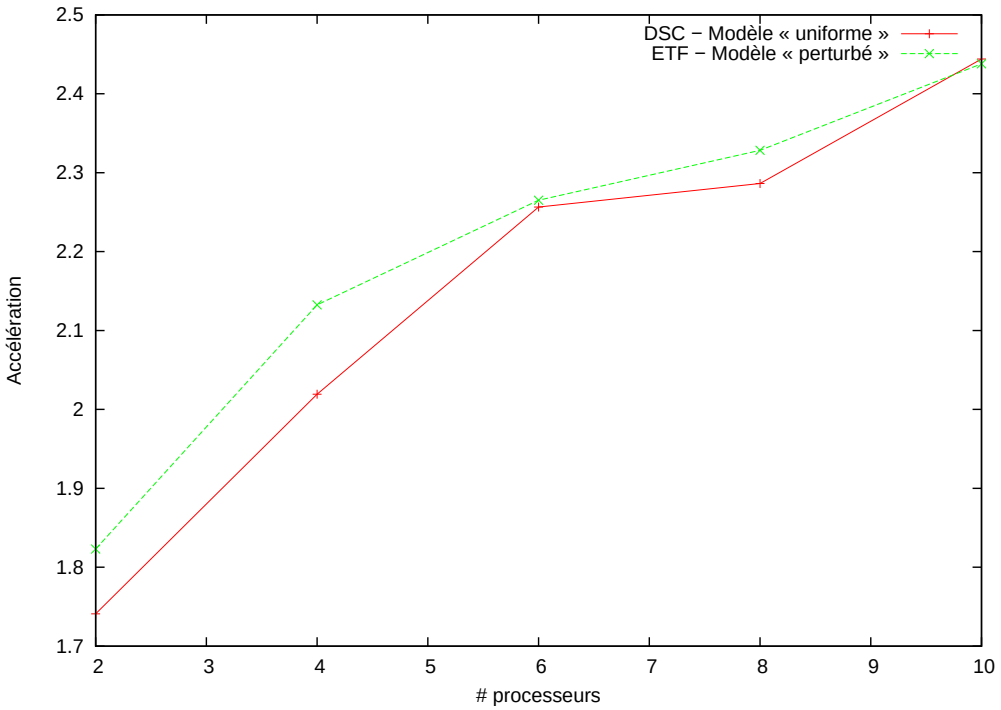


FIG. 10.12 – Comparatif des accélérations en fonction du nombre de processeurs entre DSC et ETF suivant un modèle de temps « uniforme » et « perturbé » respectivement.

La figure 10.12 présente les accélérations calculées en fonction du nombre de processeurs pour ETF et DSC dans leur configuration optimale. On constate que l'ordonnanceur qui propose le meilleur placement, ce qui se répercute sur l'accélération de l'exécution distribuée, est ETF. Plus le nombre de processeurs augmente et se rapproche du nombre optimal, plus la différence

entre les accélérations s'amoidrit. Ceci confirme que la phase de repliement utilisée par DSC est inappropriée à la minimisation du temps global d'exécution. En effet, lorsque le nombre disponible de processeurs est éloigné de l'optimal, l'heuristique de repliement dégrade les performances.

10.9 Recouvrement entre pas de temps

Cette expérience évalue l'impact du recouvrement des pas de temps sur le temps d'exécution global. Pour cela, nous faisons varier le nombre de pas de temps développés par rapport au nombre total exécutés (90 itérations). Un développement de 2 signifie que deux pas de temps sont enchaînés sans barrière de synchronisation. En revanche, il existe toujours une synchronisation liée au schéma d'exécution de la simulation : l'étape *NetworkCompute* mettant en jeu le solveur centralisé. Suivant un tel recouvrement, l'environnement KAAPI exécute alors 45 *étapes* (*c.f.* chapitre 9, page 160).

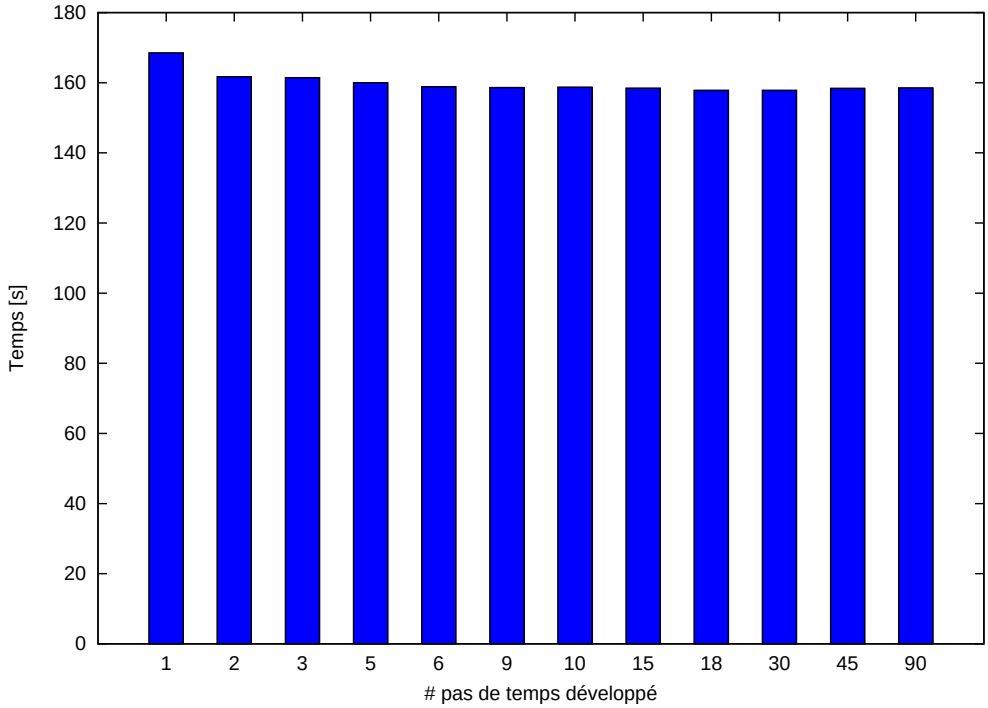


FIG. 10.13 – Influence du recouvrement entre les pas de temps sur le temps global de la simulation sur 10 processeurs avec un l'ordonnanceur ETF en configuration optimal.

La figure 10.13 présente les temps d'exécution globaux de la simulation sur 10 processeurs en fonction du nombre de pas de temps développés. Ce nombre varie entre 1 (le graphe n'est pas développé et une barrière de synchronisation existe entre chaque pas de temps) et 90 (le graphe est complètement développé : la totalité de l'exécution est réalisée sans cette barrière de synchronisation). On constate qu'un gain est obtenu *via* l'utilisation d'une telle technique. Un développement

de 2 (et donc 45 étapes KAAPI) permet de réduire le temps de près de 7 secondes (soit un gain de 4% par rapport à une exécution avec synchronisation). Le temps continue de baisser faiblement par la suite pour atteindre un minimum avec un développement de 18 ou 30 (gain de 7%). Par la suite, le temps global augmente légèrement. Ceci s'explique compte tenu de la politique de vol locale implémentée dans KAAPI. Le graphe développé sur un tel nombre d'itérations contient une dizaine de milliers de tâches ce qui influe sur le temps de vol qui parcourt le graphe pour trouver une tâche prête.

10.10 Parallélisme de type *pipeline*

Enfin, pour conclure cette série d'expériences, nous testons le placement proposé par les ordonnanceurs ETF et DSC sur un graphe « *pipeline*» (*c.f.*, section 10.2.4, page 168) où un parallélisme de type *pipeline* peut être extrait.

Jusqu'à présent, nous avons utilisé, comme représentation de l'exécution, le graphe « *topologique* » qui représente le schéma de procédé rendu acyclique *via* l'utilisation d'un algorithme de coupe métier où un unique site est calculé par l'ordonnanceur pour chaque composant. Pour tester le parallélisme de type *pipeline*, il est nécessaire de représenter les dépendances entre les méthodes pour un même composant. Il devient alors indispensable de contraindre le placement des méthodes de chaque composant à un unique site d'exécution. Ceci est réalisé en affectant un poids infini à la donnée *Composant* (`UO` sur l'exemple de la figure 10.1, page 166, par exemple).

Avec ETF, cette stratégie n'apporte aucun gain par rapport à celle employée précédemment. Mis à part le modèle de coût « *uniforme* » qui produit des performances bien plus mauvaises que ce qu'elles n'étaient ; les résultats pour les deux autres modèles («*stationnaire* » et « *perturbé* ») sont sensiblement identiques. Le placement généré est quasiment identique (à l'exception d'une unique tâche).

La figure 10.14 présente l'évolution du temps d'exécution total de la simulation en fonction du nombre de processeurs pour DSC avec le meilleur modèle de coût employé (« *perturbé* »). Nous comparons ce résultat avec le modèle de coût « *uniforme* » (proposant les meilleures performances à l'exécution) ordonnancée sur un graphe « *topologique* ».

Contrairement à ETF, un parallélisme de type *pipeline* offre, avec DSC, une amélioration sur le temps global d'exécution de l'application (de l'ordre de quelque secondes). Le modèle de coût « *uniforme* » dans la configuration « *pipeline* » propose de mauvais résultats à l'exécution comparé à son homologue sur un graphe « *topologique* ». En effet, ce modèle fut également utilisé afin d'étudier le parallélisme de type *pipeline* où chaque composant est potentiellement affecté à un site d'exécution différent (*c.f.* chapitre 6, page 108). Or, cette hypothèse s'avère être inefficace dans le cadre de la simulation du cas test TINA où, en pratique, la charge de calcul n'est pas équilibrée par partition.

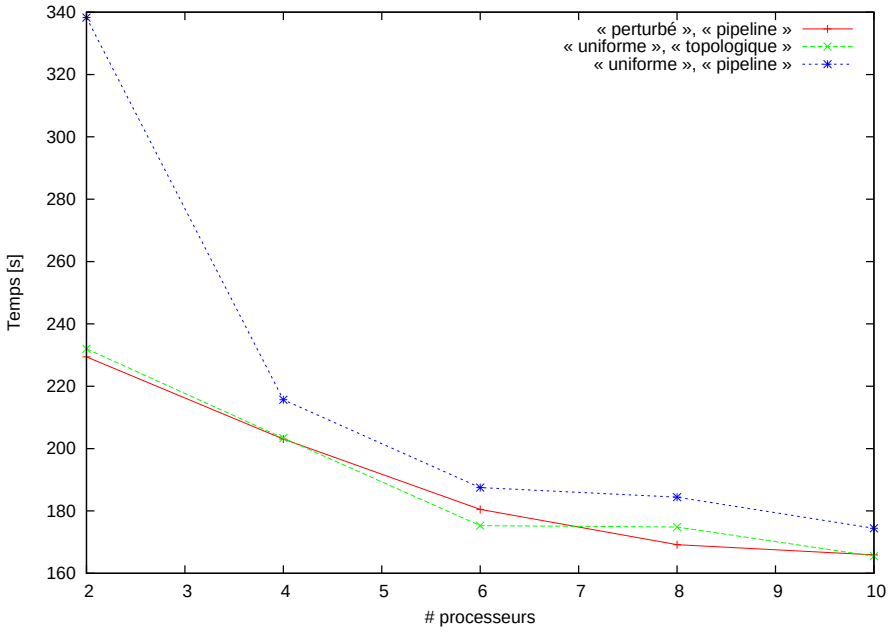


FIG. 10.14 – Evaluation de la stratégie *pipeline* de la construction du graphe selon les différents modèles de coût pour DSC en fonction du nombre de processeurs.

10.11 Bilan

Les expériences menées tout au long de ce chapitre vise à améliorer le gain d'une application du génie des procédés dont certains des composants sont au standard CAPE-OPEN. Pour cela, nous simulons le comportement de l'application de simulation dynamique TINA où les opérations unitaires de surface sont propriétaires à l'environnement d'exécution INDISS et donc placées sur un site d'exécution particulier. Nous montrons que ce démonstrateur simule parfaitement les temps observés lors de l'exécution du cas test métier.

D'après ce démonstrateur, nous avons comparés les ordonnanceurs ETF et DSC avec différentes politiques de placement. Le tableau 10.3 résume les performances, en terme d'accélérations, des deux ordonnanceurs dans leur configuration optimale selon le nombre de processeurs utilisés à l'exécution. Les accélérations calculées de référence sont également présentées comme point de comparaison. Elles sont calculées en utilisant l'ordonnanceur DSC avec un modèle de coût « *uniforme* » exploitant un parallélisme « *topologique* » où l'exécution de chaque pas de temps est entrecoupée par une barrière de synchronisation (correspondant à 1 pas de temps par étape KAAPI). Cette configuration est celle employée avec notre premier prototype. Relevons que cette valeur de référence tire bénéfice des performances à l'exécution de KAAPI comme présentées en section 10.5. Toutefois, elle offre un bon repère quant aux gains réalisés à travers cette étude.

Deux conclusions peuvent être extraites de ces résultats, l'une qualitative et l'autre quantitative. La première qualifie les deux ordonnanceurs étudiés. On constate que les performances à l'exécution, compte tenu du placement proposé par DSC, fluctuent fortement. La phase de re-

	2	4	6	8	10
ETF - Modèle « *perturbé* », Parallélisme « *topologique* », 2 pas de temps développés	1, 82	2, 18	2, 28	2, 34	2, 53
DSC - Modèle « *perturbé* », Parallélisme « *pipeline* », 2 pas de temps développés	1, 79	2, 02	2.27	2.43	2.48
Référence	1.74	2.01	2.17	2.28	2.45

TAB. 10.3 – Accélérations calculées pour les ordonnanceurs ETF et DSC dans leur configuration optimale suivant le nombre de processeurs utilisés. L'accélération de référence est également présentée comme point de comparaison.

pliement d'un nombre illimité à un nombre limité de processeurs propose principalement une politique locale d'exécution inadéquate pour les graphes issus de schémas de procédé. En particulier, il alterne l'exécution des tâches de chaque partition regroupée alors qu'il serait préférable d'exécuter totalement chaque branche avant de traiter la prochaine. On observe que le placement offrant les meilleures performances est celui visant à casser le regroupement par chaîne (propriété de la stratégie du parallélisme « *pipeline* »). En effet, un tel placement permet de mieux s'adapter aux perturbations. Cette stratégie est naturellement implémentée avec l'ordonnanceur ETF qui propose majoritairement les meilleures performances. C'est pourquoi, d'après les résultats exposés dans le tableau 10.3, l'ordonnanceur ETF est mieux adapté, car plus stable que DSC, pour les applications du génie des procédés.

Les améliorations et les études menées sur les différentes politiques utilisées pour calculer les politiques locale et globale d'exécution ont apporté, en moyenne, un rendement de 0.1 point en accélération, ce qui représente pour une exécution sur 2 processeurs, un gain de 10 secondes sur l'exécution de l'application. Il provient majoritairement de la phase de calcul « *perturbé* ». Toutefois, dans la phase « *stationnaire* », une légère amélioration est observée. Ces observations sont résumées sur la figure 10.15 pour 2 processeurs.

Pour conclure, remarquons que l'augmentation du nombre de processeurs entraîne une hausse de l'accélération. Toutefois, cette hausse est extrêmement faible. Si l'on considère les efficacités, elles ne cesse de chuter avec l'ajout de processeurs. Un rendement plus important nécessite alors de trouver plus de parallélisme que celui offert actuellement par la méthode de résolution employée.

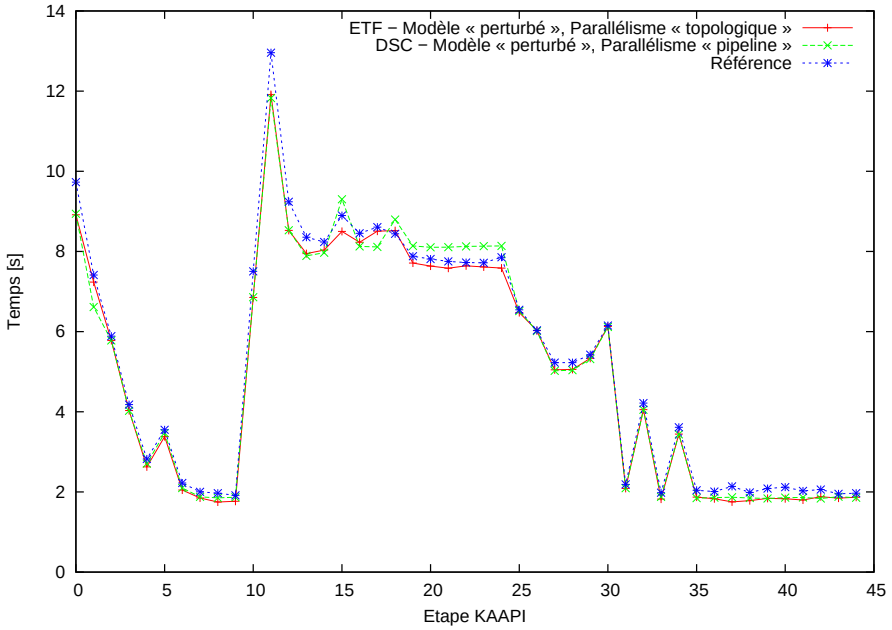


FIG. 10.15 – Evolution du temps d'exécution sur 2 processeurs en fonction des ordonnancements calculés par ETF et DSC dans leur configuration optimale. La courbe de référence, représentant le modèle utilisé pour notre premier prototype, est également présentée.

Conclusion et perspectives 11

11.1 Objectifs de la thèse

Nos travaux se sont intéressés à l'étude du schéma d'exécution des simulations de procédés dont les composants logiciels sont au standard CAPE-OPEN. Avec l'évolution des techniques logicielles et matérielles, ces simulations sont de plus en plus complexes. Cette complexité est, d'une part, logicielle où de nombreux composants informatiques hautement spécialisés doivent collaborer à la résolution de la simulation ; et d'autre part liée aux modèles mathématiques simulant la physique. La complexité logicielle est prise en charge par le standard CAPE-OPEN dans le cadre des simulations de procédés. Ce standard définit des spécifications d'interfaces pour les composants métier entrant dans la résolution d'une simulation. La complexité accrue des modèles engendre un accroissement de la puissance de calcul demandée afin d'exécuter la simulation en un temps acceptable. Malgré l'augmentation des fréquences des processeurs, la puissance développée par un calculateur séquentiel n'est pas suffisante. Un moyen incontournable de disposer de ressources de calcul importantes repose dans l'exploitation des grappes de calcul.

Dans le cadre de la simulation des procédés dont les composants logiciels sont au standard CAPE-OPEN, nous abordons ce sujet en nous basant sur les approches de résolution couramment employées en industrie. Dans ce contexte industriel, nous étudions le gain atteignable *via* l'utilisation d'une architecture de type grappe de calcul.

11.2 Démarche proposée

La démarche proposée s'articule suivant trois parties.

La première partie présente les deux domaines auxquels se rattachent cette thèse : l'ingénierie des procédés assistée par ordinateur et l'informatique parallèle et distribuée. Dans un premier temps, les méthodes de résolution adaptées à la simulation de procédés sont présentées. En outre, les interfaces exécutives du standard CAPE-OPEN sont détaillées. Ces interfaces représentent les points d'accès aux calculs métier nécessaires à la mise en application des approches de résolution. Dans un second temps, les techniques de programmation pour architectures parallèles sont exposées. La conceptions d'applications parallèles reposent sur la gestion plus ou moins explicite des propriétés liées à l'architecture. Nous argumentons que les techniques d'ordonnancement représentent la pierre angulaire à l'obtention de performances sur les architectures parallèles et distribuées. Cette partie est approfondie en incluant les différents travaux traitant du parallélisme

pour le domaine du CAPE ainsi que pour les applications de couplage de codes.

La seconde partie est dédiée, plus particulièrement, à l'approche de résolution dynamique implémentée dans INDISS de RSI [99]. A partir de l'étude du flot de données exposé par le schéma d'exécution d'une simulation au standard CAPE-OPEN, nous proposons un premier démonstrateur métier. Ce démonstrateur est utilisé afin de quantifier le gain (en terme d'accélération par rapport à une exécution monoposte) au moyen d'une approche distribuée de calcul.

La dernière partie évalue plus en détail les performances d'une exécution distribuée des simulations des procédés en se basant sur l'environnement d'exécution KAAPI. L'application test repose sur la simulation du comportement de l'exécution de la simulation dynamique utilisée dans la seconde partie. Après la validation du simulateur, nous présentons plusieurs expériences qui visent à quantifier les différentes techniques permettant d'améliorer les gains d'une approche distribuée.

11.3 Travaux réalisés

Les principaux travaux de cette thèse reposent sur l'étude du schéma d'exécution des simulations du génie des procédés dont les composants métier sont au standard CAPE-OPEN. Cette étude, présentée dans le chapitre 6, consiste majoritairement à recenser les contraintes de précédences ainsi que de dépendances de données entre les diverses tâches de l'application. D'après cette étude, nous avons implémenté deux prototypes.

Le chapitre 7 présente les techniques retenues à la définition du premier prototype ainsi que les difficultés liées à la distribution du calcul des simulations des procédés.

Dans un premier temps, nous avons entrepris de quantifier le gain atteignable sur la simulation stationnaire (ou statique) de plusieurs cas tests métier selon différentes configurations. Dans le cas du calcul stationnaire d'un procédé acyclique où le parallélisme exposé par le *flowsheet* est important, une accélération de près de 10 est atteinte sur 16 machines. Toutefois, le plus grand challenge consiste à distribuer l'exécution de simulations transitoires (ou dynamiques).

L'ordonnancement d'une application à composants logiciels n'est pas simple à résoudre dans le cas où ils exposent plusieurs méthodes. D'une part, il nécessite la sérialisation des invocations de méthodes sur chaque composant afin de conserver les dépendances de données cachées (chaque méthode accède à l'état du composant et peut le modifier). D'autre part, un composant est une entité de déploiement. Les tâches du graphe (étant les méthodes des composants) doivent être contraintes par le placement des composants. Par exemple, les méthodes A et B d'un même composant ne peuvent être affectées qu'au même site d'exécution.

Pour approximer une solution, nous considérons dans notre première approche, le graphe restreint de l'exécution qui est le schéma de procédé rendu acyclique : l'ordonnanceur affecte alors un site d'exécution à chaque composant. Cette représentation se justifie si l'on considère une exécution dynamique synchronisée à chaque pas de temps où le temps d'exécution global est largement dominé par les étapes *FirstCompute* et *LastCompute* (ces deux étapes ont le même schéma d'exécution qui correspond, en l'occurrence, au schéma de procédé acyclique).

Le caractère itératif des simulations dynamiques génère des « *fausses* » dépendances, nommées anti-dépendances, relatives à la réutilisation des espaces mémoires des variables de l'ap-

plication. Ces anti-dépendances limitent le parallélisme potentiel de l'application en ajoutant des contraintes qui pourraient être évitées. Notre première approche conserve, malgré tout, ces dépendances à travers l'insertion d'une barrière de synchronisation globale à la fin de chaque pas de temps.

L'exécution distribuée d'une simulation d'un réseau de production « *offshore* » sur 5 processeurs offre en moyenne une accélération de $2,1$ sur les 90 itérations simulées.

Au regard des optimisations et techniques laissées en suspend dans la conception de ce premier prototype, nous avons développé un second simulateur ayant comme base l'environnement d'exécution KAAPI. KAAPI offre nativement une technique optimale de gestion des anti-dépendances grâce au maintien d'une représentation causalement connectée, sous la forme d'un graphe de flot de données, à l'exécution. Toutefois, seule une technique d'ordonnancement par vol de travail est implémentée. Elle est prouvée efficace dans le cadre des applications récursives à grain fin exposant un haut degré de parallélisme. Les simulations de procédés ne rentrent pas dans cette classe d'applications. Le chapitre 9 présente l'extension apportée à KAAPI afin de l'ouvrir aux techniques « *statique* » d'exécution. Cette extension intègre principalement deux nouvelles fonctionnalités à KAAPI. La première regroupe les traitements à effectuer préalablement à l'exécution et reprend les concepts implémentés dans le premier prototype. A partir de la représentation abstraite de l'application, un calcul est appliqué sur le graphe de flot de données en vue de construire des flots d'exécution autonomes. Une fois distribués et démarrés, ils ne requièrent que peu d'intéractions avec l'entité centralisée de contrôle. Elle reste cependant indispensable lors des phases de redistribution des données d'une étape à une autre. Cette redistribution représente notre seconde contribution à KAAPI. Notre implémentation, bien que nécessitant une synchronisation avec les autres processus, permet de débuter l'exécution des tâches prêtes de l'application tout en réalisant la redistribution en parallèle.

Basé sur ce nouvel environnement, nous étudions, dans le chapitre 10, l'impact du placement des composants proposé par les ordonnanceurs ETF et DSC sur le temps d'exécution distribuée. Pour cela, trois modèles de coûts (« *uniforme*», « *perturbé* » et « *stationnaire* ») ainsi que deux types de parallélisme (« *topologique* » et « *pipeline* ») sont examinés. Les meilleures heuristiques nous permettent d'obtenir un gain de 8% par rapport aux performances mesurées avec notre premier prototype.

Nous proposons, en guise de synthèse, de résumer l'impact du standard CAPE-OPEN sur la distribution du calcul des applications du génie des procédés. De par l'approche par composants logiciels prônée par le standard, les composants métier prenant part à l'exécution d'une simulation peuvent être exécutés en dehors du moteur exécutif de l'environnement de couplage de codes. Cette propriété, définissant qu'un composant est une entité d'exécution et de déploiement, est utilisée à bon escient afin de distribuer le calcul sur une grappe. Toutefois, l'obtention de performances dépend du parallélisme exposé par le schéma de calcul de l'application qui est défini par les approches de résolution. Bien que proposant des spécifications d'interfaces pour les composants métier et non pas une approche de résolution particulière, le standard CAPE-OPEN recense les approches couramment employées et propose les interfaces et méthodes nécessaires à leur résolution. Celles couramment employées dans l'industrie sont les approches modulaire séquentielle, dans le cadre des simulations statiques, et hybride, dans le cadre des simulations dynamiques. Par

nature, une approche modulaire séquentielle limite fortement le parallélisme. L'approche hybride se base fortement sur cette dernière et les mêmes conclusions s'appliquent. Historiquement implémentée car offrant le meilleur compromis temps de calcul/stabilité, l'exécution distribuée des simulations CAPE doit se tourner vers d'autres approches telles que celles étudiées dans le chapitre 5, page 90 (approches modulaire simultanée et orientée équations) qui proposent un meilleur parallélisme.

11.4 Perspectives

Cette section présente les champs d'investigation à explorer suite à nos travaux.

11.4.1 Simulations des procédés CAPE-OPEN

Dans cette thèse, nous avons montré que le parallélisme lié aux approches numériques implémentées dans INDISS, et également dans la majorité des environnements commerciaux de simulations, est limité. Nous distinguons trois approches pour extraire un plus grand parallélisme et, de ce fait, réduire le temps de simulation.

- Les cas tests utilisés pour valider expérimentalement notre approche simulent exclusivement des installations amont (*i.e.*, avant raffinage du brut) où le nombre d'opérations unitaires est faible. Il serait des plus intéressants de quantifier le gain potentiel de notre environnement sur d'autres schémas de procédés composés de plusieurs milliers d'opérations unitaires. Cette augmentation de la complexité du problème (en terme de tâches) devrait générer un plus fort parallélisme. Par exemple, nous avons quantifié le parallélisme potentiel d'une installation aval composée de plus de 500 opérations unitaires. Les résultats préliminaires montrent que le parallélisme maximal développé par ce cas métier selon l'approche de résolution d'INDISS est de 16. Cette accroissement reste toutefois limité et il est nécessaire de s'orienter vers d'autres sources de parallélisme.
- L'exploitation d'autres méthodes numériques exposant différents schémas d'exécution sont à envisager. Par exemple, une approche de résolution orientée équation nécessitant la parallélisation du solveur est prometteuse (chapitre 5, page 90).
- Une autre technique permettant de disposer d'un plus grand parallélisme tout en conservant l'approche de résolution communément employée dans l'industrie, repose sur la parallélisation des composants. L'exécution d'une simulation de procédés proposerait ainsi deux niveaux de parallélisme. Le premier, de bas niveau, serait composant et une approche telle que celle développée par PaCO++ peut être applicable. Le second, de haut niveau, s'appuierait sur la méthodologie présentée dans ce rapport de thèse.

11.4.2 Applications itératives

L'extension à KAAPI, présentée dans le chapitre 9, fut développée dans le but de proposer un environnement de tests efficace et optimisé pour la simulation de procédés. Toutefois, son champ d'utilisation n'est pas restreint à ces simulations. Ainsi, des applications de décomposition de domaines, de type Jacobi1D ou Jacobi2D ou bien encore de multiplications matrice/matrice, matrice/vecteur, où la matrice est creuse et de grande taille, peuvent être exécutées. Afin d'améliorer

les gains potentiels, notre extension à KAAPI doit être optimisée. Actuellement, une seule étape KAAPI (correspondant au développement de plusieurs itérations applicatives) ne peut être active à un instant t. Une barrière de synchronisation est insérée entre l'exécution de deux étapes. Cette dernière peut être supprimer en implémentant, par exemple, un tampon de réception pour chaque donnée afin d'anticiper sur le calcul des étapes suivantes.

En outre, un effort important doit être mené dans le cadre de la définition de la politique locale d'exécution. En effet, cette politique est actuellement définie à travers l'ordre de référence RFO de KAAPI. La technique, présentée dans le chapitre 10 page 170, doit être automatisée. L'étape de génération (section 9.3.1, page 155), par exemple, pourrait reconstruire chaque sous-graphe DFG en tenant compte des dates de commencement des tâches calculées par les ordonnanceurs tels que ETF ou DSC. avec un nouvel ordre calculé par l'ordonnanceur. Dans le cas où l'ordonnanceur ne fournit que le placement des tâches (*e.g.* METIS, SCOTCH), une heuristique d'ordonnancement local doit être implantée. Une technique nommée SCO (*SCheduling Order*) est actuellement en cours d'implémentation. Elle consiste à exécuter les tâches de communication au plus tôt favorisant alors le recouvrement des communications par du calcul.

Pour conclure, nous avons remarqué dans le chapitre 10 page 172, qu'une optimisation pouvait être menée sur le pré-traitement « *statique* » en distribuant les partitions par ordre décroissant de tâches et ceci afin de recouvrir les temps de communication par du calcul. Toutefois, une autre optimisation consisterait à effectuer la totalité de ce pré-traitement en parallèle. En considérant que l'ordonnancement produit soit déterministe, chacun des processeurs peut exécuter cette tâche et par la suite générer la partition dont il aura la charge. Dès les premières phases de conception, les interfaces des méthodes réalisant les traitements de partitionnement et de génération tiennent compte de cette ouverture. Le code devrait être fonctionnel et seules la définition conceptuelle de même qu'une campagne de tests approfondie doivent être entreprises.

11.4.3 Ordonnancement

La majorité des techniques d'ordonnancement s'appuie sur la définition d'un graphe *acyclique* de tâches. Dans notre cas d'étude, nous avons testé deux types de parallélisme où le graphe est ordonnancé soit à partir de la représentation du calcul d'une seule itération (parallélisme topologique), soit à partir du développement de deux pas de temps restreint aux deux étapes *LastCompute* et *FirstCompute*. De par le schéma de calcul induit par l'approche de résolution INDISS (barrière de synchronisation due au solveur centralisé) ainsi que la contrainte de placement liée à la définition d'un composant (une tâche du graphe est un appel de méthodes sur un composant et chaque méthode ne peut être invoquée que sur ce composant), il n'était pas nécessaire de construire d'autres représentations et le développement de l'application sur plusieurs itérations, dans un but d'ordonnancement, est dispensable.

Toutefois, dans le cadre plus large des applications itératives, des techniques d'ordonnancement cyclique [59] peuvent être appliquées. L'heuristique développée calcule une borne théorique sur le nombre de développements du graphe qui permet d'extraire un parallélisme maximal. Sur ce graphe développé, un ordonnanceur cyclique implémente une technique de décalage d'instructions générant un pré et un post-traitement au régime permanent de l'application. Grâce à cette technique, les contraintes de précédence du graphe en régime permanent sont minimisées et offre *a priori* un parallélisme plus grand. Les ordonnanceurs cycliques ne calculent pas, en général, les

politiques globale et locale d'exécution. Ainsi, un ordonnanceur « *classique* » acyclique devra être utilisé en complément.

L'intégration de ces techniques dans KAAPI a fait l'objet d'un stage de Master [5]. L'implantation est effective et nécessite toutefois une intégration dans la version actuelle de KAAPI ainsi qu'une campagne de tests validant l'approche et quantifiant les gains.

11.4.4 Adaptabilité

Actuellement, un axe de recherche ouvert vise à proposer des solutions pour le caractère dynamique des architectures matérielles de type grappe et grille de calcul [25, 29, 113, 62]. Dans ce cadre, nous nous intéressons à la disparition et à l'ajout de ressources en cours d'exécution. Par exemple, considérons une grappe de calcul dont l'utilisation est partagée entre de nombreux utilisateurs. Un système de soumission de *jobs* organise les requêtes d'accès aux ressources suivant une politique d'équité entre les utilisateurs. Compte tenu du succès de ce type d'architectures, il est très difficile de disposer de la totalité des nœuds de caclul tout en satisfaisant l'équité des accès aux ressources. Dans le contexte des applications nécessitant un grand temps de calcul et exposant un fort parallélisme, il peut être avantageux de tirer profit de la moindre libération de ressources d'un autre utilisateur et de l'agglomérer aux ressources affectées à notre application en cours d'exécution. Cette politique d'exécution « *gloutonne* » est naturellement implémentée lors d'une heuristique d'ordonnancement par vol de travail (les nouveaux processeurs sont en famine et vont alors voler du travail sur les autres processeurs).

Dans ce contexte, KAAPI et notre extension permettent de réordonnancer à la volée le calcul restant tout en bénéficiant des propriétés d'exécution présentées dans ce manuscrit. Cette étude a fait l'objet d'une publication [63]. Dans cet article, nous présentons les différents scénarios où notre approche offre une alternative intéressante aux problématiques d'adaptabilité. L'ajout ou le retrait de ressources en cours d'exécution nécessite le calcul d'un nouveau placement qui est rendu possible d'après la représentation DFG causalement connectée à l'exécution offerte par KAAPI. De nouvelles partitions sont ainsi créées en fonction du calcul restant à effectuer et du nouveau nombre de ressources disponibles.

Bibliographie

[1] N. Abdel-Jabbar, B. Carnahan, and C. Kravaris. A multirate parallel-modular algorithm for dynamic process simulation using distributed memory multicomputers. *Computers & Chemical Engineering*, 23(6) :733–761, 1999.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks : Shared memory computing on networks of workstations. *IEEE Computer*, 29(2) :18–28, 1996.

[3] AspenTech. Aspen plus : User models. http://support.aspentech.com/Public%5CDocuments%5CEngineering%5CAspen%20Plus%5C2006/AspenPlusUserModels2006-Ref.pdf, 2006.

[4] InfiniBand Trade Association. Infiniband architecture specification, release 1.0, October 2000.

[5] J.-A. Aufauvre. Ordonnancement cyclique dans kaapi. Master's thesis, Université Joseph Fourier, 2006.

[6] P. Banks, P. Edwards, J.C. Rodriguez, R. Martin, M. Williams, S. Cebollero, and M. Halloran. CAPE-OPEN : OPEN Interface Specifications – Unit Operations V4.0. http://www.colan.org, 2001.

[7] R.W. Barkley and R. L. Motard. Decomposition of nets. *The Chemical Engineering Journal*, 3 :265–275, 1972.

[8] P. I. Barton. The equation oriented strategy for process flowsheeting. Technical report, Massachusetts Institute of Technology, Dept. Chem. Eng., Cambridge, MA 02139, March 2000.

[9] O. Beaumont, V. Boudet, P.-F. Dutot, Y. Robert, and D. Trystram. *Informatique répartie : architecture, parallélisme et système*, chapter 3, "Fondements théoriques pour la conception d'algorithmes efficaces de gestion de ressources", pages 63–94. Hermès Publications, 2005.

[10] J. P. Belaud. *CAPE-OPEN, un Standard pour l'Interopérabilité et l'Intégration des Composants Logiciels de l'Ingénierie des Procédés*. PhD thesis, Institut National Polytechnique de Toulouse, 2002.

[11] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation : numerical methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[12] X. Besseron, L. Pigeon, T. Gautier, and S. Jafar. Un protocole de sauvegarde/reprise coordonné pour les applications à flot de données reconfigurable. In *Proceedings des 17èmes rencontres francophones du parallélisme (RenPar'17)*, Perpignan, France, October 2006.

[13] R. J. Best. Process simulation using parallel computers. *Transactions of the Institute of Chemical Engineers*, 68A :418–428, September 1990.

[14] L. T. Biegler and Y. D. Lang. Process Systems Enterprise and CAPE-OPEN. In *Proceedings of the 2nd Annual U.S. CAPE-OPEN Meeting*, Morgantown, U.S.A., 25–26 May 2005.

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk : an efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1) :55–69, 1996.

[16] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5) :720–748, 1999.

[17] R. D. Blumofe and P. A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. In *Proceedings of the USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, Anaheim, California, January 1997.

[18] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet : A gigabit-per-second local area network. *IEEE Micro*, 15(1) :29–36, 1995.

[19] J. Borchardt. Newton-type decomposition methods in large-scale dynamic process simulation. *Computer & Chemical Engineering*, 25(7-8) :951–961, 2001.

[20] P. Boulet, A. Darte, G.-L. Silber, and F. Vivien. Loop parallelization algorithms : From parallelism extraction to code generation. *Parallel Computing*, 24(3-4) :421–444, May 1998. Special issue on "Languages and Compilers for Parallel Computers".

[21] D. Box. *Essential COM - 3rd printing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[22] B. Braunschweig. Vers la simulation numerique par agents apprenants. Habilitation à diriger des recherches. IFP, Université Pierre et Marie Curie, Paris, 2002.

[23] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. North-Holland, New York, 1989.

[24] P. Brinch Hansen. Structured multiprogramming. *Commun. ACM*, 15(7) :574–578, 1972.

[25] J. Buisson, F. André, and J.-L. Pazat. Dynamic adaptation for grid computing. In Peter M. A. Sloot, Alfons G. Hoekstra, Thierry Priol, Alexander Reinefeld, and Marian Bubak, editors, *Advances in Grid Computing - EGC 2005 (European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005, Revised Selected Papers)*, volume 3470 of *LNCS*, pages 538–547, Amsterdam, June 2005. Springer-Verlag.

[26] P.-Y. Calland, A. Darte, Y. Robert, and F. Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithm. *Parallel Computing*, 23(1-2) :251–266, 1997.

[27] C. D. Carothers, R. M. Fujimoto, R. M. Weatherly, and A. L. Wilson. Design and implementation of HLA time management in the RTI version F.0. In *WSC '97 : Proceedings of the 29th conference on Winter simulation*, pages 373–380, 1997.

[28] H. S. Chen and M. A. Stadtherr. A simultaneous modular approach to process flowsheeting and optimization : I. theory and implementation. *AIChE Journal*, 31, No. 11 :1843–1856, 1985.

[29] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self-adapting software for numerical linear algebra and lapack for clusters. *Parallel Compututing*, 29(11-12) :1723–1743, 2003.

[30] P. Chrétienne and C. Picouleau. Scheduling with communication delays : A survey. In P. Chretienne, E. G. Jr Coffman, J. K. Lenstra, and Z. Liu, editors, *Scheduling Theory and its Applications*, chapter 4, pages 314–325. Wiley, 1995.

[31] H. N. Cofer and M. A. Stadtherr. Reliability of iterative linear equation solvers in chemical process simulation. *Computers & chemical engineering*, 20(9) :1123–1132, 1996.

[32] V.-D. Cung, P. Fraigniaud, T. Gautier, and D. Trystram. De l'algorithme au support. In *Actes de l'école d'hiver ICaRE'97 (Conception et mise en oeuvre d'applications parallèles irrégulières de grande taille)*, Aussois, December 1997.

[33] DECHEMA. http://www.dechema.de/IK_CAPE_THERMO-lang-en.html. Website, February 2007.

[34] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier. High-Performance Computing : Clusters, Constellations, MPPs, and Future Directions. *Computing in Science and Engineering*, 07(2) :51–59, 2005.

[35] M. Doreille. *Athapascan-1 : vers un modèle de programmation parallèle adapté au calcul scientifique*. PhD thesis, Institut National Polytechnique de Grenoble, 1999.

[36] K. H. Duerre and C. Bumb. Implementing ASPEN on the Cray Computer. Los Alamos National Laboratory, 1981.

[37] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC '82 : Proceedings of the 19th conference on Design automation*, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.

[38] P. Fjallstrom. Algorithms for graph partitioning : A survey, 1998. Linkoping Electronic Atricles in Computer and Information Science, 3.

[39] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21 :948–960, 1972.

[40] Message Passing Interface Forum. MPI : A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4) :159–416, 1994.

[41] F. Galilée. *Athapascan-1 : Interprétation Distribuée du Flot de Données d'un Programme Parallèle*. PhD thesis, Institut National Polytechnique de Grenoble, 1999.

[42] F. Galilée, J.-L. Roch, G. H. Cavalheiro, and M. Doreille. Athapascan-1 : On-Line Building Data Flow Graph in a Parallel Language. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, page 88, Washington, DC, USA, 1998. IEEE Computer Society.

[43] T. Gautier and H. R. Hamidi. Automatic re-scheduling of dependencies in a RPC-based grid. In *ICS '04 : Proceedings of the 18th annual international conference on Supercomputing*, pages 89–94, Malo, France, 2004.

[44] T. Gautier, R. Revire, and J.-L. Roch. Athapascan : API for asynchronous parallel programming. Technical Report RR-0276, APACHE, INRIA Rhône-Alpes, February 2003.

[45] T. Gautier, R. Revire, and J.-L. Roch. Athapscan : Api for asynchronous parallel programming. Technical Report RR-0276, INRIA Rhônes-Alpes, February 2003.

[46] T. Gautier, J.-L. Roch, and G. Villard. Regular versus irregular problems and algorithms. In *IRREGULAR '95 : Proceedings of the Second International Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 1–25, London, UK, 1995. Springer-Verlag.

[47] C. W. Gear and F. L. Juang. The speed of waveform methods for ODEs. In Renato Spigler, editor, *Applied and industrial mathematics*, pages 37–48, Netherlands, 1991. Kluwer Academic Publishers.

[48] J. M. Geib, C. Gransart, and P. Merle. *CORBA : Des concepts à la pratique*. Dunod, 1999.

[49] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM : Parallel Virtual Machine : A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[50] R. Graham. Bounds for certain multiprocessing anomalies. *Bell Systems Technical Journal*, 45 :1563–1581, 1966.

[51] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics*, 17(2) :416–429, 1969.

[52] Grid5000. http://www.grid5000.fr/. Website, February 2007.

[53] F. Grund, K. Ehrhardt, J. Borchardt, and D. Horn. Heterogeneous dynamic process flowsheet simulation of chemical plants. *Mathematics – Key Technology for the Future II*, pages 184–193, 2003.

[54] H.-R. Hamidi. *Couplage à hautes performances de codes parallèles et distribués*. PhD thesis, Institut National Polytechnique de Grenoble (INPG), 2005.

[55] J. Handy. *The cache memory book*. Academic Press Professional, Inc., San Diego, CA, USA, 1993.

[56] B. K. Harrison. Computational inefficiencies in sequential modular flowsheeting. *Computers and Chemical Engineering*, 16(7) :637–639, 1992.

[57] H. Hellwagner and A. Reinefeld, editors. *SCI : Scalable Coherent Interface, Architecture and Software for High-Performance Compute Clusters*, volume 1734 of *Lecture Notes in Computer Science*. Springer, 1999.

[58] C. A. R. Hoare. Towards a theory of parallel programming. In C.A.R. Hoare and R.H. Perrott, editors, *Operating System Techniques*, pages 61–71. Academic Press, 1972.

[59] G. Huard. *Algorithmique du décalage d'instructions*. PhD thesis, École Normale Supérieure de Lyon, 2001.

[60] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2) :244–257, 1989.

[61] IEEE. *IEEE 1003.1c-1995 : Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2 : Threads Extension (C Language)*. IEEE Computer Society Press, 1995.

[62] S. Jafar. *Programmation des systèmes parallèles distribués : tolérance aux pannes, résilience et adaptabilité*. PhD thesis, Institut National Polytechnique de Grenoble, 2006.

[63] S. Jafar, L. Pigeon, T. Gautier, and J.-L. Roch. Self-adaptation of parallel applications in heterogeneous and dynamic architectures. In IEEE, editor, *ICTTA'06 IEEE Conference on Information and Communication Technologies : from Theory to Applications*, pages 3347–3352, Damascus, Syria, April 2006.

[64] D. Jerome. CAPE-OPEN in PRO/II. In *Proceedings of the 2nd Annual U.S. CAPE-OPEN Meeting*, Morgantown, U.S., 25–26 May 2005.

[65] E. E. Johnson. Completing an mimd multiprocessor taxonomy. *SIGARCH Comput. Archit. News*, 16(3) :44–47, 1988.

[66] KAAPI. KAAPI : Kernel for Adaptative, Asynchronous Parallel and Interactive programming. http://gforge.inria.fr/projects/kaapi/.

[67] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Supercomputing '95 : Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 29, New York, NY, USA, 1995. ACM Press.

[68] G. Karypis and V. Kumar. METIS : a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Technical report, Department of Computer Science/Army HPC Research Centre, University of Minnesota, Minneapolis, MN 55455, September 1998.

[69] K. Keahey and D. Gannon. PARDIS : A Parallel Approach to CORBA. In *HPDC '97 : Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC '97)*, page 31, Washington, DC, USA, 1997. IEEE Computer Society.

[70] P. J. Keleher. *Lazy release consistency for distributed shared memory*. PhD thesis, Rice University, Houston, TX, USA, 1994.

[71] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(1) :291–307, 1970.

[72] V. Kulikova, H. Briesena, R. Groscha, A. Yanga, L. von Wedelb, and W. Marquardt. Modular dynamic simulation for integrated particulate processes by means of tool integration. *Chemical Engineering Science*, 60-7(7) :2069–2083, 2005.

[73] A. Le Kahc Nhien. *Définition et évaluation d'INUKTITUT : une interface pour l'environnement de programmation parallèle asynchrone Athapascan*. PhD thesis, Institut National Polytechnique de Grenoble, 2005.

[74] E. Lelarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli. The waveform relaxation method for time-domain analysis of large scale integrated circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 1(3) :131–145, 1982.

[75] T. G. Lewis. *Foundations of Parallel Programming : A Machine-Independent Approach*. IEEE Computer Society Press, 1994.

[76] A. Mainwaring and D. Culler. Active message applications programming interface. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1996.

[77] J. U. Mallya, S. E. Zitney, S. Choudhary, and M. A. Stadtherr. Matrix reordering effects on a parallel frontal solver for large scale process simulation. *Computers & chemical engineering*, 23(4-5) :585–593, 1999.

[78] A. McGough and M. Halloran. Overview of AspenTech's CAPE-OPEN Support. In *Proceedings of the 2nd Annual U.S. CAPE-OPEN Meeting*, Morgantown, U.S., 25–26 May 2005.

[79] Message Passing Interface Forum. MPI2 : A message passing interface standard. *High Performance Computing Applications*, 12(1–2) :1–299, 1998.

[80] H. I. Moe and T. Hertzberg. Advanced computer architectures applied in dynamic process simulation : a review. *Computers and Chemical Engineering*, 18 Suppl :S375–S384, 1994.

[81] F. Mueller. A library implementation of POSIX threads under unix. In *Proceedings of theWinter 1993 USENIX Technical Conference and Exhibition*, pages 29–41, San Diego, CA, USA, 1993.

[82] OMG. *The Common Object Request Broker : Architecture And Specification*. Object Management Group, December 2001. Revision 2.6.

[83] OMG. *Data Parallel CORBA Specification*. Object Management Group, November 2001. ptc/2001-11-09.

[84] OMG. *CORBA Component Model Specification*. Object Management Group, April 2006. version 4.

[85] OmniORB. http://omniorb.sourceforge.net/. Website, February 2007.

[86] S. Pakin, V. Karamcheti, and A. A. Chien. Fast messages : Efficient, portable communication for workstation clusters and mpps. *IEEE Parallel Distrib. Technol.*, 5(2) :60–73, 1997.

[87] C. Pantelides, B. Keeping, J. Bernier, and C. Gautreau. CAPE-OPEN : OPEN Interface Specifications – Numerical Solvers V1.08. http://www.colan.org, 1999.

[88] F. Pellegrini and J. Roman. SCOTCH : A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *HPCN Europe 1996 : Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pages 493–498, London, UK, 1996. Springer-Verlag.

[89] F. Petrini, W.-C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The quadrics network (qsnet) : High-performance clustering technology. In *HOTI '01 : Proceedings of the The Ninth Symposium on High Performance Interconnects (HOTI '01)*, page 125, Washington, DC, USA, 2001. IEEE Computer Society.

[90] L. Pigeon. Exécution CAPE-OPEN distribuée d'un champ de production sur un cluster de PC. Journée IEP (Informatique & Procédé), May 2006. Communication orale, Grenoble, France.

[91] L. Pigeon, B. Braunschweig, T. Gautier, and P. Roux. Simulation dynamique d'un réseau de production sur cluster de PC. In *Colloque Systèmes d'Information, Modélisation, Optimisation et Commande en Génie des Procédés*, Toulouse, France, 2006. Sélectionné pour publication dans Journal of Chemical Engineering and Processing.

[92] L. Pigeon, P. Roux, B. Braunschweig, and T. Gautier. Dynamic CAPE-OPEN Simulation Approach on Cluster Oriented Architecture. In *Proceedings of the 3rd Annual U.S. CAPE-OPEN Meeting*, San Francisco, USA, 2006.

[93] L. Pigeon, P. Roux, B. Braunschweig, T. Gautier, and Paen D. Distributed Dynamic CAPE-OPEN Simulation of Oilfield Production Networks on Clusters of PCs. CAPE-OPEN Interoperability Showcase and Workshop events, March 2006. Communication orale, Cannes, France.

[94] L. Pigeon, L. Testard, T. Gautier, and P. Roux. Towards A Distributed High Performance Computing Environment For CAPE-OPEN Standard. In *Proceedings of 7th World Congress of Chemical Engineering (WCCE)*, page 339, Glasgow, Scotland, 14–18 July 2005.

[95] J. W. Ponton, E. M. Johnston, J. M. Forsyth, A. Matheson, and F. M. Rutherford. Real time dynamic simulation using networked computers. *Computers & chemical engineering*, 13(11-12) :1245–1253, 1989.

[96] C. Pérez, T. Priol, and A. Ribes. PaCO++ : A parallel object model for high performance distributed systems. In *Distributed Object and Component-based Software Systems Minitrack in the Software Technology Track of the 37th Hawaii International Conference on System Sciences (HICSS-37)*, page 274a, Big Island, Hawaii, USA, January 2004. IEEE Computer Society Press.

[97] P. Renault, T. Uslander, L. Indesteege, J. Theunisz, T. Malik, and L. Jourda. OPERA : Operators training distributed real-time simulations, Final Report. Technical report, ESPRIT Project 24950, 2000.

[98] R. Revire. *Ordonnancement de graphe dynamique de tâches sur architecture de grande taille. Régulation par dégénération séquentielle et distribuée.* PhD thesis, Institut National Polytechnique de Grenoble, 2004.

[99] RSI. INDISS : INDustrial Integrated Simulation Software. Parc Technologique de Pré Milliet - 38330 Montbonnot France (http://www.simulationrsi.net).

[100] Y. Saad and M. H. Schultz. Gmres : a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3) :856–869, 1986.

[101] G. Schopfer, A. Yang, L. von Wedel, and W. Marquardt. CHEOPS : A tool-integration platform for chemical process modelling and simulation. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3) :186–202, 2004.

[102] J. A. Scott. Parallel frontal solvers for large sparse linear systems. *ACM Trans. Math. Softw.*, 29(4) :395–417, 2003.

[103] SGI. Standard Template Library Programmer's Guide. Website, 2007. http://www.sgi.com/tech/stl/index.html.

[104] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2) :123–169, 1998.

[105] F. G. Smith and R. A. Dimenna. Simulation of a batch chemical process using parallel computing with pvm and speedup. *Computers & Chemical Engineering*, 28(9) :1649–1659, 2004.

[106] Java Soft. Java Remote Method Invocation Specification, revision 1.5, JDK 1.2 edition, October 1998.

[107] Sun Microsystems, Inc. RFC 1057 : RPC : Remote procedure call protocol specification : Version 2. http://www.faqs.org/rfcs/rfc1057.html, June 1988.

[108] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.3 Reference Manual*, January 2007.

[109] A. S. Tanenbaum. *Distributed operating systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[110] Top500. http://www.top500.org/. Website, November 2006.

[111] R. S. Upadhye and E. A. Grens. Selection of decompositions for chemical process simulation. *AIChE Journal*, 21(1) :136–143, 1975.

[112] A. Vacher, S. Dechelotte, and O. Baudouin. ProSimPlus : New Cape-Open Capabilities. In *Proceedings of the 3rd Annual U.S. CAPE-OPEN Conference*, San Francisco, U.S., 12–17 November 2006.

[113] S. Vadhiyar and J. Dongarra. Self adaptivity in grid computing : Research articles. *Concurrency and Computation : Practice & Experience*, 17(2-4) :235–257, 2005.

[114] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8) :103–111, 1990.

[115] J. van Baten. A look into the CAPE-OPEN kitchen of COCO. In *CAPE-OPEN Interoperability Showcase and Workshop Events*, Cannes, France, 2006.

[116] J. A. Vegeais and M. A. Stadtherr. Parallel processing strategies for chemical process flowsheeting. *AIChE Journal*, 38(9) :1399–1407, 1992.

[117] J. H. Wegstein. Accelerating convergence of iterative processes. *Communication of the ACM*, 1(6) :9–13, 1958.

[118] P. Werstein, M. Pethick, and Z. Huang. A Performance Comparison of DSM, PVM, and MPI. In *Proceedings of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 476–482, Chengdu, China, 2003. SW Jiaotong University.

[119] A. W. Westerberg, H. P. Hutchinson, R. L. Motard, and P. Winter. *Process Flowsheeting*. Cambridge University Press, New York, 1979.

[120] A. W. Westerberg and P. C. Piela. Equational-based process modeling. Technical report, Departement of Chemical Engineering and the Engineering Design Research Center - Carnegie Mellon University, Pittsburgh, 1994. Available from www.cs.cmu.edu/~ascendFTP/pdffiles/ProcessModeling.pdf.

[121] T. Yang. *Scheduling and code generation for parallel architectures*. PhD thesis, Rutgers University, New Brunswick, NJ, USA, 1993.

[122] T. Yang and A. Gerasoulis. PYRROS : static task scheduling and code generation for message passing multiprocessors. In *ICS '92 : Proceedings of the 6th international conference on Supercomputing*, pages 428–437. ACM Press, 1992.

[123] T. Yang and A. Gerasoulis. PYRROS : static task scheduling and code generation for message passing multiprocessors. In *ICS '92 : Proceedings of the 6th international conference on Supercomputing*, pages 428–437, Washington, D. C., United States, 1992.

[124] T. Yang and A. Gerasoulis. DSC : Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Trans. Parallel Distrib. Syst.*, 5(9) :951–967, 1994.

[125] S. E. Zitney, J. Mallya, T. A. Davis, and M. A. Stadtherr. Multifrontal vs frontal techniques for chemical process simulation on supercomputers. *Computers & Chemical Engineering*, 20(6-7) :641–646, 1996.

[126] S. E. Zitney and M. A. Stadtherr. Frontal algorithms for equation-based chemical process flowsheeting on vector and parallel computers. *Computers & Chemical Engineering*, 17(4) :319–338, 1993.

Résumé

Environnement interopérable distribué pour les simulations numériques avec composants CAPE-OPEN.

La complexité des applications numériques de calcul scientifique ne cesse de croître. Cette difficulté revêt alors deux formes. La première est une complexité logicielle qui nécessite l'intégration de divers codes de calcul toujours plus sophistiqués et spécialisés à la simulation de phénomènes physiques complexes. La seconde forme de complexité est *calculatoire* où les composants de calcul nécessitent toujours plus de ressources et de capacité de stockage afin de modéliser, entre autres, les phénomènes au plus proche de la physique « *réelle* ».

Dans le domaine de la simulation des procédés assistée par ordinateur, la complexité logicielle est masquée par le standard CAPE-OPEN qui répond aux besoins d'intégration de codes tiers. Il propose des spécifications d'interfaces, basées sur une approche par composants logiciels tels que DCOM ou CORBA. Cette thèse apporte une solution à la complexité *calculatoire*. Pour cela, nous étudions le problème de la distribution de la charge de calcul des simulations des procédés sur des architectures de type grappe de calcul dont les composants sont au standard CAPE-OPEN. Une exécution distribuée performante requiert la distribution des activités concurrentes de l'application tout en minimisant le volume de données à échanger *via* le support de communication.

Dans ce contexte, nous présentons une analyse fine du schéma d'exécution des simulations de procédés qui conduit à la conception de deux environnements distribués d'exécution. Le premier nous a permis de quantifier le gain atteignable sur une grappe de calcul à travers la simulation de plusieurs cas tests métier. Toutefois, les contraintes technologiques industrielles se sont avérées peu propices à l'implémentation d'un environnement distribué visant à s'approcher de l'optimal. Par conséquent, la définition d'un second prototype basé sur le moteur exécutif KAAPI a été mené à bien. Afin de répondre à nos besoins, nous l'avons étendu aux techniques « *statiques* » d'exécution. Fort de cet environnement, nous avons entrepris d'étudier différentes politiques d'ordonnancement. L'environnement KAAPI couplé à notre extension ouvre de larges perspectives d'études dans le cadre plus large des applications numériques de calcul scientifique.

Mots-clés : Environnement distribué d'exécution, simulation des procédés, CAPE-OPEN, KAAPI, ordonnancement.

Abstract

Distributed interoperable environment for numerical simulations using CAPE-OPEN components.

The complexity of numerical simulations has constantly grown by means of software and computation purposes. The former, the software complexity, is arising from specialized codes which aim at simulating complex physical phenomena. The latter, the computation complexity, is of general concern regarding scientific applications because approaching "*real*" physical behaviour requires more and more storage and power capacities.

The CAPE-OPEN standard, in the process simulation area, provides a solution to the needs of tierce codes integration by hiding software complexity. As a consequence, interface specifications are set and software components such as DCOM or CORBA are used. In this work, we study the problem of computation load distribution over clusters which software components are CAPE-OPEN compliant. An efficient distributed execution needs to share concurrent activities from the application while minimizing exchanged data loads through the network.

In this context, we present a tightly analysis of process simulations computational scheme that leads to the implementation of two computational distributed environments. The former is used to quantify the benefit on business test cases simulations over a cluster. Nevertheless, industrial technologies limit the optimization implementations necessary to reach the optimal. Then, the latter prototype has been implemented on top of KAAPI runtime. To come to our expectations, KAAPI is extended in order to manage "*static*" scheduling and various scheduling policies are studied. The KAAPI environment coupled with the extension that we develop is the first step to several experiments in the more general scope of numerical applications.

Keywords : Distributed computational environment, process simulation, CAPE-OPEN, KAAPI, scheduling.