THÈSE

présentée à

L'Université PARIS-XIII
Institut Galilée

pour obtenir le titre de

Docteur en sciences

Spécialité
INFORMATIQUE

par

Annie Vicard

Formalisation et optimisation des systèmes informatiques distribués temps réel embarqués

Soutenue le 5 Juillet 1999 devant le jury composé de :

M. Marc BUI	Rapporteur
M. Christian LAVAULT	Examinateur
M. Yves SOREL	Examinateur
M. Denis TRYSTRAM	Rapporteur
M. Vassilis ZISSIMOPOULOS	Président

Table des matières

Remerciements

J'exprime toute ma reconnaissance à Yves Sorel, Directeur de Recherche à l'INRIA, grâce à qui j'ai pu mener à bien cette thèse. Il m'a accueillie dans son équipe, guidée et encadrée en m'accordant sa confiance. Sa compétence et sa rigueur scientifique, m'ont permis d'accomplir ce travail dans de très bonnes conditions. Je le remercie chaleureusement de ne pas avoir mesuré le temps qu'il a passé avec moi pour la rédaction de cette thèse.

Ma reconnaissance va également aux autres membres du projet Sosso en particulier Christophe Lavarenne, Ingénieur de Recherche avec qui j'ai eu de nombreuses discussions animées mais toujours fructueuses. Que Thierry Grandpierre et Rémy Kocik, tous deux thésards, et Mihaela Sighireanu, chercheur post-doctorante, qui m'ont souvent écoutée, encouragée et aidée, soient assurés de mon amicale gratitude.

Je remercie sincèrement Christian Lavault, Professeur à l'Université Paris XIII, qui a accepté de m'inscrire en thèse sous sa direction et a suivi avec intérêt mes travaux, ainsi que Vassilis Zissimopoulos, Professeur à l'Université Paris XIII qui a bien voulu être membre du jury.

Je tiens à remercier Marc Bui, Professeur à l'Université Paris VIII et Denis Trystram, Professeur à l'INPG de Grenoble pour leurs remarques, suggestions et conseils. Ils ont accepté de juger cette thèse et me font l'honneur d'en être les rapporteurs.

Je dis un grand merci à tous les membres du bâtiment 12 pour les nombreux bons moments passés en leur compagnie et à Xavier Blondel, Ingénieur Expert à l'INRIA, qui a relu et commenté ma thèse avec attention.

Enfin, que mes amis et ma famille et surtout Philippe, qui n'ont cessé de m'encourager et de me réconforter dans les moments difficiles, trouvent ici l'expression de mon affectueuse gratitude.

Notations

Nous présentons dans ce chapitre les notations et les notions de base utilisées dans cette thèse.

Ensembles

On utilisera dans la suite les notations ensemblistes usuelles suivantes :

notation	*signification*
$\|E\|$	cardinal de l'ensemble E
$\{x \in E/Q(x)\}$	ensemble des éléments x de E qui satisfont la propriété $Q(x)$
$E_1 \times E_2$	produit cartésien des ensembles E_1 et E_2
$E_1 \times \ldots \times E_n$	produit cartésien généralisé à une famille finie d'ensembles E_1, ..., E_n
$(a_1,...,a_n)$	liste d'éléments indexés de 1 à n
$\mathcal{P}(E)$	ensemble des parties de l'ensemble E

Fonctions

Pour tous ensembles A et B, $f : A \to B$ dénote une *fonction* de l'ensemble A vers B, c'est-à-dire un sous-ensemble de $A \times B$ tel que, pour tout élément $a \in A$ il existe au plus un élément $b \in B$ vérifiant $(a,b) \in f$. Pour les fonctions, on utilise les notations suivantes :

- Le *domaine* de f, noté $Dom(f)$, est l'ensemble $\{a \in A/(\exists b \in B) \quad (a,b) \in f\}$.
- Une fonction $f : A \to B$ est *injective* si et seulement si :
  $\forall a_1,a_2 \in A \;\; f(a_1) = f(a_2) \Rightarrow a_1 = a_2$.
- Une fonction $f : A \to B$ est *surjective* si et seulement si : $\forall b \in B, \exists a \in A \;/\; b = f(a)$.

Relations d'ordre

Pour tout ensemble E, $R \subseteq E \times E$ est une *relation binaire* sur E. Pour indiquer qu'une paire (e,e') de $E \times E$ appartient à R, on utilise, selon les cas, l'une des notations suivantes : $(e,e') \in R$, $e \; R \; e'$, $R(e,e')$.

Si $\mathcal{R}$ est une relation de A vers A et $A_1 \subset A$ est un sous-ensemble de A, alors "$\mathcal{R} \downarrow A_1$" représente la *réduction* de $\mathcal{R}$ à l'ensemble A_1 ; cette relation est définie comme suit :
$Dom(\mathcal{R} \downarrow A_1) = A_1$ et $(\mathcal{R} \downarrow A_1) = \{(a_1,a_2)/(a_1,a_2) \in A_1 \times A_1, \quad a_1 \; \mathcal{R} \; a_2\}$.

Soit $\mathcal{R}$ une relation sur un ensemble E. Alors : R est *antisymétrique* si : $(\forall(e,e') \in E^2) \quad (e,e') \in R$ et $(e',e) \in R \Rightarrow e' = e$

Une relation $\mathcal{R}$ réflexive, antisymétrique et transitive induit un *ordre partiel sur l'ensemble* E si pour certains couples (e,e'), on a $e \,\mathcal{R}\, e'$ ou $e' \,\mathcal{R}\, e$. On dit que l'ensemble E est partiellement ordonné.

En revanche, si pour tout couple (e,e') appartenant à l'ensemble $E \times E$, on a $e \,\mathcal{R}\, e'$ ou $e' \,\mathcal{R}\, e$, la relation $\mathcal{R}$ induit *un ordre total.* E est dit totalement ordonné. Dans ce cas : $\mathcal{R} = E \times E$.

Une *extension linéaire* d'un ordre partiel $\preceq$ sur un ensemble E est un ordre total $\prec$ compatible avec l'ordre partiel, c'est-à-dire qu'il vérifie : $\preceq \subseteq \prec$.

Introduction

Cadre de l'étude

Cette thèse se situe dans le cadre des recherches menées dans le projet Sosso de l'INRIA-Rocquencourt sur les méthodes d'Adéquation Algorithme Architecture pour les systèmes informatiques temps réel [66, 76] distribués embarqués [72].

De tels systèmes se rencontrent dans de nombreux domaines d'application tels que la robotique, l'automobile, les transports aériens, les sous-marins, les télécommunications... Ils appartiennent à la catégorie des *systèmes réactifs*, appellation introduite par Harel et Pnueli dans [38]. Un système réactif est un système qui réagit continûment avec son environnement à un rythme imposé par cet environnement. Il reçoit, par l'intermédiaire de capteurs, des événements (données) en entrées provenant de l'environnement, appelés stimuli, réagit à tous ces stimuli en effectuant un certain nombre d'opérations et produit, grâce à des actionneurs, des événements en sorties utilisables par l'environnement, appelés *réactions* ou *commandes*.

Un système est dit *temps réel* [6] s'il est réactif et si en plus, son temps de réaction est borné par une valeur imposée par l'environnement. Ce délai est très variable d'un système à un autre. Temps réel dans le domaine avionique correspond à un délai de l'ordre de la micro-seconde, tandis que dans le domaine de la météorologie par exemple, le délai est de l'ordre de l'heure. Ainsi, la notion de temps réel n'est pas uniquement liée à la rapidité d'exécution mais aussi au respect d'une contrainte sur cette rapidité, sachant que si cette contrainte n'est pas respectée, cela peut rendre le système incontrôlable, conduisant éventuellement à sa destruction. Les contraintes temps réel sont de deux types : contrainte de *latence* et contrainte de *cadence*. Une latence correspond à l'intervalle de temps qui s'écoule entre l'acquisition effectuée par un capteur, qui délivre la donnée associée au stimulus, et la production de la donnée résultat par l'actionneur correspondant. La cadence est la fréquence à laquelle les capteurs acquièrent les données associées à des stimuli consécutifs.

Un des moyens de respecter les contraintes temps réel d'un système est d'augmenter sa puissance de calcul en utilisant des machines parallèles multiprocesseur. Par ailleurs, il est parfois nécessaire de décentraliser certains calculs pour les rapprocher des capteurs et des actionneurs, ce qui permet de limiter les problèmes liés au câblage. Nous appellerons par la suite, systèmes *distribués*, les systèmes qui présentent ces caractéristiques. Ces systèmes sont également soumis à des contraintes d'embarquabilité, c'est-à-dire que les ressources

matérielles mises en jeu doivent être minimisées en termes de consommation, de coût, de poids, de prix...

Un système est composé d'un ensemble de programmes informatiques et d'une machine multiprocesseur sur laquelle ils s'exécutent. Ces programmes codent des algorithmes d'application issus principalement du domaine du contrôle-commande et du traitement du signal et des images. Par la suite, nous utiliserons le terme *algorithme* qui est plus générique que celui de programme, celui-ci étant un codage particulier à l'aide d'un des langages disponibles à l'utilisateur, et nous utiliserons le terme *architecture* pour désigner la machine multiprocesseur.

Les systèmes sont d'abord spécifiés [26, 12, 39], puis vérifiés [50] et enfin optimisés [88]. La spécification d'un système consiste à décrire l'algorithme d'application, l'architecture, et finalement l'implantation de l'algorithme sur l'architecture. Lors de la spécification de l'algorithme, on décrit l'ensemble des opérations (calculs mathématiques, formatage des données...) nécessaires à la réalisation de l'algorithme, et les relations de dépendances de données entre ces opérations. Lors de la spécification de l'architecture, on décrit le type et le nombre de processeurs qui la composent, le réseau d'interconnexion, les communications, les synchronisations. Lors de la spécification de l'implantation, on décrit la distribution et l'ordonnancement non seulement des opérations mais aussi des communications inter-processeur, ceci pour éviter leur inter-blocage lors de l'exécution du système. Afin de concevoir de manière sure ces systèmes, il est intéressant d'utiliser des méthodes formelles pour les vérifier. En effet, des erreurs de conception peuvent avoir des conséquences dramatiques tant au niveau humain que financier (centrale nucléaire, Ariane 5...). Nous n'avons pas traité ici ces aspects, mais nous supposons que les spécifications d'algorithme ont été vérifiées formellement en termes d'ordre sur les événements qui entrent et sortent du système [35]. C'est pourquoi dans la suite nous montrons qu'à l'issue de l'implantation, l'ordre partiel associé à cette implantation contient l'ordre partiel initial de l'algorithme, garantissant la cohérence de l'ordre des événements de sortie relativement à l'ordre des événements d'entrée qui les déclenchent. Enfin, l'optimisation consiste à chercher une implantation respectant les contraintes temps réel et d'embarquabilité. Une mauvaise optimisation peut entraîner que les résultats produits par le système soient conformes au cahier des charges mais que les contraintes temps réel ou d'embarquabilité ne soient pas respectées. Ici nous ne traitons que le respect des contraintes temps réel.

Objectifs de la thèse

Ce travail de thèse est en continuité avec les travaux précédents menés par le projet Sosso. L'objectif de la thèse est double : nous nous intéressons d'une part à la formalisation, d'autre

part à l'optimisation des systèmes distribués temps réel, en vue de les programmer de manière efficace, sachant qu'il est primordial de respecter les contraintes temps réel du système. Pour ce faire, il faut étudier le parallélisme du système qui se définit en termes de parallélisme potentiel de l'algorithme et en termes de parallélisme disponible de l'architecture. On cherche à définir un formalisme permettant d'exploiter le parallélisme potentiel de l'algorithme pour le réduire au parallélisme disponible de l'architecture afin de construire une implantation optimisée. Si les contraintes temps réel ne peuvent pas être respectées par le système, le parallélisme de l'algorithme (redécoupage de l'algorithme), le parallélisme de l'architecture (ajout ou remplacement de ressources), doivent être remis en cause.

Dans un premier temps, il s'agit de modéliser en intention, avec un formalisme commun issu de la théorie des ensembles et des graphes, l'algorithme, l'architecture et l'ensemble des implantations *valides*, c'est-à-dire l'ensemble des implantations dont l'ordre partiel d'exécution contient l'ordre partiel du graphe initial, de l'algorithme sur l'architecture. Dans un deuxième temps, il s'agit de développer des heuristiques permettant de choisir parmi l'ensemble précédent une implantation ayant une durée d'exécution inférieure ou égale à la contrainte de latence du système. Dans la suite, nous ne parlons plus que de contrainte de latence. Nous supposons en effet que toutes les entrées du système sont disponibles en même temps au début de l'exécution de chaque itération de l'implantation, et que toutes les sorties ne sont disponibles qu'à l'issue de l'exécution de chaque itération de l'implantation. Ainsi la contrainte de latence est l'intervalle maximal de temps entre l'acquisition des entrées par les capteurs et la production des sorties par les actionneurs. Cela revient à résoudre un problème d'allocation de ressources sur une architecture multiprocesseur. Enfin, il s'agit d'implanter ces heuristiques au cœur du "logiciel SynDEx[1] de CAO niveau système pour l'implantation d'applications distribuées temps réel embarquées", qui permet de générer un exécutif temps réel dédié [34] qui supporte l'exécution d'un algorithme d'application sur une architecture multiprocesseur.

L'approche modélisation

La modélisation d'un système est une abstraction mathématique qui n'en retient que les aspects pertinents, dépendant de ce qu'on cherche à faire avec la modélisation. Dans notre cas, nous cherchons à avoir un modèle précis d'implantation conduisant à une optimisation précise en vue de la génération d'un exécutif, supportant l'exécution de cette implantation, sans inter-blocage et ayant un faible surcoût. Les modèles utilisés pour représenter les systèmes peuvent être de type différent [89, 26] (algèbre, automates, statecharts [37], réseaux de Pétri, systèmes basés sur la logique, approches orientées programmation telles ESTEREL,

1. www-rocq.inria.fr/syndex

LOTOS, SIGNAL [7, 59]...). Cependant on peut considérer qu'ils sont tous fondés sur des systèmes de transition [4] ou encore des machines à états finies, constitués d'un ensemble d'états qui représentent les configurations du système et d'un ensemble de transitions qui indiquent comment le système passe d'une configuration à une autre. Les modèles proposés sont aussi de ce type.

Nous avons cherché à modéliser l'algorithme d'application en faisant le plus possible abstraction de l'architecture sur laquelle l'algorithme va être implanté, ceci dans un souci de portabilité et de réutilisabilité. En effet, si l'on construit un programme parallèle destiné à être implanté sur une machine composée de trois processeurs et qu'on se rend compte que finalement l'exécution de ce programme ne respecte pas la contrainte temps réel, il faudra essayer avec quatre processeurs ou plus. Le programme devra être entièrement revu, il faudra de nouveau aussi programmer le schéma de communications inter-processeur. Nous avons modélisé l'algorithme par un hypergraphe orienté infiniment itéré et conditionné, encore appelé graphe flot de données conditionné lorsqu'il est factorisé. En effet, un système réactif réalise une infinité de fois la séquence Acquisition-des-entrées-Calculs-Productions-des-sorties (itérations infinies), et ce ne sont pas toujours les mêmes calculs (conditionnement) qui sont exécutés suivant les booléens calculés dans le graphe de l'algorithme ou venant de l'extérieur. De plus, l'état du système est mémorisé par des sommets spéciaux insérés dans le graphe. L'ordre partiel associé à ce graphe définit le parallélisme potentiel de l'algorithme.

Nous avons modélisé l'architecture par un hypergraphe non orienté où les sommets sont les automates séquentiels de l'architecture de type calcul ou communication et les arêtes sont des liaisons entre ces automates. Ce modèle permet de décrire des architectures multiprocesseur, et est suffisamment fin pour prendre en compte les caractéristiques importantes de ces architectures concernant principalement les communications inter-processeur. Ce modèle permet d'exhiber le parallélisme disponible dont l'architecture dispose.

Enfin, nous avons proposé un modèle relationnel d'implantations de l'algorithme sur l'architecture. Ce modèle est le résultat de la composition de trois relations : le routage, la distribution et l'ordonnancement qui décrit l'ensemble des implantations valides d'un algorithme donné sur une architecture donnée. Chaque implantation de cet ensemble est également un hypergraphe orienté conditionné, où la distribution et l'ordonnancement à la fois des calculs et des communications sont effectués.

L'approche optimisation

L'optimisation consiste à choisir, parmi l'ensemble des implantations valides, une implantation de durée d'exécution minimale. La durée d'exécution d'une implantation correspond

en termes de graphes à la longueur du chemin critique. Nous cherchons à minimiser la longueur du chemin critique de l'implantation en exploitant la marge d'ordonnancement, c'est-à-dire la différence entre les dates de début au-plus-tard et de début au-plus-tôt d'exécution relativement aux dépendances de données, de chaque opération. Ce problème de placement et d'ordonnancement est un problème de recherche reconnu comme un problème NP-difficile [27] dans les cas réalistes (plus de deux processeurs, structure de graphes quelconques, opérations de durée non unitaire...) ; nous devons donc utiliser des méthodes approchées pour trouver une solution. Deux types de méthodes statiques sont particulièrement étudiées ici : les heuristiques gloutonnes et les méthodes de voisinage. Les heuristiques gloutonnes, souvent implantées sous forme d'algorithme de type liste [88], réalisent, à chaque étape de l'algorithme d'ordonnancement, des choix d'implantation partiels et que l'on ne remet pas en cause par la suite. Elles construisent ainsi une solution rapidement, et si le critère est bien choisi, la solution est de bonne qualité. Les méthodes de voisinage partent d'une implantation solution (obtenue éventuellement par une heuristique gloutonne) et cherchent à améliorer cette solution en cherchant dans un voisinage une solution meilleure. Nous avons défini deux méthodes, l'une gloutonne et l'autre de voisinage, qui construisent toutes deux une implantation optimisée. Ces deux méthodes ont été implantées dans le logiciel SynDEx de CAO niveau système qui supporte la méthodologie AAA.

Structure de la thèse

La première partie de la thèse, appelée "modélisation relationnelle", présente les trois modèles d'algorithme, d'architecture et d'implantation qui définissent un système temps réel distribué embarqué. Dans le premier chapitre, nous définissons le modèle d'architecture. Après un état de l'art sur les architectures parallèles, nous nous plaçons dans le contexte et définissons deux modèles d'architecture, le premier dit *encapsulé* et le deuxième dit *développé*. Le modèle encapsulé est un hypergraphe non orienté où les sommets sont les processeurs et les arêtes sont les liaisons inter-processeur. Ce modèle identifie les processeurs à leur unité de calcul, et permet ainsi la distribution et l'ordonnancement des opérations de calcul sur les processeurs ; cependant comme il ne permet qu'une distribution sur les liaisons il est insuffisant. La granularité du modèle développé est plus fine que celle du modèle encapsulé. Les sommets de l'hypergraphe non orienté sont les unités de calcul et les unités de communication des processeurs, et les arêtes sont les liaisons inter-unité. Ce modèle prend en compte la capacité finie des liaisons inter-processeur et permet non seulement une distribution et un ordonnancement des calculs sur les unités de calcul, mais aussi une distribution et un ordonnancement des communications sur les liaisons inter-processeur. Dans le deuxième

chapitre, nous définissons le modèle d'algorithme. C'est un hypergraphe orienté infiniment itéré et conditionné. Le conditionnement permet de prendre en compte le fait qu'en fonction de booléens, calculés dans le graphe ou venant de l'extérieur, certaines parties du graphe peuvent ne pas être exécutées. Le conditionnement est modélisé par des dépendances de conditionnement (différentes des dépendances de données), appelées conditions d'activation comme celles utilisées dans le format commun DC des langages synchrones [78]. Ces dépendances définissent une relation d'exclusivité entre opérations qui sera exploitée par les heuristiques d'optimisation définies dans la deuxième partie de la thèse. Dans le troisième chapitre, nous définissons le modèle d'implantation. Plus précisément, on construit, en intention, l'ensemble des implantations valides d'un algorithme sur une architecture, en composant trois relations sur des graphes de type différent : le routage, la distribution et l'ordonnancement. Le routage permet de rendre complètement connectée une architecture, qui ne l'est pas nécessairement, en construisant en intention toutes les routes (chemins) possibles du graphe la décrivant. La distribution réalise deux partitions successives du graphe de l'algorithme. Les sommets opérations sont d'abord distribués sur les unités de calcul. Ensuite, les dépendances entre opérations distribuées sur des unités de calcul différentes, sont distribuées sur les routes du graphe de l'architecture construites précédemment. L'ordonnancement consiste à rendre total l'ordre partiel d'exécution associé à chaque sous-graphe alloué à chaque unité, car celle-ci est une machine à états finie séquentielle. En composant ces trois relations, on définit l'ensemble des implantations valides.

Dans la seconde partie de la thèse, appelée "optimisation et heuristique", nous cherchons une implantation valide, parmi celles construites dans le troisième chapitre de la première partie dont la longueur du chemin critique est minimale. Le premier chapitre concerne la caractérisation d'une implantation : à chaque sommet du graphe de l'algorithme implanté, on associe une durée d'exécution, ce qui nous permettra de calculer la longueur du chemin critique. Cette durée d'exécution est calculée en exploitant les caractéristiques de l'architecture. Le deuxième chapitre est un état de l'art des différentes méthodes de résolution des problèmes de placement et d'ordonnancement dans les systèmes multiprocesseurs, puis nous nous plaçons dans ce contexte en indiquant quelles méthodes de résolution nous allons choisir. Le troisième chapitre décrit deux méthodes d'optimisation de l'implantation d'un graphe d'algorithme sur un graphe d'architecture. La première heuristique est une heuristique gloutonne qui construit une implantation valide, prenant en compte les opérations conditionnées et la deuxième heuristique est une méthode approchée de voisinage de recherche locale avec retour-arrière. Elle part d'une implantation initiale construite avec l'heuristique gloutonne et cherche à améliorer cette implantation en cherchant dans son voisinage une solution meilleure. Afin d'évaluer la qualité de notre heuristique gloutonne, nous l'avons comparée

sur quelques exemples d'applications à d'autres heuristiques du même type décrites dans la littérature.

La troisième partie de la thèse présente les développements logiciels effectués. Ces développements concernent le modèle d'algorithme et les heuristiques et sont incorporés au cœur du logiciel SynDEx. Le premier chapitre décrit les modifications apportées à l'interface homme machine de SynDEx écrite en Tcl/Tk pour qu'elle puisse supporter des graphes d'algorithmes conditionnés. Nous avons intégré à cette interface les dépendances de conditionnement et avons modifié la base de données correspondante écrite en C++. Le deuxième chapitre décrit le codage de l'ordonnancement conditionné dans le cœur de SynDEx, puis détaille la spécification des deux algorithmes d'ordonnancement correspondant aux heuristiques gloutonne et de voisinage présentées dans la deuxième partie de la thèse. Enfin, les annexes donnent quelques programmes en Tcl/Tk et C++ correspondant aux algorithmes décrits dans les chapitres précédents.

Première partie

Modélisation relationnelle

1. Modèle d'architecture

1.1 État de l'art

1.1.1 Introduction

La complexité des algorithmes ne cessant d'augmenter, l'utilisation de machines *multiprocesseur distribuées* est nécessaire pour minimiser les temps d'exécution. Ce type d'architecture permet, d'une part, d'augmenter la puissance de calcul par rapport à un monoprocesseur grâce au parallélisme *disponible* de la machine et, d'autre part de prendre en compte la délocalisation de certaines fonctionnalités pour se rapprocher des capteurs et des actionneurs et ainsi limiter le câblage, source de détérioration de signaux analogiques. De plus, les contraintes temps réel et d'embarquabilité sont telles qu'il est parfois nécessaire d'utiliser en plus des processeurs, des circuits intégrés spécialisés (ASIC[1] figés ou FPGA[2] configurables). Ainsi, les architectures distribuées regroupent les architectures multiprocesseur et les circuits intégrés.

Remarque 1 *Dans la suite, on ne s'intéressera qu'aux processeurs et pas aux circuits intégrés, c'est pourquoi on ne parlera que de multiprocesseur.*

Définition 1 *Un processeur est composé d'un CPU*[3] *ou* unité de calcul*, et d'une ou plusieurs* unités de communication *(pour communiquer avec les périphériques ou d'autres processeurs).*

Définition 2 *Le CPU est une machine séquentielle à flot de contrôle de type Von Neumann. En général, il est constitué d'une unité de contrôle, encore appelée séquenceur d'instructions (SI), d'une unité de traitement (UT) et d'une mémoire. Par la suite, nous l'identifierons uniquement à son séquenceur. Toutes les instructions sont traitées de manière séquentielle, il n'y a pas de parallélisme entre elles, l'ordre d'exécution des instructions effectuées par le CPU est donc total.*

Par contre, un processeur peut permettre du parallélisme entre les calculs exécutés par le CPU et les communications *inter-processeur*. Le parallélisme *disponible* d'une architecture

1. Application Specific Integrated Circuit
2. Field Programmable Gate Array
3. Central Processing Unit

multiprocesseur existe donc à la fois au niveau intra-processeur entre instructions et communications, et aussi au niveau inter-processeur lorsqu'il s'agit d'échanger des informations entre les processeurs. Les performances d'une telle architecture dépendent de la capacité du programmeur à exploiter le parallélisme de la machine.

Plus précisément, ces performances dépendent non seulement du nombre de processeurs utilisés, de la structure de chacun d'entre eux, mais aussi et surtout, de la manière dont ils communiquent. Les communications inter-processeur sont critiques dans la conception de système multiprocesseur, car si elles ne sont pas effectuées de façon efficace, les performances de la machine multiprocesseur sont dégradées.

Plusieurs classifications sont proposées dans la littérature pour tenter de définir les différents types de machines multiprocesseur.

Nous présentons brièvement trois de ces classifications qui permettent de préciser le contexte de l'étude. Puis, nous définissons deux modèles de graphes non orientés bien adaptés à la modélisation de machines multiprocesseur : tout d'abord un modèle dit *modèle encapsulé*, puis un modèle plus précis dit *modèle développé*. Ces deux modèles sont comparés aux modèles PRAM –Parallel Random Access Machine– [29, 57, 61, 79] et DRAM –Distributed Random Access Machine– [18, 19] de la littérature. Le modèle *encapsulé* correspond à une approche classique où chaque sommet est un processeur et chaque arête est une liaison physique de communication qui permet des transferts de données entre les mémoires des processeurs, au besoin par l'intermédiaire d'une mémoire commune. Le modèle *développé* est plus précis, la granularité de ce modèle n'est plus le processeur mais *l'unité*. Nous allons détailler ces modèles plus loin.

1.1.2 Classification des architectures parallèles

Suivant le type de parallélisme

Flynn a proposé dans [25] en 1972, une classification des architectures suivant le type de parallélisme de la machine : parallélisme de données, parallélisme d'instructions ou les deux à la fois.

Flynn distingue quatre catégories d'architectures :

- *SISD -Single Instruction Single Data*
  Ce genre d'architecture correspond à une architecture de type Von Neumann c'est-à-dire une machine purement séquentielle ne disposant pas de parallélisme. C'est donc une machine monoprocesseur.
- *SIMD -Single Instruction Multiple Data*
  Il y a un seul séquenceur d'instruction et plusieurs unités arithmétiques et logiques.

La même instruction opère sur des données différentes. C'est une machine disposant de parallélisme de données.

Remarque 2 *Nous ne modélisons pas ce type de machine. En effet, nous verrons plus loin que notre modèle ne prend en compte pour l'instant que le séquenceur du CPU. En effet, ce qui nous intéresse ici est d'affecter des parties de programmes à des séquenceurs d'instructions, nous ne nous intéressons ni à la mémoire de programme, ni à celle de données.*

- *MISD -Multiple Instructions Single Data*
  Il y a plusieurs séquenceurs d'instructions et plusieurs unités arithmétiques et logiques. Plusieurs instructions opèrent sur la même donnée. Ces machines possèdent du parallélisme d'instructions.
- *MIMD -Multiple Instructions Multiple Data*
  Il y a plusieurs séquenceurs d'instructions et plusieurs unités arithmétiques et logiques. Plusieurs instructions opèrent sur des données différentes. Ces machines possèdent à la fois du parallélisme d'instructions et à la fois du parallélisme de données.

  Remarque 3 *Les architectures* SPMD -Single Program Multiple Data- *sont des architectures MIMD dans lesquelles tous les programmes sont identiques sur tous les processeurs, mais s'appliquent sur des données différentes.*

Les modèles que nous allons présenter permettent de décrire des architectures monoprocesseur SISD et multiprocesseur MISD, MIMD et SPMD.

Suivant les types de communications

La classification de Flynn ne tient pas compte des différents types de communication entre processeurs.

Les communications des machines MIMD peuvent être classifiées en deux sous-catégories : on distingue les communications par *passage de messages* et les communications par *partage de données*. Les deux types de communications se font par l'intermédiaire d'un réseau d'interconnexion et de mémoires.

On distingue trois sortes de liaisons composant le réseau d'interconnexion (cf. table 1.1.1) :

- liaison point à point SAM[4] : c'est une liaison bidirectionnelle entre deux mémoires SAM de processeurs différents, elle est appelée *lien* dans la modélisation définie à la page 22. Ce type de liaison ne supporte que des communications par passage de messages.

4. Sequential Access Memory. L'accès à ce type de liaison est FIFO - First In First Out-. Ainsi, l'ordre d'émission des messages est aussi l'ordre de réception.

		Différentes liaisons	Modélisation
Réseau d'interconnexion SAM-RAM	Réseau d'interconnexion SAM	Liaisons point à point SAM	Lien
		Liaisons multipoints SAM	Bus SAM
	Réseau d'interconnexion RAM	Liaisons multipoints RAM	Bus RAM

TAB. 1.1.1 – *Composition des réseaux d'interconnexion*

- liaison multipoint SAM : c'est une liaison multidirectionnelle reliant les mémoires SAM de plus de deux processeurs, elle est appelée *bus SAM* dans la modélisation. Ce type de liaison ne supporte que les communications par passage de messages.
- liaison multipoint RAM[5] que l'on appellera par la suite *bus RAM* : c'est une liaison multidirectionnelle disposant d'une mémoire RAM. Chaque communication supportée par ce type de liaison est une communication par partage de données.

Nous allons maintenant définir les notions de communication par partage de données et par passage de messages.

Définition 3 Communication par partages de données
Les processeurs communiquent au travers d'une mémoire globale que l'ensemble des processeurs se partagent. Le problème que pose ce type de mémoire est celui de la contention. *En effet, bien que le bus d'accès à cette mémoire dispose d'une bande passante très large, il peut y avoir des conflits si les processeurs veulent tous accéder au bus pour atteindre la mémoire ou s'ils veulent accéder à la même case mémoire. Ceci correspond à deux types de ressource partagée. Un* arbitre *est nécessaire pour gérer ces conflits.*

Les réseaux de processeurs communiquant par partage de données au travers d'une mémoire, souvent qualifiée de *mémoire partagée*, sont modélisés par des PRAMs -Parallel Random Access Machine- [29, 57, 61, 79]. Cette modélisation est insuffisante dans la mesure où elle ne modélise pas l'arbitrage et où elle suppose une bande passante infinie. Pour ce

5. Random Access Memory. L'accès à ce type de liaison est en général CREW, c'est-à-dire lecture concurrente (CR) et écriture exclusive (EW). Autrement dit, l'écriture d'une donnée précède sa(ses) lecture(s).

modèle, le coût des communications est nul et donc la durée d'exécution d'une application ne dépend que de la durée des différentes opérations qui la composent.

Remarque 4 *Il existe des modèles plus précis pour modéliser les types d'accès en lecture et en écriture. Ainsi, on distingue, les PRAMs à lecture concurrente (CR), lecture exclusive (ER), à écriture concurrente (CW) ou à écriture exclusive (EW). Le modèle le plus courant est d'accéder à la mémoire en lecture simultanée et en écriture exclusive (CREW) et c'est le modèle que l'on a choisi pour notre modélisation.*

Définition 4 Communication par passage de messages
Pour remédier au problème de contention lié au partage du bus, chaque processeur possède sa propre mémoire à laquelle il peut accéder très rapidement et surtout sans conflit, et communique par passage de messages avec les autres processeurs.

Les réseaux de processeurs communiquant par passage de messages sont modélisés par des DRAMs –Distributed Random Access Machine–. Ce sont des RAMs –Random Access Machine– qui communiquent par des liaisons SAM.

Définition 5 *On appelle réseau d'interconnexion SAM (resp. RAM), un réseau composé uniquement de liaisons SAM (resp. RAM) et on appelle réseau d'interconnexion SAM-RAM un réseau composé des deux types de liaisons.*

La table 1.1.2 décrit l'équivalence entre les supports de communication et les types de communication.

Support de la communication		Types de communication
Réseau d'interconnexion SAM	⇔	Par passage de messages
Réseau d'interconnexion RAM	⇔	Par partage de données

TAB. 1.1.2 – *Equivalence entre supports et types de communication*

Dans cette thèse, nous cherchons à modéliser des processeurs pouvant communiquer en utilisant un réseau d'interconnexion SAM (ce qui correspond aux zones grisées des tables

1.1.1 et 1.1.2), que l'on appellera par la suite simplement réseau d'interconnexion ou encore réseau.

Remarque 5 *Dans les communications par passage de messages, le message peut soit être envoyé dans sa totalité (c'est le mode que l'on a choisi), soit être envoyé par paquets de taille fixe. Le premier mode d'envoi présuppose que l'ensemble du chemin permettant d'aller de l'émetteur du message au récepteur soit libre au moment de l'envoi. Ainsi, on verra que la durée d'une communication est égale à la durée d'attente, due à l'éventuelle indisponibilité du média, plus la durée effective du transfert du message sur le réseau. Dans le second mode, le chemin doit être libre "par morceaux" et l'attente peut se faire à des points de connexion du réseau.*

Nous allons maintenant voir une classification des architectures MIMD communiquant par passage de messages suivant le réseau d'interconnexion.

Suivant le réseau d'interconnexion

Le réseau d'interconnexion d'une architecture MIMD communiquant par passage de messages peut être *statique* ou *dynamique*.

Définition 6 Réseau statique *Un réseau est statique s'il n'évolue pas au cours de l'application.*

Définition 7 Réseau dynamique *A l'inverse, un réseau est dynamique si sa structure n'est pas constante au cours du déroulement de l'application.*

Associées aux réseaux, il existe des topologies classiques de connexion des processeurs [24, 19], les plus connues sont les connexions en anneau, tore, hypercube...

1.1.3 Contexte de la modélisation

Nous cherchons à modéliser des architectures de type MISD, MIMD ou SPMD. Le réseau d'interconnexion de ces architectures est composé de deux types de liaisons, d'une part les liaisons point à point SAM et d'autre part les liaisons multipoints SAM.

Remarque 6 *Dans cette modélisation, on ne prend en compte que la mémoire nécessaire pour les communications qui est contenue implicitement dans les média comme on va le voir par la suite. On ne prend en compte ni la mémoire de données ni la mémoire de programmes.*

De plus, nous modélisons des réseaux uniquement statiques qui sont connexes, mais pas nécessairement complètement connectés. Ainsi, nous modélisons le routage des communications (cf. chapitre 3). De plus, on effectue également l'ordonnancement des communications sur les

supports de communication. Les processeurs composant ces architectures peuvent être tous identiques, on parle alors d'une *architecture homogène*, ou bien posséder des caractéristiques différentes, l'architecture est dans ce cas qualifiée d'*hétérogène*.

Une fois cette modélisation ainsi que celle de l'algorithme effectuées, nous allons décrire dans la section 3.3.5 page 131, l'ensemble des implantations possibles d'un algorithme sur une architecture.

1.2 Modèle d'architecture *encapsulé*

1.2.1 Introduction

Définition 8 *Un hypergraphe non orienté est un couple (S,E) où :*

$S = \{s_1,s_2,...,s_n\}$ *est l'ensemble des sommets de l'hypergraphe.*

$E = \{e_1,e_2,...,e_m\}$ *est l'ensemble des arêtes du graphe. E est une famille de parties de S vérifiant :*

$$\begin{cases} e_i \neq \emptyset \quad 1 \leq i \leq m \\ \bigcup_1^m e_i = S \end{cases}$$

Le modèle encapsulé est un hypergraphe (P,H), où les sommets P de cet hypergraphe désignent les processeurs composant l'architecture et les arêtes H désignent les liaisons physiques entre processeurs. Ce modèle correspond à l'architecture *hôte* définie dans [14].

Remarque 7 *Ce modèle identifie les processeurs à leur CPU, que l'on appellera par la suite,* unité de calcul.

Remarque 8 *Ce modèle permet de décrire le type de réseau d'interconnexion qui relie les différents processeurs qui le composent.*

Remarque 9 *Ce modèle est utilisé dans beaucoup d'heuristiques comme modèle d'architecture. On verra par la suite qu'il n'est pas suffisamment fin pour décrire l'ordonnancement des communications. Étant donné que dans notre cas, l'ordonnancement est suivi de la génération d'exécutif distribué et temps réel, nous devons modéliser finement l'ordonnancement des communications. C'est pourquoi nous n'allons pas retenir ce modèle mais nous le présentons car il permet une meilleure compréhension du modèle d'architecture suivant.*

Remarque 10 *Par la suite, pour alléger le discours on parlera de graphe plutôt que d'hypergraphe.*

1.2.2 Formalisation

Soit G_{ar}, le graphe de l'architecture encapsulé. G_{ar} est un couple (P,H) où :

- P est l'ensemble des processeurs, appelés *nœuds* processeurs, Card $P = n_P$, $P = \{p_i\}_{1 \leq i \leq n_P}$.

  Remarque 11 *Chaque sommet est une machine à états finie séquentielle.*

- $H \subseteq \mathcal{P}(P)$ est l'ensemble des hyperarcs, appelés *liaisons physiques*, Card $H = n_H$.
  $h_i = \{p_{i_1},...,p_{i_{k(i)}}\}_{2 \leq k(i) \leq n_P}$, soit $h_i = \{p_{i_j}\}_{1 \leq j \leq k(i)\,,\, 2 \leq k(i) \leq n_P}$
  Les liaisons physiques H peuvent être de deux types :
  - *point à point* : ce sont des liaisons SAM physiques de communication bidirectionnelles inter-processeur que nous appellerons *liens*. On désigne par L cet ensemble, Card $L = |L|$, $L \subseteq H$
    $L = \{l_i\}_{1 \leq i \leq |L|}$ et $l_i = \{p_{i_j}\}_{1 \leq j \leq 2}$
    Chaque lien est donc modélisé par un couple de processeurs.
  - *multipoints* : ce sont des bus *SAM*[6] *–Sequential Access Memory–* de communication que nous appellerons *bus*. On désigne par B cet ensemble.
    $B \subseteq H$, Card $B = |B|$.
    $B = \{b_i\}_{1 \leq i \leq |B|}$ et $b_i = \{p_{i_j}\}_{1 \leq j \leq k(i)\,,\, 3 \leq k(i) \leq n_P}$
    Chaque bus est modélisé par au moins un triplet de processeurs.

Relation associée au modèle encapsulé

Définition 9 *Associé au modèle encapsulé, on définit la relation $\mathcal{R}$: "est en communication directe avec", notée $p_i \,\mathcal{R}\, p_j$.*
p_i est en communication directe avec p_j si et seulement si : $\exists h_k \in H \quad / \quad \{p_i,p_j\} \subseteq h_k$.

Propriété 1 *$\mathcal{R}$ est une relation symétrique par construction.*

On définit, la fonction ρ qui, à chaque processeur p_i associe l'ensemble des hyperarcs qui contiennent p_i.

$$\boxed{\begin{array}{llll} \rho: & P & \rightarrow & H \\ & p_i & \mapsto & \rho(p_i) = \{h_j \in H \quad tq \quad p_i \in h_j\} \end{array}}$$

On définit, $|\rho(p_i)|$, le nombre d'hyperarcs de G_{ar} qui contiennent p_i.

6. Ceci pour les différencier de ce que l'on appelle généralement bus, c'est-à-dire le bus lui-même plus une mémoire RAM.

Avantages et inconvénients de ce modèle

Ce modèle est équivalent au modèle PRAM [18] communiquant par mémoire partagée pour lesquels les temps de communication sont nuls. C'est donc un modèle très simplifié de la réalité qui ne modélise ni les coûts de communication ni les arbitrages, c'est pourquoi nous ne l'avons pas retenu.

1.2.3 Représentation graphique

Un lien est représenté par un trait entre les 2 opérateurs . Un bus est une boucle fermée contenant les opérateurs qui communiquent grâce à ce bus (cf. figure 1.2.1).

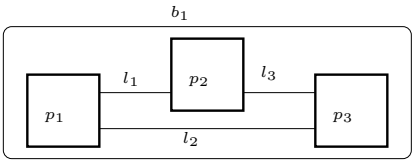


FIG. 1.2.1 – *Graphe de l'architecture encapsulé*

$P = \{p_1,p_2,p_3\}$
$H = \{L,B\}$ avec : $L = \{l_1,l_2,l_3\}$ et $B = \{b_1\}$
$l_1 = \{p_1,p_2\}$, $l_2 = \{p_1,p_3\}$, $l_3 = \{p_2,p_3\}$, $b_1 = \{p_1,p_2,p_3\}$

1.3 Modèle d'architecture *développé*

1.3.1 Introduction

La granularité du modèle développé [3] est plus fine que celle du modèle encapsulé. Un processeur n'est plus identifié à son unité de calcul. Il est composé d'une unité de calcul et d'au moins une unité de communication. Les unités d'un même processeur sont connectées entre elles par un bus RAM dit *intra-processeur*. Ainsi, l'ensemble des sommets du graphe développé de l'architecture est un ensemble d'unités de calcul et d'unités de communication et l'ensemble des hyperarcs est l'ensemble des liaisons entre ces unités. Les unités de calcul de ce graphe ne sont pas directement connectées entre elles. Pour communiquer, elles utilisent les unités de communication, qui elles, forment un réseau connexe. Les liaisons qui connectent les unités de communication de processeurs différents correspondent aux liaisons H précédemment définies et sont qualifiées d'*inter-processeur*. Chaque liaison inter-processeur est donc un ensemble d'unités de communication. Une liaison inter-processeur et ses unités de

communication associées forment ce que l'on appelle un *média de communication* encore appelé média.

Un transfert de données entre deux unités de calcul emprunte un bus intra-processeur auquel est connectée l'unité de calcul émettrice, un ensemble de média et enfin un autre bus intra-processeur auquel est connecté l'unité de calcul réceptrice.

Ce nouveau modèle, comme on le verra dans le chapitre 3 modèle d'implantation, permet d'ordonnancer à la fois les opérations sur les unités de calcul, et les transferts de données sur les média.

1.3.2 Formalisation

Soit G_{ar}, le graphe de l'architecture développé. G_{ar} est un couple (S,H_d) où :

- S est l'ensemble des sommets du graphe appelés *opérateurs*. Card $S = |S|$. Ces opérateurs sont de deux types :
  - *les opérateurs de calcul* : ce sont les unités de calcul des processeurs. On désigne par S_{cal} cet ensemble.
    $S_{cal} = \bigcup_{p \in P} S_{p,cal}$ où $S_{p,cal}$ désigne l'unité de calcul du processeur p.
    Remarque 12 *Card $S_{cal} = n_P$ puisqu'il y a une et une seule unité de calcul par processeur.*

    Remarque 13 *$S_{p,cal}$ est une machine séquentielle.*
  - *les opérateurs de communication* : ce sont les unités de communication des processeurs. On désigne par S_{com} cet ensemble.
    $S_{com} = \bigcup_{p \in P} (\bigcup_j S_{p,com^j})$ où $\bigcup_j S_{p,com^j}$ désigne l'ensemble des unités de communication du processeur p, noté $S_{p,com}$.

  Ainsi, $S = \bigcup_{p \in P} (S_{p,cal} \cup S_{p,com}) = S_{cal} \cup S_{com}$.
- H_d est l'ensemble des hyperarcs. H_d est constitué de deux sous-ensembles :
  - H est l'ensemble des hyperarcs constitué de deux sous-ensembles L et B. Ces trois ensembles correspondent aux trois ensembles H,L et B définis dans le modèle encapsulé. Cette fois-ci sur le modèle développé, ce sont des ensembles d'unités de communication au lieu d'être des ensembles de processeurs. Ils sont qualifiés *d'inter-processeur*.
    $H \subset \mathcal{P}(S_{com})$, Card $H = n_H$.
    $H = L \cup B$ et tout élément de l'ensemble L est de dimension 2 et tout élément de l'ensemble B est de dimension strictement supérieure à 2.

– C est l'ensemble des hyperarcs inter-opérateur qui appartiennent au même noeud processeur, ils sont qualifiés d'*intra-processeur*.
$C = \bigcup_{p \in P} c_p$ et $c_p = S_{p,cal} \cup S_{p,com}$
c_p est l'hyperarc reliant les opérateurs du processeur p. Chaque hyperarc c_p modélise un bus RAM qui permet à l'unité de calcul $S_{p,cal}$ de communiquer par mémoire partagée avec les unités $S_{p,com}$ associées.

Ainsi, chaque *nœud* processeur p du modèle encapsulé est cette fois-ci vu comme un graphe dont l'ensemble des sommets est composé d'une unité de calcul $S_{p,cal}$ et d'un ensemble d'unités de communication $(\bigcup_j S_{p,com^j})$ et dont l'ensemble des hyperarcs est réduit à l'hyperarc c_p.

Définition 10 *On appelle média m_j du graphe de l'architecture, l'ensemble composé de l'hyperarc h_j (de type lien ou bus) et de ses opérateurs de communication associés. M désigne l'ensemble des média qui composent le graphe de l'architecture.*

Remarque 14 *Le graphe* développé *sera utilisé lors de la formalisation de l'implantation pour modéliser les communications inter-processeur et plus précisément leur ordonnancement.*

Remarque 15 *Les hyperarcs H sont cette fois-ci des ensembles d'unités de communication, par conséquent les unités de calcul devront utiliser les unités de communication pour communiquer.*

Relation associée au modèle développé

Définition 11 *Associé au modèle développé, on définit la relation $\mathcal{R}$: "est en communication directe avec", notée $S_{p,cal}$ $\mathcal{R}$ $S_{q,cal}$. $S_{p,cal}$ est en communication directe avec $S_{q,cal}$ si et seulement si : $\exists h_k \in H \quad / \quad \{S_{p,com^q}, S_{q,com^p}\} \subseteq h_k$.*

Propriété 2 *$\mathcal{R}$ est une relation symétrique par construction.*

1.3.3 Représentation graphique

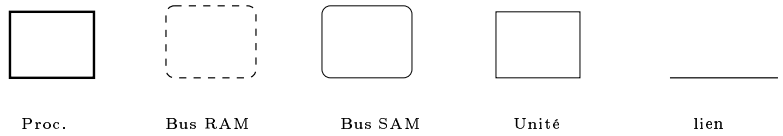


FIG. 1.3.2 – *Sémantique graphique du graphe de l'architecture développé*

Les boîtes dessinées avec un trait continu épais désignent les sommets du modèle encapsulé. Ces boîtes sont également dessinées sur le modèle développé afin de pouvoir passer d'un modèle à un autre. Les sommets du graphe développé sont dessinés en trait continu. Les boîtes qui encapsulent des boîtes, dessinées en trait continu, sont les hyperarcs du modèle développé. Ils sont de deux types : bus SAM entre unités de communication de processeurs différents (représenté en trait continu) et bus RAM entre l'unité de calcul et ses unités de communication associées (représenté en trait pointillé). Enfin, les traits continus reliant deux unités de communication désignent des liaisons point à point (cf. fig. 1.3.2).

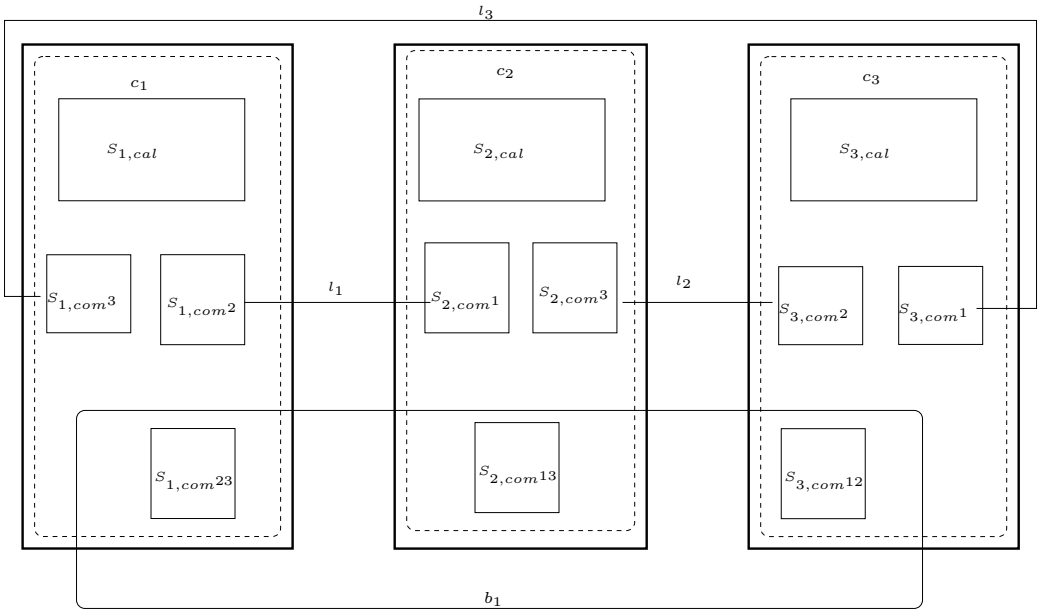


FIG. 1.3.3 – *Graphe de l'architecture développé*

Légende associée au modèle développé (cf. fig. 1.3.3) :

Unités : $S = S_{cal} \cup S_{com}$

Unités de calcul : $S_{cal} = \{S_{1,cal}, S_{2,cal}, S_{3,cal}\}$

Unités de communication : $S_{com} = \{S_{1,com^3}, S_{1,com^2}, S_{1,com^{23}}, S_{2,com^1}, S_{2,com^3}, S_{2,com^{13}}, S_{3,com^2}, S_{3,com^1}, S_{3,com^{12}}\}$

Hyperarcs : $H_d = H \cup C$ avec : $C = \{c_1, c_2, c_3\}$

$c_1 = \{S_{1,cal}, S_{1,com^3}, S_{1,com^2}, S_{1,com^{23}}\}$, $c_2 = \{S_{2,cal}, S_{2,com^1}, S_{2,com^3}, S_{2,com^{13}}\}$,

$c_3 = \{S_{3,cal}, S_{3,com^1}, S_{3,com^2}, S_{3,com^{12}}\}$

Hyperarcs inter-processeur : $H = L \cup B$, avec : $L = \{l_1, l_2, l_3\}$ et $B = \{b_1\}$

$l_1 = \{S_{1,com^2}, S_{2,com^1}\}$, $l_2 = \{S_{2,com^3}, S_{3,com^2}\}$, $l_3 = \{S_{1,com^3}, S_{3,com^1}\}$

$b_1 = \{S_{1,com^{23}}, S_{2,com^{13}}, S_{3,com^{12}}\}$.

1.4 Exemples d'architecture

Nous présentons deux exemples d'architecture, une architecture homogène utilisée pour une application de traitement d'images [3] et une architecture hétérogène utilisée pour une application de contrôle commande [45].

1.4.1 Architecture homogène

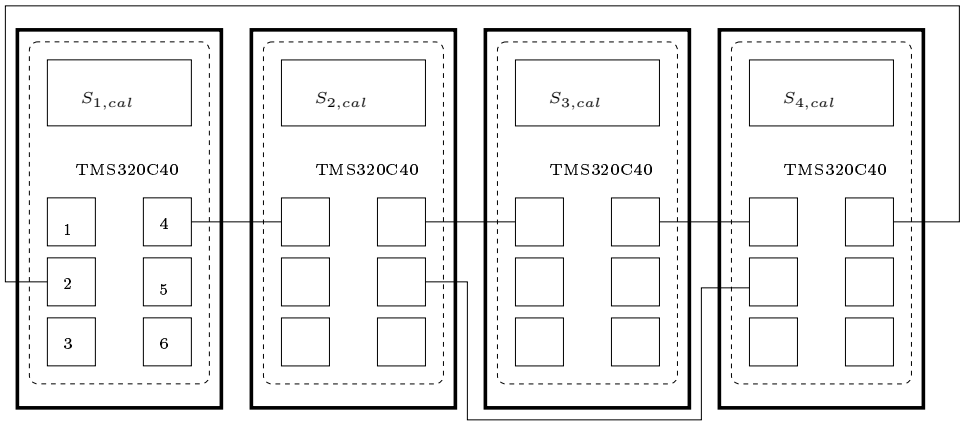


FIG. 1.4.4 – *Exemple d'architecture homogène composée de 4 TMS320C40*

L'architecture homogène, que nous considérons ici, est composée de quatre processeurs TMS320C40, communément appelés C40. Le C40, proposé par Texas Instruments [80], est un processeur de traitement du signal (DSP -Digital Signal Processing-). Il comporte un CPU, six ports de communication bidirectionnels à grands débits (5-M'egamots par seconde pour un temps de cycle processeur de 40 MHz) et six canaux de DMA -Direct Memory Access-. Les canaux de DMA permettent des transferts de données rapides via les ports de communication sans aucune intervention du CPU. Ainsi, les calculs et les communications peuvent être effectués en parallèle. Le C40 possède également une mémoire que le CPU et les six canaux de DMA se partagent. Les ports de communication sont connectés entre eux par un seul type de liaison, les liaisons point à point. Chaque port de communication contient les composants suivants :

- un tampon d'entrées FIFO et un tampon de sorties FIFO de 8 niveaux de 32 bits de large ;
- une unité d'arbitrage du port (PAU) qui gère l'arbitrage associé au déplacement de données entre un TMS320C40 et un élément externe au travers du bus de données du port de communication ;

– un registre de contrôle du port de communication (CPCR) qui autorise le contrôle des fonctions du port de communication et des opérations de transferts de données entre un TMS320C40 et un élément externe via le bus de données du port de communication.

Une modélisation de cette architecture à base de TMS320C40 est donnée figure 1.4.4. Dans cette représentation, le DMA est inclus dans la mémoire interne représentée ici par le bus RAM. Chaque processeur est donc composé d'une unité de calcul $S_{p,cal}$ et de 6 unités de communication numérotées de 1 à 6. Les processeurs sont reliés entre eux par des liaisons point à point.

1.4.2 Architecture hétérogène

Nous présentons l'architecture du prototype de véhicule semi-autonome réalisé dans le projet LARA de l'INRIA. Sur ce véhicule, chacune des roues ainsi que son frein sont commandés par un micro-contrôleur MC68332. Ce véhicule possède également un micro-contrôleur MC68332 pour commander la direction du véhicule à l'aide d'un joystick, un PC 486 pour gérer une interface homme machine, et une caméra reliée à un DSP i80196 [45]. Chaque MC68332 possède une unité de communication qui est une interface CAN (IT-CAN). Le PC dispose d'une unité de communication pour la liaison point à point RS232 avec le DSP et d'une IT-CAN.

Les cinq MC68332 et le PC communiquent au travers de leur IT-CAN, grâce à un bus CAN qui est une liaison multipoints. Cette architecture est donnée figure 1.4.5. La légende associée à cette figure est celle donnée dans la figure 1.3.2.

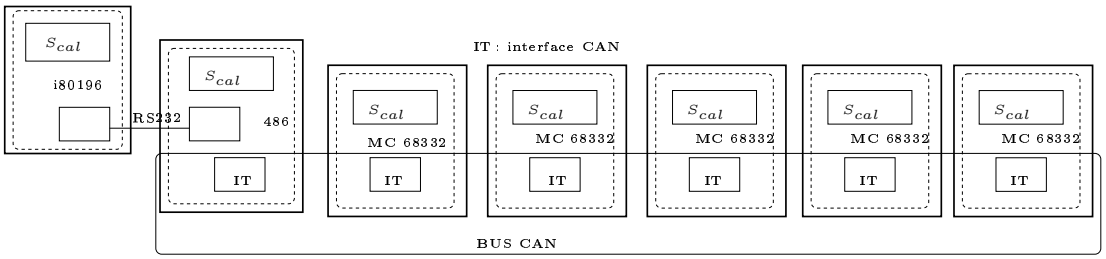


FIG. 1.4.5 – *Exemple d'architecture hétérogène*

2. Modèle d'algorithme

2.1 Hypergraphe orienté

2.1.1 Introduction

Notre graphe d'algorithme est un hypergraphe orienté [23, 43, 10] que nous allons enrichir avec les modèles définis ci-après.

Les sommets de l'hypergraphe orienté sont les *opérations de calcul* de l'algorithme et les arcs sont les *dépendances de données* entre opérations, également appelées *dépendances de données inter-opération*. Chaque opération est une suite indivisible d'instructions, appelée *région atomique* dans [89], c'est-à-dire qu'on ne pourra pas la partager pour en exécuter une partie sur un processeur et une autre sur un autre processeur. Autrement dit, ce sont des opérations non préemptives.

Les arcs induisent un ordre partiel d'exécution sur les opérations.

Une opération peut s'exécuter lorsque ses données d'entrée sont présentes. Elle consomme ses données d'entrée et produit des données en sortie qui sont ensuite utilisées par ses successeurs.

2.1.2 Formalisation

Soit G_{al}, le graphe de l'algorithme. G_{al} est un couple (O',D') où :

- O' est l'ensemble des opérations, appelés *opérations de calcul*, du graphe G_{al}. Card $O' = n'$, $O' = \{o'_i\}_{1 \leq i \leq n'}$.
- $D' \subseteq O' \times \mathcal{P}(O')$, est l'ensemble des arcs du graphe de l'algorithme, appelés *dépendances de données inter-opération*. Card $D' = k'$, $D' = \{d'_i\}_{1 \leq i \leq k'}$
  $d'_i = (o'_{i_1}, \{o'_{i_2}, \cdots, o'_{i_{k(i)}}\}_{2 \leq k(i) \leq n'})$, o'_{i_j} tous différents.
  $d'_i = (o'_{i_1}, \{o'_{i_j}\}_{2 \leq j \leq k(i) \leq n'})$, o'_{i_j} tous différents.

Associé à l'ensemble des dépendances D', on définit la fonction γ^{-1} qui, à chaque dépendance d'_i associe son opération émettrice $\gamma^{-1}(d'_i)$ et on définit la fonction γ qui, à chaque dépendance d'_i, associe l'ensemble des opérations réceptrices.

$$\boxed{\begin{array}{llll} \gamma^{-1}: & D' & \rightarrow & O' \\ & d'_i & \mapsto & \gamma^{-1}(d'_i) = (o'_{i_1}) \text{ où } d'_i = (o'_{i_1}, \{o'_{i_j}\}_{2 \leq j \leq k(i)}) \end{array}}$$

$$\boxed{\begin{array}{lll} \gamma: & D' & \rightarrow \mathcal{P}(O') \\ & d'_i & \mapsto \gamma(d'_i) = \{o'_{i_j}\}_{2 \leq j \leq n'} \text{ où } d'_i = (o'_{i_1}, \{o'_{i_j}\}_{2 \leq j \leq k(i)}) \end{array}}$$

Définition 12 *Un chemin de longueur q est une séquence de q hyperarcs tels que :* $\exists d'_{k_1},...,d'_{k_q} \in D' \times ... \times D'$ vérifiant *:*

$$\begin{array}{ll} \gamma^{-1}(d_{k_1}) = o'_i, & \gamma(d_{k_1}) \supseteq \{o'_{i+1}\} \\ \gamma^{-1}(d_{k_2}) = o'_{i+1} & \gamma(d_{k_2}) \supseteq \{o'_{i+2}\} \\ ... & ... \\ \gamma^{-1}(d_{k_q}) = o'_{i+q-1} & \gamma(d_{k_q}) \supseteq \{o'_{i+q}\} \end{array}$$

Définition 13 *Le graphe est sans circuit. Il n'existe pas de chemin ayant à la fois même origine et même destination.*

$$\begin{array}{l} \nexists\ d'_{k_1},...,d'_{k_n} \in D' \times ... \times D' \quad \text{vérifiant :} \\ \gamma^{-1}(d_{k_1}) = o'_i, \\ \gamma(d_{k_n}) \supseteq \{o'_i\} \\ \quad et \quad \forall l\ 1 < l < n \ \left\{ \begin{array}{l} \gamma(d_{k_l}) \supseteq \{o'_.\} \\ \gamma^{-1}(d_{k_{l+1}}) = \{o'_.\} \end{array} \right. \end{array}$$

Définition 14 *Chaque dépendance de données a un et un seul émetteur et au moins 1 récepteur.*

$$\boxed{\forall d'_i \in D' \quad \text{Card}(\gamma^{-1}(d'_i)) = 1 \quad \text{et} \quad Card(\gamma(d'_i)) \geq 1}$$

Remarque 16 *Quand $Card(\gamma(d'_i)) = 1$, d'_i est un arc possédant donc un émetteur et un unique récepteur.*

Quand $Card(\gamma(d'_i)) > 1$, d'_i est un hyperarc possédant un émetteur et plusieurs récepteurs, on dit que l'on a de la diffusion.

Associé au graphe G_{al}, on définit les notions de successeur, de descendant, de prédécesseur et d'ancêtre d'une opération.

Définition 15 Successeurs d'une opération o'_i au sens Gondran & Minoux [30] : $\Gamma(o'_i)$
Soit $\Gamma(o'_i)$, l'ensemble des successeurs d'une opération o'_i.

$$\Gamma(o'_i) = \{o'_j \in O' \quad / \quad \exists\, d'_k \in D' \quad avec \quad o'_i = \gamma^{-1}(d'_k) \quad et \quad o'_j \subseteq \gamma(d'_k)\}$$

Les opérations sans successeur du graphe G_{al} sont appelées les sorties.

$$Sorties(G_{al}) = \{o'_i \in O' \quad / \quad \Gamma(o'_i) = \emptyset\}$$

Définition 16 Descendants d'une opération o'_i au sens Gondran & Minoux [30] : $\hat{\Gamma}(o'_i)$
On suppose que l'opération o'_i est au maximum à n arcs d'un nœud sans successeur. Soit $\hat{\Gamma}(o'_i)$, l'ensemble des descendants d'une opération o'_i. C'est la fermeture transitive de l'ensemble des successeurs, c'est-à-dire :

$$\hat{\Gamma}(o'_i) = \bigcup_{k=1}^{n} \Gamma^k(o'_i)$$

où $\Gamma^k(o'_i)$ désigne l'ensemble des opérations que o'_i atteint en utilisant exactement k arcs.

Définition 17 Prédécesseurs d'une opération o'_i au sens Gondran & Minoux [30] : $\Gamma^{-1}(o'_i)$
Soit $\Gamma^{-1}(o'_i)$, l'ensemble des prédécesseurs d'une opération o'_i.

$$\Gamma^{-1}(o'_i) = \{o'_j \in O' \quad / \quad \exists\, d'_k \in D' \quad avec \quad o'_j = \gamma^{-1}(d'_k) \quad et \quad o'_i \subseteq \gamma(d'_k)\}$$

Les opérations sans prédécesseur du graphe G_{al} sont appelées les entrées.

$$Entrées(G_{al}) = \{o'_i \in O' \quad / \quad \Gamma^{-1}(o'_i) = \emptyset\}$$

Définition 18 Ancêtres d'une opération o'_i au sens Gondran & Minoux [30] : $\hat{\Gamma}^{-1}(o'_i)$
On suppose que l'opération o'_i est à exactement n arcs d'un nœud sans prédécesseur. Soit $\hat{\Gamma}^{-1}(o'_i)$, l'ensemble des ancêtres d'une opération o'_i. C'est la fermeture transitive de l'ensemble des prédécesseurs, c'est-à-dire :

$$\hat{\Gamma}^{-1}(o'_i) = \bigcup_{k=1}^{n} (\Gamma^{-1})^k(o'_i)$$

où $(\Gamma^{-1})^k(o'_i)$ désigne l'ensemble des opérations atteignant o'_i en utilisant exactement k arcs.

Relations de dépendances associées au graphe de l'algorithme

Il existe deux types de relations de dépendances entre les opérations composant le graphe de l'algorithme G_{al} .

1. les opérations dites *logiquement dépendantes* [13].
   Parmi cet ensemble, on distingue les opérations dites *données-dépendantes* et les opérations dites *transitive-dépendantes.*

   Définition 19 *L'ensemble des opérations données-dépendantes, noté $\preceq$ est défini par :*

   $$\preceq = \{(o'_i, o'_j) \in O' \times O' \quad / \quad \exists\, d'_k \in D' \quad avec \quad o'_i = \gamma^{-1}(d'_k) \quad et \quad \{o'_j\} \subseteq \gamma(d'_k)\}$$

   Ceci peut encore s'écrire de la manière suivante :

   $$\forall d'_i \in D', \quad si \quad o'_j \in \gamma(d'_i) \quad alors \quad \gamma^{-1}(d'_i) \preceq o'_j$$

Cet ensemble traduit un ordre d'exécution entre des opérations données-dépendantes. La relation $\preceq$ est la relation "est exécutée avant" sur l'ensemble des opérations.

Remarque 17 *L'ensemble $\preceq$ définit une relation d'ordre partiel d'exécution sur l'ensemble des opérations O'.*

Définition 20 *Soit $\preceq^*$, la fermeture transitive de la relation $\preceq$.*

Remarque 18 *La relation $\preceq^*$ définit un ordre partiel d'exécution sur l'ensemble des opérations du graphe de l'algorithme.*

Définition 21 *L'ensemble $\preceq_T$ des opérations transitive-dépendantes est l'ensemble des opérations qui appartiennent à la fermeture transitive $\preceq^*$ de la relation $\preceq$ mais pas à $\preceq$. Ces opérations sont donc dépendantes mais il n'y a pas de dépendances de données entre elles.*

$$\preceq_T = \preceq^* \setminus \preceq$$

$$\preceq_T = \{(o'_i, o'_j) \in O' \times O' \quad tq \quad \begin{array}{l} \exists d'_{k_1}, \ldots, d'_{k_n} \in D' \times \ldots \times D' \quad \text{vérifiant :} \\ \gamma^{-1}(d_{k_1}) = o'_i, \\ \gamma(d_{k_n}) \supseteq \{o'_j\} \\ \gamma(d_{k_l}) \supseteq \{o.\} \quad (1 < l < n) \\ \gamma^{-1}(d_{k_{l+1}}) = o.\} \end{array}$$

Finalement, l'ensemble des opérations logiquement dépendantes est caractérisée par la fermeture transitive $\preceq^*$ de la relation $\preceq$.

2. les opérations dites *logiquement indépendantes* [13] ou encore *concurrentes* [89].

Définition 22 *L'ensemble des opérations logiquement indépendantes est l'ensemble, noté $\gtrless$, des opérations O' appartenant au complémentaire de la relation $\preceq^*$. La relation $\gtrless$ est définie par :*

$$\gtrless \;=\; O' \times O' \setminus \preceq^*$$

$$\gtrless = \{(o'_i, o'_j) \in O' \times O' \; tq \; (o'_i, o'_j) \notin \preceq^*\}$$

Remarque 19 *Cet ensemble $\gtrless$ caractérise le parallélisme potentiel de l'algorithme et on cherchera à l'exploiter si le parallélisme disponible de l'architecture le permet.*

Propriété 3 *$\gtrless$ et $\preceq^*$ forment une partition du produit cartésien de l'ensemble O' des opérations.*
$\gtrless \cup \preceq^* = O' \times O'$ et $\gtrless \cap \preceq^* = \emptyset$

Définition 23 *Exécution*
On appelle exécution du graphe (O',D')*, un ordre partiel* ξ *sur l'ensemble* O'*, qui inclut l'ordre partiel* $\preceq^*$ *initial. Soit :*

$$\boxed{\xi \supseteq \preceq^*}$$

Propriétés du graphe G_{al}

Propriété 4 *L'ensemble des exécutions possibles du graphe* (O',D') *est l'ensemble* Ξ *des ordres qui incluent l'ordre partiel initial* $\preceq^*$*.*

$$\boxed{\Xi = \{\xi \in O' \times O' \quad tq \quad \xi \supseteq \preceq^*\}}$$

Propriété 5 *L'ensemble des opérations sans prédécesseur définit l'ensemble des éléments minimaux par rapport à* $\preceq$ *de* O'*.* Preuve : une opération o'_i sans prédécesseur vérifie :

$$\Gamma^{-1}(o'_i) = \emptyset \Rightarrow \nexists d'_k \quad tq \quad \gamma(d'_k) \supseteq \{o'_i\}$$
$$\Rightarrow \nexists o'_j = \gamma^{-1}(d'_k) \quad tq \quad o'_j \preceq o'_i$$
$$\Rightarrow \forall o'_j \in O' \quad o'_j \preceq o'_i \Rightarrow o'_j = o'_i$$

Propriété 6 *L'ensemble des opérations sans successeur définit l'ensemble des éléments maximaux par rapport à* $\preceq$ *de* O'*.* Preuve : une opération o'_i sans successeur vérifie :

$$\Gamma(o'_i) = \emptyset \Rightarrow \nexists d'_k \quad tq \quad o'_i = \gamma^{-1}(d'_k)\}$$
$$\Rightarrow \nexists o'_j \in \gamma(d'_k) \quad tq \quad o'_i \preceq o'_j$$
$$\Rightarrow \forall o'_j \in O' \quad o'_i \preceq o'_j \Rightarrow o'_j = o'_i$$

2.1.3 Représentation graphique

1. Mise en évidence des dépendances intra-opération
   Si l'on considère le graphe dual défini dans [55], les sommets sont les dépendances de données et les arcs les opérations qui émettent et qui reçoivent ces dépendances. Cette représentation met en évidence les dépendances intra-opération. On préférera l'autre représentation, car bien que celle-ci soit duale à celle que l'on va maintenant définir, elle est plus difficile à "lire".
2. Mise en évidence des dépendances inter-opération
   Les opérations du graphe sont représentées par des cercles en trait continu et les dépendances de données entre ces opérations sont représentées par des arcs en trait continu.

   Remarque 20 *On verra dans la section suivante un nouveau type de dépendances : les dépendances de conditionnement qui sont représentées par des traits pointillés.*

On constate que le graphe de l'algorithme est composé parfois de parties régulières, c'est-à-dire d'opérations identiques connectées de manière régulière. Prenons l'exemple d'un sommet qui a deux entrées x et y, qui réalise une addition $+$ et qui donne en sortie le résultat de cette addition $x+y$. Si l'on veut additionner les 10 composantes de deux vecteurs U et V, c'est plus simple de ne saisir qu'un sommet et d'indiquer qu'il doit être effectué sur chacune des 10 composantes des deux vecteurs, ce qui revient à effectuer une addition vectorielle. Bien que dans la formalisation du graphe tous les sommets sont différents, il est intéressant lors de la représentation du graphe de donner le même nom aux sommets qui effectuent la même opération. Ainsi, les sommets portant le même nom peuvent être encapsulés en un seul que l'on appelle *opération répétitive* (cf. fig. 2.1.1), cela correspond à la factorisation finie d'une partie du graphe de l'algorithme. La sémantique associée à un sommet répétitif est donnée figure 2.1.1. La figure 2.1.2 donne la liste des répétitions élémentaires et des factorisations associées que l'on peut trouver dans un graphe.

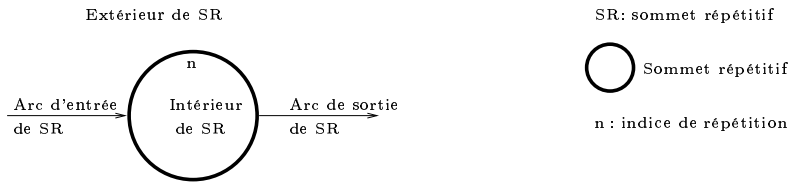


FIG. 2.1.1 – *Définitions associées à un sommet répétitif*

Définition 24 *Sommet répétitif*
C'est un sommet particulier qui encapsule n sommets identiques. Ce sommet particulier a la spécificité suivante : un arc émis ou reçu par ce sommet n'a pas la même signification à l'extérieur et à l'intérieur de ce sommet. Le sommet répétitif est représenté par un cercle dessiné avec un trait continu épais.

Définition 25 *Arc d'entrée du sommet répétitif*
C'est un arc de dépendances de données particulier qui a, parmi ses récepteurs, le sommet répétitif.

Définition 26 *Arc de sortie du sommet répétitif*
C'est un arc de dépendances de données émis par le sommet répétitif.

Un sommet répétitif possède trois types d'arcs d'entrée et deux types d'arcs de sorties (cf. tableaux 2.1.1 et 2.1.2) illustrés à la figure 2.1.2.

Plus précisément, l'arc d'entrée I du sommet répétitif (SR) fournit à chaque sommet répété A la sortie du précédent sommet répété (une valeur initiale au premier sommet répété). L'arc de sortie I du (SR) transmet la sortie produite par le dernier sommet répété.

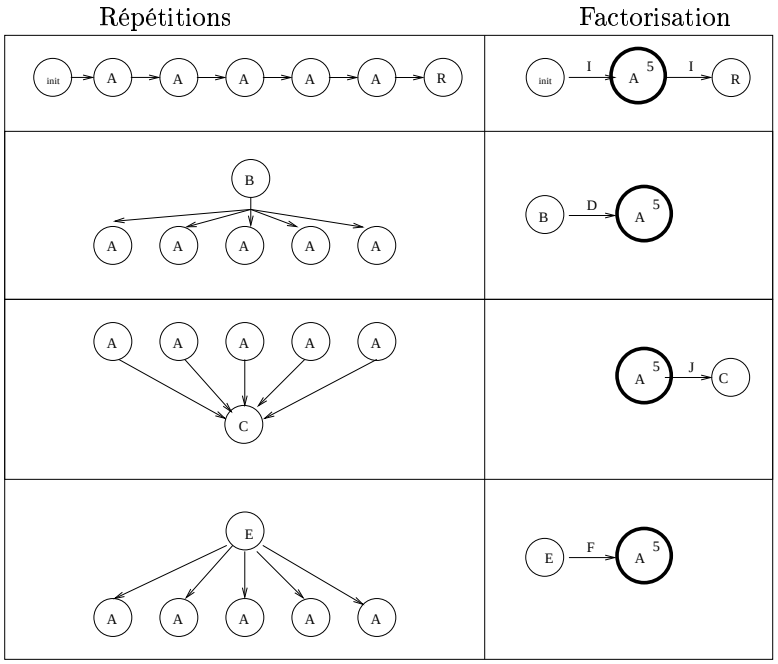


FIG. 2.1.2 – *Factorisations élémentaires*

L'arc d'entrée D du SR fournit à chaque sommet répété A la même donnée.

L'arc de sortie J du SR fournit à C l'ensemble des composantes d'un vecteur dont chaque composante a été produite par un sommet répété A.

Enfin, l'arc d'entrée F du sommet répétitif fournit à chaque sommet répété A une des composantes du vecteur fourni par E.

	extérieur	intérieur
D(n)	n	n
F(n)	n	1
I(n)	n	n

TAB. 2.1.1 – *Arcs d'entrée*

	intérieur	extérieur
J(n)	1	n
I(n)	n	n

TAB. 2.1.2 – *Arcs de sortie*

Un exemple de graphe factorisé en parties est donné figure 2.1.3. Il montre la simplification de la représentation graphique que représente la factorisation. On passe dans le cas de

cet exemple de 10 à 5 sommets.

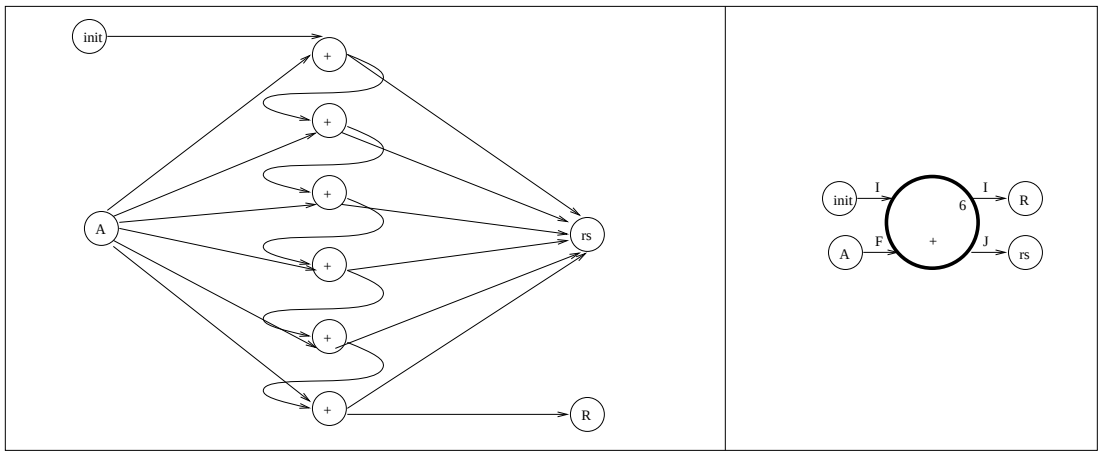


FIG. 2.1.3 – *Exemple de factorisation finie de graphe*

La factorisation que nous venons de définir réduit la granularité du graphe de l'algorithme, ce qui permet d'augmenter la rapidité de l'heuristique d'ordonnancement. Pour avoir une granularité convenable, Cosnard, Jeannot et Yang proposent dans [20] de construire un GTP- Graphe de tâches paramétrées-. Ce GTP est un modèle de calcul qui est petit et indépendant de la taille des données. Il se déduit d'un programme séquentiel. Plus précisément considérons un programme contenant des nids de boucles. Le programme est annoté avec les mots clé *tâche* et *fin de tâche* pour délimiter les parties du programme qui vont devoir s'exécuter en séquentiel, ces parties vont correspondre aux sommets du GTP. Ensuite, des lois sont élaborées pour construire les dépendances du GTP. Une fois ce graphe construit, des heuristiques de clustering [28] ayant une faible complexité sont appliquées pour construire un ordonnancement associé au graphe de l'algorithme initial.

2.2 Hypergraphe conditionné

2.2.1 Introduction

On ajoute un nouveau type d'arcs de dépendances de données : ce sont des arcs de dépendances dit de *conditionnement*, qui induisent une condition d'activation sur chaque opération réceptrice d'un tel arc. Ce type de dépendances existe également dans le format commun DC [78] des langages synchrones. Chaque opération qui reçoit un arc de dépendance de conditionnement est dite *conditionnée* et cette opération ne sera exécutable que si sa condition d'activation est vraie et que ses autres données d'entrée sont présentes.

Définition 27 *Un graphe G_{al} représentant un graphe conditionné est un graphe qui a deux*

sortes d'opérations et deux sortes d'arcs :

- *des opérations* non conditionnées, *c'est-à-dire des opérations qui s'exécutent sans condition,*
- *des opérations* conditionnées, *c'est-à-dire des opérations dont l'exécution dépend d'une condition d'activation définie ci-après. De plus, chaque opération conditionnée est conditionnée par un et un seul booléen, plus précisément par l'une des deux valeurs* Vrai *ou* Faux *que pourra prendre le booléen qui la conditionne.*
- *des arcs de dépendances de données qui induisent un ordre partiel d'exécution entre les opérations,*
- *des arcs de dépendances de conditionnement, encore appelés* conditions d'activation. *Les arcs de dépendances de conditionnement induisent une condition d'activation sur les opérations réceptrices, i.e. l'exécution de chaque opération réceptrice dépend du booléen qui la conditionne. En d'autres termes, si le booléen b est vrai et que l'opération est conditionnée par b vrai alors l'opération pourra s'exécuter. Dans le cas inverse, si le booléen est faux et que l'opération est conditionnée par vrai alors l'opération ne pourra pas s'exécuter.*

  Ces arcs sont représentés dans la suite par des arcs en trait pointillé, comme défini à la section 2.2.3.

2.2.2 Formalisation

Graphe conditionné

G_{al} est un couple (O',D'), O' et D' ayant déjà été définis dans le cas de l'hypergraphe.

- O', l'ensemble des opérations se décompose en deux sous-ensembles disjoints :
  - O'_{ncond}, l'ensemble des opérations non conditionnées,
  - O'_{cond}, l'ensemble des opérations conditionnées.
- D', l'ensemble des dépendances de données se décompose en deux sous-ensembles disjoints :
  - D'_d, l'ensemble des dépendances de données "simples",
  - D'_c, l'ensemble des dépendances de données *et* de conditionnement, qu'on appellera par la suite dépendances de conditionnement.

Soit B, l'ensemble des booléens qui peuvent conditionner les opérations.

$$B = \{b_1,\bar{b}_1,b_2,\bar{b}_2,...,b_i,\bar{b}_i,...,b_n,\bar{b}_n\}$$

Nous notons b et $\bar{b}$, les deux valeurs que peuvent prendre chaque booléen b. Par la suite, au lieu de parler de valeur associée à un booléen, on parlera de booléen pour alléger le texte. On dira alors que b et $\bar{b}$ sont deux booléens complémentaires ou encore corrélés.

Soit ϕ, la fonction qui, à une opération associe le booléen qui la conditionne.

$$\boxed{\begin{array}{llll} \phi: & O' & \rightarrow & B \\ & o'_i & \mapsto & \phi(o'_i) = b \end{array}}$$

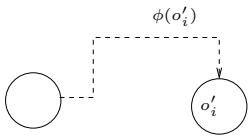


FIG. 2.2.4 – *o'_i est conditionnée par $\phi(o'_i)$*

Le domaine de définition $\mathcal{D}_\phi$ de la fonction ϕ est l'ensemble des opérations conditionnées soit : $\mathcal{D}_\phi = O'_{cond}$.

Soit $\phi(O')$, l'ensemble des valeurs de la fonction ϕ.

Pour construire l'application réciproque d'une fonction, la fonction doit être surjective. Soit ϕ_s, la fonction ϕ à valeurs dans $\phi(O')$. ϕ_s est une fonction surjective.

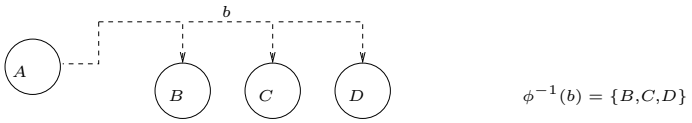


FIG. 2.2.5 – *b conditionne $\phi^{-1}(b)$*

Soit ϕ^{-1}, l'application réciproque de ϕ_s qui associe à chaque booléen l'ensemble des opérations qu'il conditionne.

$$\boxed{\begin{array}{llll} \phi^{-1}: & \phi(O') & \rightarrow & \mathcal{P}(O') \\ & b & \mapsto & \phi^{-1}(b) = \{o'_i \in O' / \phi_s(o'_i) = b\} \end{array}}$$

Ainsi, $\phi^{-1}(b)$ est l'ensemble des opérations conditionnées par b et $\phi^{-1}(\bar{b})$ est l'ensemble des opérations conditionnées par $\bar{b}$.

Définition 28 *Soient $\phi^{-1}(b)$ et $\phi^{-1}(\bar{b})$ deux ensembles d'opérations conditionnées par deux booléens corrélés. On dit que les ensembles sont exclusifs. Les deux ensembles ne seront pas exécutés au cours de la même exécution.*

Ainsi, si on considère le graphe de l'algorithme de la figure 2.2.8, on a : B,C,D et E,F sont des ensembles exclusifs, c'est-à-dire qu'ils ne seront pas présents au cours de la même exécution car $\{B,C,D\} \in \phi^{-1}(b)$ et $\{E,F\} \in \phi^{-1}(\bar{b})$.

Soit η, l'application qui, à chaque booléen de conditionnement associe son opération émettrice.

$$\begin{array}{llll} \eta : & B & \rightarrow & O' \\ & b_i & \mapsto & \eta(b_i) = o' \end{array}$$

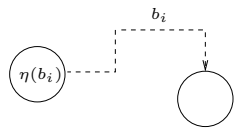


FIG. 2.2.6 – b_i *est émis par* $\eta(b_i)$

On suppose que tout booléen est émis. Soit η^{-1}, l'application qui, à chaque opération associe les booléens et leurs complémentaires qu'elle émet.

$$\begin{array}{llll} \eta^{-1} : & O' & \rightarrow & \mathcal{P}(B \times B) \\ & o'_i & \mapsto & \eta^{-1}(o'_i) = \{(b_{i_j},\bar{b}_{i_j})_{1 \leq j \leq n_i}\} \end{array}$$

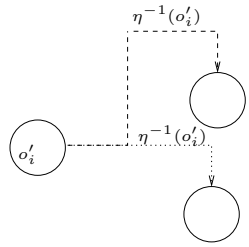


FIG. 2.2.7 – o'_i *émet* $\eta^{-1}(o'_i)$

Ainsi, si on considère le graphe de l'algorithme décrit dans la figure 2.2.8, on a : $\eta(b) = \eta(\bar{b}) = A$, $\eta(c) = \eta(\bar{c}) = F$ et en utilisant la fonction réciproque η^{-1} : $\eta^{-1}(A) = \{b,\bar{b}\}$, $\eta^{-1}(F) = \{c,\bar{c}\}$.

Arborescence de conditionnement

Soit G^h_{al}, *l'arborescence de conditionnement* associée au graphe G_{al}.

Définition 29 *On appelle* arborescence de conditionnement *du graphe* G_{al}*, le graphe* G^h_{al}

qui contient tous les sommets du graphe et qui n'a conservé que les dépendances de conditionnement. Le graphe de l'arborescence de conditionnement est indépendant des dépendances de données du graphe G_{al}.

$$G_{al}^h = (O',D'_c)$$

Définition 30 (issue de [30].) *$\mathcal{F} = (\mathcal{X},\mathcal{T})$ est une arborescence s'il existe un sommet r, appelé racine, qui est relié à tous les autres sommets par un chemin unique d'origine r. On appelle feuille de l'arborescence une opération sans successeur.*

Définition 31 *G_{al}^h est composé d'un ensemble de composantes connexes et chaque composante connexe est une arborescence. De plus,G_{al}^h peut avoir plusieurs opérations sans prédécesseur et plusieurs opérations sans successeur.*

Un exemple d'arborescence de conditionnement est donné figure 2.2.8. Les arcs de G_{al}^h sont

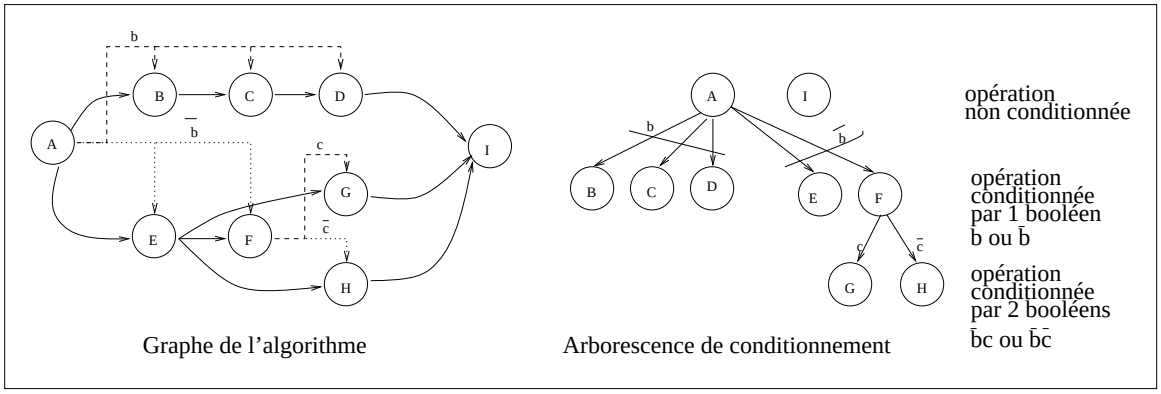


FIG. 2.2.8 – *Exemple d'arborescence de conditionnement*

étiquetés par les booléens de conditionnement. Ce graphe représentant les dépendances de conditionnement permet de mettre en évidence les opérations exclusives. Dans l'exemple de la figure 2.2.8, les ensembles $\{B,C,D\}$ et $\{E,F\}$ sont exclusifs ainsi que les singletons $\{G\}$ et $\{H\}$. De même par héritage, on sait que :

G est conditionné par c émis par F.

F est conditionné par $\bar{b}$ émis par A.

A n'est pas conditionné.

On en déduit que G ne pourra être exécuté que si c est *Vrai* lui-même produit par F qui n'est exécuté que si b est *Faux*. Ainsi, G appartient à la branche $\bar{b}c$ (notion définie à l'issue de l'exemple). Il pourra donc être ordonnancé après la dernière opération provenant de cette branche. Plus précisément, à chaque booléen, on associe l'opération qui émet ce booléen,

on regarde par quoi est conditionnée cette opération et on remonte ainsi jusqu'à un nœud racine.

Branche de conditionnement

La branche de conditionnement $\mathcal{B}(o'_i)$ de l'opération o'_i est la séquence des booléens de conditionnement qui conditionne l'opération o'_i et ses ancêtres.

Construction de la branche de conditionnement d'une opération o'_i : On suppose que n est le nombre d'arcs qui composent le chemin d'une des racines de G^h_{al} à o'_i. La construction de la branche de conditionnement de o'_i est décrite dans la table 2.2.3.

o'_i	est conditionnée par	$\phi(o'_i)$	émis par	$\eta[\phi(o'_i)]$.
$\eta[\phi(o'_i)]$	est conditionnée par	$\phi \circ (\eta \circ \phi)(o'_i)$	émis par	$(\eta \circ \phi)^2(o'_i)$.
$(\eta \circ \phi)^2(o'_i)$	est conditionnée par	$\phi \circ (\eta \circ \phi)^2(o'_i)$	émis par	$(\eta \circ \phi)^3(o'_i)$.
...	est conditionnée par	...	émis par	...
$(\eta \circ \phi)^{n-2}(o'_i)$	est conditionnée par	$\phi \circ (\eta \circ \phi)^{n-2}(o'_i)$	émis par	$(\eta \circ \phi)^{n-1}(o'_i)$.
$(\eta \circ \phi)^{n-1}(o'_i)$	n'est pas conditionnée			

TAB. 2.2.3 – *Construction d'une branche de conditionnement*

L'ensemble des booléens de conditionnement qui conditionnent o'_i par héritage figure dans la colonne centrale du tableau 2.2.3.

Définition 32 *Soit $\mathcal{B}(o'_i)$ la branche de conditionnement de o'_i : c'est l'ensemble des booléens qui conditionnent o'_i par héritage.*

$$\mathcal{B}(o'_i) = \{\phi \circ (\eta \circ \phi)^{n-2}(o'_i), \phi \circ (\eta \circ \phi)^{n-1}(o'_i), ..., \phi \circ (\eta \circ \phi)^1(o'_i), \phi \circ (\eta \circ \phi)^0(o'_i)\}$$

en supposant que $(\eta \circ \phi)^0(o'_i) = o'_i$. Ceci peut encore s'écrire :

$$\mathcal{B}(o'_i) = \bigcup_{k=0}^{n-2} \{\phi \circ (\eta \circ \phi)^k(o'_i)\}$$

Ainsi, si on reprend l'exemple de la figure 2.2.8, on a :

$\mathcal{B}(A) = \mathcal{B}(I) = \emptyset$

$\mathcal{B}(B) = \mathcal{B}(C) = \mathcal{B}(D) = \{b\}$

$\mathcal{B}(E) = \mathcal{B}(F) = \{\bar{b}\}$

$\mathcal{B}(G) = \{\bar{b}, c\}$

$\mathcal{B}(H) = \{\bar{b}, \bar{c}\}$

Définition 33 Prédécesseur de conditionnement d'une opération o'_i : $\Gamma_c^{-1}(o'_i)$

On appelle prédécesseur de conditionnement d'une opération o'_i, l'opération notée $\Gamma_c^{-1}(o'_i)$,

qui a émis le booléen qui conditionne o'_i. Or o'_i est conditionné par $\phi(o'_i)$ (par définition) qui est émis par $(\eta \circ \phi)(o'_i)$ (toujours par définition). Ainsi : $\Gamma_c^{-1}(o'_i) = (\eta \circ \phi)(o'_i)$.

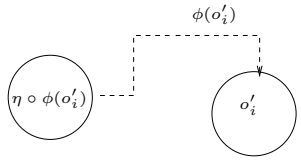


FIG. 2.2.9 – *o'_i est conditionnée par $\phi(o'_i)$ émis par $\eta \circ \phi(o'_i)$*

Cette notion peut être étendue en prenant la fermeture transitive à tous les ancêtres de conditionnement de o'_i.

Définition 34 Ancêtres de conditionnement d'une opération o'_i : $\hat{\Gamma}_c^{-1}(o'_i)$
On appelle ancêtres de conditionnement d'une opération o'_i noté $\hat{\Gamma}_c^{-1}(o'_i)$, l'union du prédécesseur de conditionnement de o'_i, du prédécesseur de conditionnement du prédécesseur de conditionnement de o'_i ... jusqu'à ce qu'on arrive à une des racines de l'arborescence, par exemple la racine $r_{\mathcal{A}}$. C'est donc égal à la fermeture transitive des prédécesseurs de conditionnement. Ainsi : $\hat{\Gamma}_c^{-1}(o'_i) = \bigcup_{k=1}^{n} \{(\eta \circ \phi)^k(o'_i)\}$ *où n désigne la longueur du chemin de o'_i à $r_{\mathcal{A}}$.*

Si on reprend le tableau 2.2.3 sur la construction de la branche de conditionnement de l'opération o'_i, on constate que les ancêtres de conditionnement correspondent à la dernière colonne du tableau.

Définition 35 Successeur de conditionnement d'une opération o'_i : $\Gamma_c(o'_i)$
On appelle successeur de conditionnement d'une opération o'_i, l'ensemble des opérations noté $\Gamma_c(o'_i)$, qui sont conditionnées par le booléen que o'_i a émis. Or o'_i émet le booléen de conditionnement $\eta^{-1}(o'_i)$ (par définition) et ce booléen conditionne $(\phi^{-1} \circ \eta^{-1})(o'_i)$ (toujours par définition). Ainsi : $\Gamma_c(o'_i) = (\phi^{-1} \circ \eta^{-1})(o'_i)$.

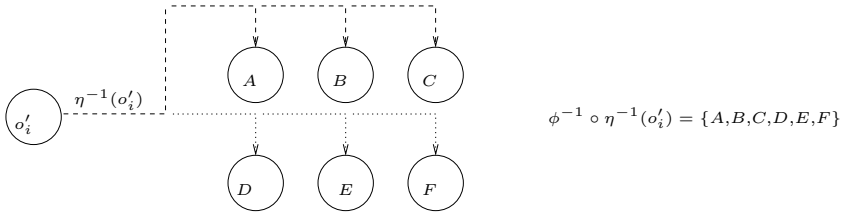


FIG. 2.2.10 – *o'_i émet $\eta^{-1}(o'_i)$ qui conditionne l'ensemble $\phi^{-1} \circ \eta^{-1}(o'_i)$*

Remarque 21 *On aurait pu directement à partir du prédécesseur de conditionnement définir le successeur du conditionnement. En effet :* $\Gamma_c(o'_i) = (\Gamma_c^{-1})^{-1}(o'_i) \stackrel{def}{=} (\eta \circ \phi)^{-1}(o'_i) \stackrel{def}{=} (\phi^{-1} \circ \eta^{-1})(o'_i)$.

La notion de successeur de conditionnement peut être étendue en prenant la fermeture transitive à tous les descendants de conditionnement de o'_i.

Définition 36 Descendants de conditionnement d'une opération o'_i : $\hat{\Gamma}_c(o'_i)$
On appelle descendants de conditionnement d'une opération o'_i noté $\hat{\Gamma}_c(o'_i)$, l'union des successeurs de conditionnement de o'_i, des successeurs de conditionnement des successeurs de conditionnement de o'_i ... jusqu'à ce qu'on arrive à une feuille de l'arborescence. C'est donc égal à la fermeture transitive des successeurs de conditionnement. Ainsi : $\hat{\Gamma}_c(o'_i) = \bigcup_{k=1}^{n} (\phi^{-1} \circ \eta^{-1})^k(o'_i)$ *où n désigne la longueur maximale de tous les chemins de o'_i à une feuille de l'arborescence.*

Nous allons étendre la notion de descendants de conditionnement d'une opération à la notion de descendants de conditionnement d'un ensemble d'opérations car on en a besoin pour la suite.

Définition 37 Descendants de conditionnement d'un ensemble d'opérations E : $\hat{\hat{\Gamma}}_c(E)$
On définit les descendants de conditionnement d'un ensemble d'opérations E comme l'union de tous les descendants de conditionnement de chaque opération appartenant à l'ensemble E. Ainsi : $\hat{\hat{\Gamma}}_c(E) = \bigcup_{o'_i \in E} \hat{\Gamma}_c(o'_i)$

Propriétés du graphe G_{al}

Propriété 7 *Les ensembles O'_{cond} et O'_{ncond} forment une partition de l'ensemble des opérations O'.* En effet, on a vu par construction que chaque opération du graphe est soit conditionnée, soit non conditionnée.

Propriété 8 *ϕ est une fonction non injective et non surjective.* En effet, toute opération conditionnée du graphe est conditionnée par un et un seul booléen. ($\forall o'_i \in O'_{cond} \exists! \phi(o'_i) \in B$). *De plus, un booléen peut conditionner plusieurs opérations. ($\phi(o'_i) = \phi(o'_j) \not\Rightarrow o'_i = o'_j$). Enfin, un booléen peut ne pas conditionner d'opérations. L'ensemble $\phi(O')$ n'est qu'un sous-ensemble de B. ($\phi(O') \subset B$).*

Propriété 9 *ϕ^{-1} est une injection.* En effet, toute opération conditionnée est conditionnée par un et un seul booléen. ($\forall\, b,b' \in \phi_s(O') \times \phi_s(O')\ [\phi^{-1}(b) = \phi^{-1}(b') \Rightarrow b = b']$).

Propriété 10 *η est une fonction non injective et non surjective.* En effet, un booléen de conditionnement ne peut être émis que par une et une seule opération. De plus, une opération qui émet un booléen de conditionnement, émet toujours un booléen et son complémentaire, $(\eta(b_i) = \eta(b_j) \not\Rightarrow b_i = b_j)$. Enfin, les opérations de G_{al} n'émettent pas forcément toutes des booléens de conditionnement.

Propriété 11 *A un graphe de dépendances conditionné donné correspond un unique graphe représentant l'arborescence de conditionnement. Réciproquement, à une arborescence de conditionnement donnée correspond plusieurs graphes de dépendances de données.* En effet, pour passer d'un graphe de dépendances conditionné à son arborescence associée, on ne fait que supprimer les arcs de dépendances de données. Réciproquement, pour passer d'une arborescence de conditionnement à un graphe de dépendances, on ajoute des arcs de dépendances de données.

Propriété 12 *Une opération a un et un seul prédécesseur de conditionnement.* En effet, une opération conditionnée n'est conditionnée que par un et un seul booléen (par définition).

Propriété 13 *Chaque opération possède une et une seule branche de conditionnement.* Ceci découle de la propriété précédente, l'opération a un unique prédécesseur de conditionnement qui lui-même a un unique prédécesseur de conditionnement et ainsi de suite.

Nous allons maintenant définir la notion d'exclusivité associée au conditionnement.

Relations associées au graphe de l'algorithme conditionné

On associe deux nouvelles relations au graphe de l'algorithme conditionné : la relation de dépendances de conditionnement notée $\preceq_{if}$ et la relation d'exclusivité notée Ex. Nous allons maintenant définir successivement ces deux relations.

Remarque 22 *Les relations définies sur le graphe non conditionné sont encore valables dans le cas conditionné.*

Relation de dépendances de conditionnement

On définit la relation $\preceq_{if}$ par $o'_i \preceq_{if} o'_j \Leftrightarrow o'_i$ "conditionne" o'_j.

$$\preceq_{if} = \{(o'_i, o'_j) \in O' \times O'_{cond} \quad / \quad \exists d'_k \in D'_c \quad avec \quad o'_i = \gamma^{-1}(d'_k) \quad et \quad \{o'_j\} \subseteq \gamma(d'_k)\}$$

ceci est équivalent à :

$$\preceq_{if} = \{(o'_i, o'_j) \in O' \times O'_{cond} \quad / \quad o'_i = \Gamma_c^{-1}(o'_j)\}$$

ce qui peut encore s'écrire :

$$\forall d'_k \in D'_c, \quad si \quad o'_j \in \gamma(d'_k) \quad alors \quad \gamma^{-1}(d'_k) \preceq_{if} o'_j$$

Relation d'exclusivité

L'opération o'_i est conditionnée par $\phi(o'_i)$ (par définition). Les opérations conditionnées par le complémentaire de $\phi(o'_i)$, noté $\bar{\phi}(o'_i)$, sont définies par l'ensemble $\phi^{-1} \circ \bar{\phi}(o'_i)$ (par définition).

Au cours d'une exécution du graphe, soit o'_i sera exécutée, soit l'ensemble des opérations $\phi^{-1} \circ \bar{\phi}(o'_i)$ seront exécutées. On dit que o'_i et $\phi^{-1} \circ \bar{\phi}(o'_i)$ sont exclusifs. Nous allons maintenant définir la notion d'exclusivité plus précisément.

Définition 38 Opérations o'_i et o'_j exclusives : $o'_i \ Ex \ o'_j$
o'_i et o'_j sont exclusives si et seulement si o'_i et o'_j ne sont pas présentes à la même exécution.

$$Ex = \{(o'_i, o'_j) \in O' \times O' \quad / \quad \phi(o'_i) = \bar{\phi}(o'_j)\}$$

Définition 39 *La relation Ex se construit à partir des dépendances de conditionnement du graphe et est indépendante des dépendances de données du graphe.*

La notion d'exclusivité se généralise aux ensembles.

Définition 40 Ensembles exclusifs
O'_1 et O'_2 sont des ensembles exclusifs si et seulement si : $\forall o'_1 \in O'_1, \forall O'_2 \in o'_2$ on a : $o'_1 \ Ex \ o'_2$.
On notera $O'_2 = Ex(O'_1) \Leftrightarrow O'_1 = Ex(O'_2)$, par symétrie de la relation.

On peut étendre la notion d'exclusivité à tous les descendants de conditionnement d'une opération ou d'un ensemble d'opérations. Comme on l'a vu précédemment, l'opération o'_i et l'ensemble des opérations $\phi^{-1} \circ \bar{\phi}(o'_i)$ sont exclusifs. Et donc, par héritage du booléen de conditionnement, les descendants de conditionnement de o'_i sont exclusifs des descendants de l'ensemble $\phi^{-1} \circ \bar{\phi}(o'_i)$, soit

$$\forall o'_k \in \hat{\Gamma}_c(o'_i), \ \forall o'_l \in \hat{\hat{\Gamma}}_c(\phi^{-1} \circ \bar{\phi}(o'_i)) \quad o'_k \ Ex \ o'_l$$

Nous allons maintenant donner quelques propriétés associées à la relation Ex.

Propriété 14 *Ex n'est pas réflexive.*
En effet, si une opération est présente à une exécution, elle ne peut être absente à cette même exécution.

Propriété 15 *Ex est une relation symétrique : si $o'_1 \ Ex \ o'_2 \Leftrightarrow o'_2 \ Ex \ o'_1$. (par définition)*

Propriété 16 *Ex n'est pas une relation transitive.*
Si o'_1 Ex o'_2 et o'_2 Ex o'_3 alors ou o'_1 et o'_3 ne sont pas exclusives (cas 1 de la figure 2.2.11) ou o'_1 Ex o'_3 (cas 2 de la figure 2.2.11).

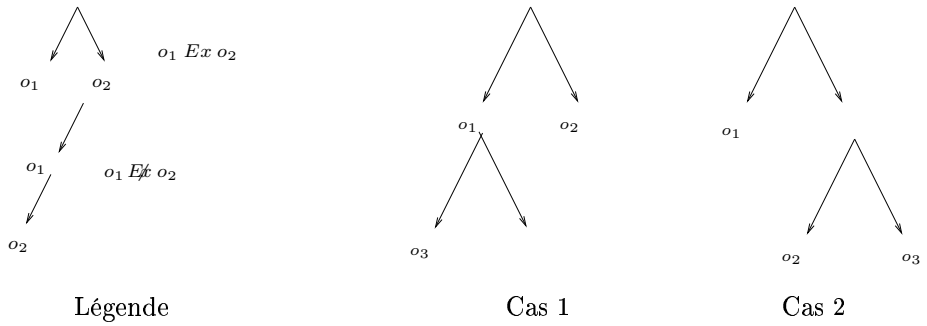


FIG. 2.2.11 – *La relation Ex n'est pas transitive. En effet, si l'on considère le cas 1, o_1 et o_2 sont exclusives, o_2 et o_3 sont exclusives, mais o_1 et o_3 ne sont pas exclusives. Si l'on considère le cas 2, o_1 et o_2 sont exclusives tout comme o_2 et o_3, et cette fois-ci o_1 et o_3 sont également exclusives.*

Propriété 17 *Si o'_1 Ex o'_2 alors $\hat{\Gamma}_c(o'_1)$ Ex o'_2.* En effet, la branche de conditionnement de o'_1 est incluse dans la branche de conditionnement de ses descendants.

Propriété 18 *Toute exécution du graphe conditionné (O',D') est un ordre partiel ξ sur un sous-ensemble SO' de O' vérifiant :*

$$\boxed{\xi \supseteq \preceq \downarrow \ (O' \setminus \ Ex(SO'))^*}$$

où $Ex(SO')$ désigne l'ensemble exclusif de SO'. Ainsi, l'ordre d'exécution ξ sur le sous-ensemble SO' inclut l'ordre partiel $\preceq$ projeté sur l'ensemble des opérations O' privé de l'ensemble des opérations exclusives $Ex(SO')$ de l'ensemble SO'.

2.2.3 Représentation graphique

Les arcs de conditionnement sont représentés en trait pointillé afin que l'on puisse les différencier des arcs de dépendances de données simples.

2.3 Hypergraphe conditionné infiniment itéré

Les applications réactives numériques interagissent avec leur environnement physique de manière discrète, donc répétitive. Une séquence lecture capteur, calculs, écriture actionneur

est réalisée à partir de l'événement valué fourni par le capteur qui effectue l'interface avec l'environnement. Chaque application est modélisée par un graphe acyclique de dépendances de données entre opérations, constitué d'un motif infiniment répété, dont la factorisation correspond à la notion classique de *graphe flot de données* [6]. Une opération sans prédécesseur (resp. sans successeur) représente une interface d'*entrée*, capteur, (resp. de *sortie*, actionneur) avec l'environnement. Ce motif est un graphe orienté conditionné tel que défini précédemment. Il peut exister des dépendances entre ces motifs qui, elles aussi, sont répétitives. Certaines de ces dépendances inter-motif peuvent conduire à des cycles lorsqu'on factorise le graphe selon son motif répétitif. Ces cycles doivent être marqués afin d'optimiser la distribution et l'ordonnancement du graphe pour son exécution.

Prenons l'exemple de la régulation du niveau d'eau dans un bassin. L'eau de ce bassin peut s'évaporer et un certain niveau d'eau doit être maintenu pour préserver l'écosystème. Un capteur transmet à l'application le niveau du fluide. A chaque fois que le capteur transmet ce niveau au graphe modélisant l'application, si la valeur du capteur est inférieure à un certain seuil, c'est une partie du graphe qui s'exécute pour réaliser les calculs conduisant à l'ouverture d'une vanne pour remplir le bassin. Si cette valeur est supérieure au seuil, c'est une autre partie du graphe qui s'exécute pour vider le bassin. On voit ainsi que, d'une part, un graphe est exécuté à chaque fois qu'une valeur arrive dans l'application, ce qui correspond à de la répétition de graphes, et d'autre part, que suivant la valeur du capteur, ce n'est pas la même partie du graphe qui s'exécute, ce qui correspond à du conditionnement.

Et donc, le modèle choisi pour modéliser les systèmes réactifs est un hypergraphe conditionné infiniment itéré, ce qui correspond à un *graphe flot de données conditionné*.

2.3.1 Introduction

Nous cherchons à modéliser la répétition infinie de la séquence Entrées-Calculs-Sorties. Chaque séquence Entrées-Calcul-Sorties définit une *instance* ou encore *itération* du graphe de l'algorithme. Chaque séquence de ce graphe itératif correspond à un instant logique au sens des langages synchrones [6]. L'ensemble des itérations du graphe décrit un ensemble d'instants logiques. Cet ensemble totalement ordonné est appelé le temps logique. Il existe principalement deux modes de représentation de ce graphe :

- un mode où les instances sont distinctes (on en représente juste un certain nombre et on met des pointillés de part et d'autre pour indiquer la répétition) : le graphe est dit *défactorisé* [55]. Un exemple de ce mode de représentation est donné figure 2.3.12 page 52.
- un mode où toutes les instances sont superposées (on parle alors de graphe flot de données) : le graphe est dit *factorisé* [55]. Deux exemples de factorisation associés au

graphe défactorisé sont donnés figure 2.3.13 page 52. Ce mode factorisé a l'avantage d'éviter la lourdeur de la répétition.

Nous allons définir successivement les deux modes de représentation.

2.3.2 Formalisation

Graphe itératif défactorisé

G_{al} est un couple (O',D') où :

- O' est l'ensemble infini des opérations itérées, $O' = \{(o'^t_i)_{1\leq i\leq n'<\infty}\}_{t\in\mathbb{N}^*}$,

  Définition 41 *On appelle* itération *d'une opération o'_i, l'opération o'^t_i à l'instant logique t ou à l'itération t ($t \in \mathbb{N}^*$).*

- D' est l'ensemble des dépendances de données, $D' = \{(d'^{p,q}_i)_{1\leq i\leq k'<\infty}\}_{p,q\in\mathbb{N}^*}$.

  Définition 42 *On appelle itération d'une dépendance d'_i, la dépendance $d'^{p,q}_i$ ($p \neq q$), émise à l'itération p et reçue à l'itération q (avec $0 < p \leq q$).*

  - $d'^{p,p}_i$ désigne une dépendance entre opérations de la même itération. On note $D'^{p,p}$ l'ensemble des dépendances intra-itération appartenant à l'itération p et on note D'^1, l'ensemble des dépendances intra-itération.

    Remarque 23 *Pour alléger les notations, et quand il n'existe pas de confusion, $d'^{p,p}_i$ pourra être également noté d'_i.*

  - $d'^{p,q}_i$ désigne une dépendance entre opérations appartenant à des itérations différentes. $(q-p)$ désigne le nombre d'itérations qui séparent l'émetteur $\gamma^{-1}(d'^{p,q}_i)$ de l'ensemble des récepteurs $\gamma(d'^{p,q}_i)$ de la dépendance $d'^{p,q}_i$. On note $D''^{p,q}$, l'ensemble des dépendances engendrées par les données émises à l'itération p et reçues à l'itération q. On note D'^2, l'ensemble des dépendances inter-itération .

Définition 43 *On appelle motif du graphe, l'ensemble des opérations appartenant à la même itération et l'ensemble des dépendances de données entre ces opérations. Ainsi, si on appelle* motif(t)*, le motif de l'itération t, on a :*

$$\text{motif}(t) = \{(o'^t_i)_{1\leq i\leq n'<\infty},(d'^{t,t}_i)_{1\leq i\leq k'<\infty}\}$$

On appelle dépendance inter-motif associée à ce motif, l'ensemble des dépendances de données engendrées par des données émises avant l'itération considérée et reçues à cette itération, plus l'ensemble des données émises à l'itération considérée et reçues après cette même itération. C'est donc l'ensemble suivant :

$$(D'^{q,t} \cup D'^{t,p}) \quad q < t\ ,\ t < p$$

Relations associées au graphe infiniment itéré

- *Relation de précédence entre les itérations d'une même opération :* $<$

$$o'^{t}_{i} < o'^{t+1}_{i} < ... < o'^{t+m}_{i} < ...$$

avec $m \in \mathbb{N}$ et $m > 1$. L'ensemble $(\mathbb{N}^*, <)$ est un ensemble totalement ordonné et donc il existe une bijection entre l'ensemble des entiers naturels et les itérations d'une même opération.

Remarque 24 *Il y a une relation d'ordre total entre ces opérations bien qu'il n'y ait pas forcément d'arcs entre elles.*

- *Relation de dépendances intra-itération :* $\preceq$
$o'^{p}_{i} \preceq o'^{p}_{j} \quad \Leftrightarrow \exists d'_k \in D' \quad tq \quad \gamma^{-1}(d'_k) = o'_i \quad et \quad \gamma(d'_k) \supseteq \{o'_j\}$

Remarque 25 *Cette relation correspond à la relation $\preceq$ précédemment définie à la page 31.*

- *Relation de dépendances inter-itération : Int*
$o'^{p}_{i} \quad Int \quad o'^{q}_{j} \quad \Leftrightarrow \exists d'^{p,q}_{k} \in D' \quad tq \quad \gamma^{-1}(d'^{p,q}_{k}) = o'^{p}_{i} \quad et \quad \gamma(d'_k) \supseteq \{o'^{q}_{j}\} \quad p \neq q$

Remarque 26 *On ne fait pas la même classification que celle du langage SIGNAL [7]. L'ensemble des dépendances intra-itération et inter-itération sont définies dans SIGNAL comme des* dépendances opératoires.

Propriété 19 *Le graphe défactorisé est connexe s'il contient des dépendances inter-itération de type $d'^{p,p+1}_{\cdot}$.*

Un exemple de graphe itératif est donné figure 2.3.12. Cet exemple contient 5 itérations et des dépendances inter-itération sont mises en évidence.

Factorisation infinie

Nous allons présenter deux modèles de factorisation associés au graphe itéré.

- *Premier modèle de factorisation* Le premier modèle de factorisation de graphe itéré [86] consiste à étiqueter les arcs avec les distances de dépendances $q - p$ en ne notant que les distances strictement positives. Ainsi les dépendances $d'^{p,p}_{i}$ ne sont pas étiquetées alors que les dépendances $d'^{p,q}_{i}$ sont étiquetées par $q - p$.

  Un exemple de ce modèle de factorisation est donné figure 2.3.13 cas b).

  Une variante de cette factorisation est *le graphe uniforme* de Hanen et Munier [36]. Dans ce graphe, les arcs sont étiquetés par une longueur et une hauteur, la longueur

est égale à la durée d'exécution de l'opération émettrice et la hauteur est égale au nombre d'itérations séparant l'opération émettrice de l'opération réceptrice.

Remarque 27 *Nous n'allons pas formaliser cette factorisation car nous ne l'avons pas retenu. En effet, dans la deuxième partie de la thèse on étiquète le graphe avec les durées d'exécution donc pour éviter la confusion au niveau des étiquettes, nous avons choisi l'autre modèle de factorisation.*

- *Deuxième modèle de factorisation issu de [55]* L'ensemble des itérations $(o_i'^p)_{p\in\mathbb{N}^*}$ est remplacé par une seule opération o_i' factorisée.

$$(o_i'^p)_{p\in\mathbb{N}^*} \overset{factorisation}{\rightarrow} o_i'$$

L'ensemble des itérations $(d_i'^{p,p})_{p\in\mathbb{N}^*}$ d'une dépendance de donnée intra-itération d_i' est remplacé par une seule dépendance d_i' factorisée.

$$(d_i'^{p,p})_{p\in\mathbb{N}^*} \overset{factorisation}{\rightarrow} d_i'$$

Ce premier ensemble de dépendances est l'ensemble D'^1 défini à la page 48.

Etudions maintenant l'ensemble des dépendances $D'^{p,q}$. L'ensemble des itérations $(d_i'^{p,q})_{p,q\in\mathbb{N}^*}$ d'une dépendance de donnée inter-itération d_i' est remplacé par le graphe suivant :

$$(d_i'^{p,q})_{p,q\in\mathbb{N}^*} \overset{factorisation}{\rightarrow} (\gamma^{-1}(d_i'^{p,q}),\$_{q-p}(\gamma^{-1}(d'^{p,q})),\gamma(d'^{p,q}))$$

$\$_{q-p}(\gamma^{-1}(d'^{p,q}))$ est un sommet ajouté au graphe de l'algorithme et appelé *sommet retard* ou *opération de mémorisation*. Si on est à l'itération q, le sommet $\$_{q-p}(\gamma^{-1}(d'^{p,q}))$ représente le sommet $\gamma^{-1}(d'^{p,q})$ à l'itération p. Ce deuxième ensemble de dépendances correspond à l'ensemble D'^2 défini à la page 48.

Ainsi, G_{al} est un couple $(O'^{\$},D'^{\$})$ où :

- $O'^{\$}$ est l'ensemble fini des opérations factorisées $O' = \{o_i')_{1\leq i\leq n'<\infty}$ et des opérations de mémorisation $\$ = \bigcup_i \$_{q-p}(o_i')$, avec o_i' appartenant à O'.

$$O'^{\$} = O' \cup \$$$

- $D'^{\$}$ est l'ensemble des dépendances de données factorisées,

$$D'^{\$} = D'^1 \cup D'^2$$

Un exemple de ce deuxième modèle de factorisation est donné figure 2.3.13 cas a).

Relations associées au graphe flot de données

Associé à l'ensemble D'^1, on définit l'ordre partiel $\preceq$. Cet ordre correspond à l'ordre partiel de l'hypergraphe orienté définit précédemment.

On définit deux sous-ensembles D'^{21} et D'^{22} de l'ensemble des dépendances D'^2 qui forment une partition de l'ensemble D'^2.

$$\begin{aligned} &D'^2 = D'^{21} \cup D'^{22} \quad \text{avec} \\ &D'^{21} = \{d' \in D'^2 \text{ tq } \gamma^{-1}(d') \in O' \text{ et } \gamma(d') \in \$\} \\ &D'^{22} = \{d' \in D'^2 \text{ tq } \gamma^{-1}(d') \in \$ \text{ et } \gamma(d') \subseteq O'\} \end{aligned}$$

Associé à l'ensemble D'^{21}, on définit la relation $<_m$. Cette relation correspond à la mémorisation.

$o'_i <_m \$_k(o'_i)$ si et seulement si l'opération o'_i est mémorisée dans l'opération de mémorisation $\$_k(o'_i)$ avec k itérations d'écart.

Associé à l'ensemble D'^{22}, on définit la relation Int. Cette relation correspond à une relation entre opérations n'appartenant pas à la même itération.

$\$_k(o'_i)$ Int o'_j si et seulement si l'opération o'_j, à une itération donnée, dépend de l'opération o'_i exécutée k itérations avant.

Propriété 20 *Le graphe factorisé contient au moins autant de retards qu'il y a de circuits dans le graphe.*

Preuve : En effet, tout circuit est engendré par un retard mais tout retard n'engendre pas nécessairement de circuit.

2.3.3 Représentation graphique

Un exemple de graphe défactorisé est donné figure 2.3.12 et deux exemples de graphes factorisés sont donnés figure 2.3.13.

L'exemple de factorisation de la figure 2.3.13 correspond au graphe itératif défactorisé de la figure 2.3.12.

Comparaison avec d'autres langages graphiques

Simulink est l'interface bloc diagramme graphique de Matlab, tout comme Scicos qui est celle de Scilab [71]. Scicos [62] est un langage graphique permettant de saisir des boîtes. Chaque boîte est un *bloc Scicos* et l'ensemble de ces boîtes est connecté par des arcs appelés *transitions ordonnées*. L'ensemble des boîtes et des connexions forment un *schéma-blocs* ou *super-bloc*.

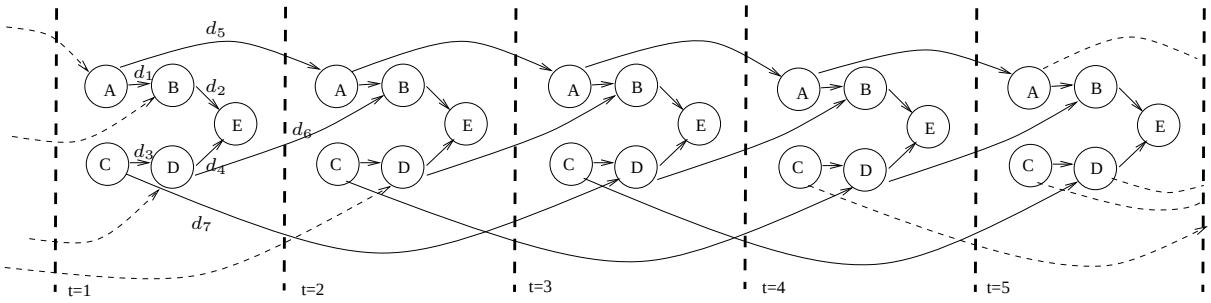
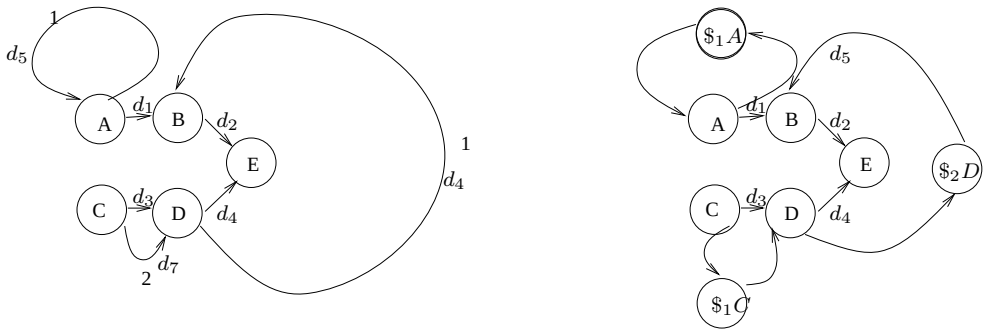


FIG. 2.3.12 – *Graphe itératif défactorisé*
$O' = \{A,B,C,D,E\}$,$D' = \{d_1,d_2,d_3,d_4,d_5^{i,i+1},d_6^{i,i+1},d_7^{i,i+2}\}$,$d_1 = (A,B)$,$d_2 = (B,E)$,$d_3 = (C,D)$,$d_4 = (D,E)$,$d_5^{i,i+1} = (A^i,A^{i+1})$,$d_6^{i,i+1} = (D^i,B^{i+1})$,$d_7^{i,i+2} = (C^i,D^{i+2})$



Factorisation a) Factorisation b)

FIG. 2.3.13 – *Graphe itératif factorisé*

Les transitions reçues par un bloc sont de deux types : type *Entrée* et type *Entrée Événement*. Les transitions émises par un bloc sont de deux types : type *Sortie* et type *Sortie Événement*.

Ainsi, on a le tableau d'équivalence donné table 2.3.4.

SynDEx	Scicos
graphe flot de données conditionné de l'algorithme	schéma-blocs
opération	bloc
dépendances de données simples	transitions ordonnées
condition d'activation	entrée événement

TAB. 2.3.4 – *Tableau des équivalences Scicos-SynDEx*

3. Modèle d'implantation

L'ensemble des implantations valides d'un graphe d'algorithme sur un graphe d'architecture est défini par la composition de trois relations : le *routage*, la *distribution* et *l'ordonnancement*.

Tout d'abord, nous présentons le routage qui modifie le graphe de l'architecture afin de le rendre complètement connecté. Ainsi, le routage crée tous les chemins possibles qui sont des combinaisons d'arêtes du graphe de l'architecture initial.

Ensuite, nous définissons la distribution. C'est une allocation spatiale du graphe de l'algorithme sur le graphe de l'architecture. La distribution est une relation qui réalise successivement deux partitions du graphe de l'algorithme. Dans un premier temps, l'algorithme est décomposé en sous-graphes disjoints sous la contrainte que le nombre de sous-graphes obtenus soit inférieur ou égal au nombre d'unités de calcul. Cela correspond à une partition sur le modèle encapsulé que l'on appelle *partitionnement*. Dans un deuxième temps, les dépendances de données entre opérations appartenant à des éléments de partition différents sont à leur tour distribuées selon le nombre de liaisons et d'unités de communication correspondantes. Ces dépendances donnent lieu à des transferts de données sur chacun des liens et unités de communication associées, composant la route reliant les deux processeurs. Chaque transfert de données sur un média est représenté par deux sommets spéciaux ajoutés au graphe initial de l'algorithme et appelés *opérations de communication*. Les opérations de communication associées à un transfert sont distribuées respectivement sur l'unité de communication du processeur émetteur du transfert et sur l'unité de communication du processeur récepteur. Ainsi chaque ensemble d'opérations de communication distribuées sur une unité de communication forme un élément de la deuxième partition du graphe de l'algorithme.

L'ordonnancement est une relation qui renforce, en ajoutant des arcs, appelés dépendances d'ordonnancement, chaque ordre partiel associé à chaque unité de calcul et à chaque média, et le transforme en un ordre total. Cet ordre total est une extension linéaire donc est compatible avec l'ordre partiel du graphe initial de l'algorithme.

En composant les trois relations routage, distribution et ordonnancement, on obtient l'ensemble des implantations valides associées à un couple algorithme, architecture.

A ce modèle d'implantation, on associe un seul instant logique correspondant à l'exécution du graphe d'algorithme sur le graphe de l'architecture et une relation d'ordre partiel,

notée $\trianglelefteq$, $o_i \trianglelefteq o_j$ si et seulement si l'opération o_i "est exécutée avant" l'opération o_j.

3.1 Routage

3.1.1 Introduction

Beaucoup d'heuristiques font l'hypothèse simplificatrice que l'architecture est complètement connectée. Ainsi, le but du routage est d'obtenir une architecture complètement connectée à partir d'une architecture qui n'a pas cette propriété. Cela conduit à définir des chemins dans le graphe de l'architecture, chacun d'eux étant une combinaison de liaisons si on considère le modèle encapsulé et une combinaison de média si on considère le modèle développé. Le coût d'un transfert de données entre opérations va dépendre du nombre de liaisons (ou de média) utilisées pour l'effectuer sur une route donnée. En ce sens, ce modèle de communication est plus précis que le modèle utilisé dans les PRAMs qui suppose que les transferts de données s'effectuent par une mémoire partagée qui est accessible en un temps infiniment petit. On suppose que les liaisons inter-processeur sont *half-duplex*, c'est-à-dire qu'une seule communication peut circuler à la fois à un instant donné entre deux processeurs donnés sur une liaison donnée. Lorsqu'un processeur a un message à transmettre à un autre processeur, il utilise une table de routage qui lui indique quel chemin le plus court prendre pour accéder à ce processeur en utilisant un ou éventuellement plusieurs processeurs intermédiaires. Le mode de communication décrit ici correspond au mode de communication par commutation de messages selon la terminologie de Rumeur dans [21]. Plus précisément les messages avancent dans le réseau vers leur destination en passant par les processeurs intermédiaires. A chaque étape, la liaison empruntée est aussitôt libérée. Cette technique est appelée *store and forward*.

Le routage est une relation qui s'applique au graphe de l'architecture encapsulée (P,H) ou au graphe de l'architecture développée (S,H_d). Bien que le modèle encapsulé ne soit pas utilisé par la suite, pour des raisons de clarté de présentation, nous allons définir le routage sur les deux modèles.

3.1.2 Formalisation

Routage du modèle encapsulé

Il conduit à deux types de routes : tout d'abord l'ensemble des liaisons qui existent sur le graphe initial et puis l'ensemble de toutes les combinaisons possibles de ces liaisons. On appelle R, l'ensemble de toutes ces routes.

Le chemin $r \in R$ qui connecte le processeur p_1 et le processeur p_n en utilisant $n-2$ processeurs intermédiaires $p_2,...,p_{n-2}$ est défini par : $r = (l_1,l_2,...,l_{n-1})$.

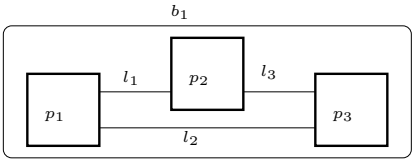


FIG. 3.1.1 – *Graphe de l'architecture encapsulé*

Si l'on considère l'exemple décrit dans la figure 3.1.1, l'ensemble des chemins R est défini par : $R = \{r_i\}_{1 \leq i \leq 7}$ avec $r_i = l_i$ pour $1 \leq i \leq 3$, $r_4 = b_1$, $r_5 = (l_1,l_2)$,$r_6 = (l_2,l_3)$ et $r_7 = (l_1,l_3)$. Pour communiquer par exemple entre p_1 et p_3, deux types de communication peuvent être utilisés : d'une part la route directe l_2, et d'autre part, la route qui comprend les liaisons l_1 et l_3. Par exemple, si p_1 a deux communications à envoyer à p_3, au lieu de les envoyer séquentiellement, l'une d'elle peut être envoyée par la route $r_2 = l_2$ et l'autre par la route $r_7 = (l_1,l_3)$.

Propriété 21 $H \subseteq R$

Routage du modèle développé

Il conduit à deux types de routes : tout d'abord l'ensemble des média qui existent sur le graphe initial et puis l'ensemble de toutes les combinaisons possibles de ces média. On appelle R_d, l'ensemble de toutes ces routes.

Le chemin $r \in R_d$ qui connecte l'unité de calcul du processeur p_I et l'unité de calcul du processeur p_{II} en utilisant n processeurs intermédiaires $p_1,...,p_n$ est défini par : $r = (m_I,m_1,...,m_n)$, où m_i désigne le média associé à la liaison h_i.

Si on considère l'exemple de la figure 3.1.2, l'ensemble des média est $m_1 = \{S_{1,com^2},l_1,S_{2,com^1}\}$, $m_2 = \{S_{3,com^1},l_2,S_{1,com^3}\}$, $m_3 = \{S_{2,com^3},l_3,S_{3,com^2}\}$, $m_4 = \{S_{1,com^{23}},S_{2,com^{13}},S_{3,com^{12}},b_1\}$. L'ensemble R_d de la figure 3.1.2 est défini par :$R_d = \{r_i\}_{1 \leq i \leq 7}$, avec $r_i = m_i$ pour $1 \leq i \leq 4$, $r_5 = (m_1,m_2)$, $r_6 = (m_1,m_3)$ et $r_7 = (m_2,m_3)$.

Ainsi, comme dans le cas précédent, pour communiquer entre deux processeurs, on peut soit utiliser un seul média qui correspond à la route directe, soit utiliser une combinaison de média.

Propriété 22 $H_d \subseteq R_d$

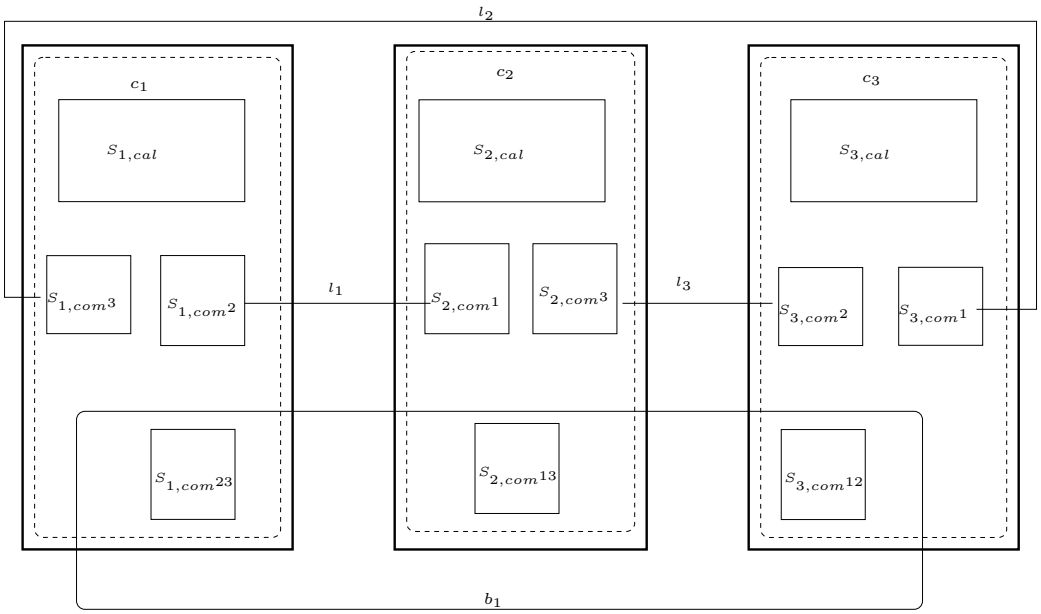


FIG. 3.1.2 – *Graphe de l'architecture développé*

Le graphe routé associé au graphe (P,H) (resp. (S,H_d)) est un graphe (P,R) (resp. (S,R_d)) où :

- l'ensemble des sommets est l'ensemble des sommets du graphe initial,
- l'ensemble des arêtes R (resp. R_d) est composé de l'ensemble initial des arêtes H (resp. H_d), plus l'ensemble des combinaisons possibles des arêtes de l'ensemble H (resp. H_d). On suppose que le nombre de routes R (resp. R_d) est égal à n_R (resp. n_{R_d}).

On appelle $\mathcal{R}_{routage}$, la relation qui associe, à un graphe d'architecture donné, le graphe d'architecture routé correspondant.

3.2 Distribution

3.2.1 Introduction

Ici, on suppose que le nombre de processeurs est inférieur ou égal au nombre d'opérations de calcul du graphe de l'algorithme. On appelle *distribution*, l'allocation spatiale du graphe de l'algorithme au graphe de l'architecture. La distribution correspond au placement dans la théorie des graphes.

La distribution est définie sur les deux modèles d'architecture définis au chapitre 1.

Sur le modèle encapsulé, c'est une distribution gros grain, appelée *partitionnement*, où les opérations du graphe de l'algorithme sont distribuées sur les processeurs de l'architecture

identifiés à leur unité de calcul. Les unités de communication sont ignorées.

Sur le modèle développé, la distribution est plus précise. Elle correspond à un partitionnement des opérations sur les unités de calcul et à une distribution, appelée *communication*, qui distribue les dépendances de données entre opérations distribuées sur des unités de calcul différentes, sur les routes du graphe de l'architecture construite lors du routage. La *communication* permet de détailler les communications inter-processeur.

3.2.2 Partitionnement

Introduction

La connaissance du graphe de l'architecture guide le partitionnement de l'algorithme, en effet, le cardinal de la partition doit être inférieur ou égal au nombre de processeurs.

Le partitionnement consiste à décomposer le graphe de l'algorithme en sous-graphes disjoints sous la contrainte que le nombre de sous-graphes soit inférieur ou égal au nombre de processeurs et d'associer chacun de ces sous-graphes à un processeur différent. On réalise donc une partition des sommets du graphe de l'algorithme. Par conséquent, le partitionnement conduit à un *onto mapping*, au lieu de l'habituel *one-to-one mapping* [14], où une seule opération est distribuée sur chaque processeur.

Formalisation

Soit Π', l'application qui, à chaque opération de calcul, associe le processeur sur lequel elle est distribuée.

$$\boxed{\begin{array}{llll} \Pi' : & O' & \rightarrow & P \\ & o'_i & \mapsto & \Pi'(o'_i) = p_j \end{array}}$$

Soit Π'^{-1}, l'application réciproque de Π', qui associe à chaque processeur, l'ensemble des opérations de calcul distribuées sur ce processeur.

$$\boxed{\begin{array}{llll} \Pi'^{-1} : & P & \rightarrow & \mathcal{P}(O') \\ & p_j & \mapsto & \Pi'^{-1}(p_j) = \{o'_i \in O' \ / \ \Pi'(o'_i) = p_j\} \end{array}}$$

Remarque 28 *Les propriétés associées aux applications Π' et Π'^{-1} sont étudiées dans la deuxième partie de la thèse, au chapitre Caractérisation page 85.*

Ainsi, $\Pi'^{-1}(p)$ correspond à l'ensemble des opérations distribuées sur le processeur p, on notera encore O'_p cet ensemble d'opérations.

$$O'_p = \bigcup_{i=1}^{n'_p} o'_i \text{ avec } o'_i \in O', \forall p \in P \text{ et } \sum_{p=1}^{\text{Card }(P)} n'_p = \text{Card}(O') = n'$$

$$\begin{cases} O'_{p1} \cap O'_{p2} = \emptyset \quad \forall\, p_1,p_2 \in P \\ \bigcup_{p \in P} O'_p = O' \end{cases}$$

Propriété 23 *L'ensemble des $(O'_p)_{p \in P}$ forment une partition de l'ensemble des opérations O'.* En effet, par construction : une opération est distribuée sur un et un seul processeur.

Définition 44 *$D'_p \subseteq O'_p \times \mathcal{P}(O'_p) \quad \forall p \in P$, est l'ensemble des dépendances de données entre opérations distribuées sur le processeur p. Ainsi, c'est un ensemble de dépendances de données dit* intra-processeur. *On note D'_P, l'ensemble des dépendances intra-processeur de G_{al}. On a : $D'_P \subseteq D'$.*

Remarque 29 *Une dépendance de donnée d'_i appartenant à D' a un unique émetteur et au moins un récepteur. Par exemple, si d'_i relie un émetteur o'_e et deux récepteurs o'_{r_1} et o'_{r_2}, il a la forme $(o'_e,\{o'_{r_1},o'_{r_2}\})$. Ainsi l'ensemble D'_p est inclus dans le produit cartésien $O'_p \times \mathcal{P}(O'_p)$.*

$D'_p = \bigcup_i d'_i \ \forall p \in P$, où $\gamma(d'_i)$ et $\gamma^{-1}(d'_i)$ distribuées sur le processeur p.

$\forall\, d'_i \in D'_p, \forall p \in P,$ si $o'_j \in \gamma(d'_i)$ alors $\gamma^{-1}(d'_i) \preceq_p o'_j$ avec $\preceq_p$= "est exécuté avant sur le même processeur". Ainsi, à chaque processeur p, on associe l'ordre partiel $\preceq_p$ sur l'ensemble des opérations O'_p du processeur p et cet ordre partiel est induit par les dépendances de données D'_p.

Remarque 30 *L'ensemble D'_p des dépendances sur le processeur p peut être vide, dans ce cas il n'y a pas de relation d'ordre entre les opérations distribuées sur ce processeur.*

$$\begin{cases} D'_{p_1} \cap D'_{p_2} = \emptyset \quad \forall\, p_1,p_2 \in P \\ \bigcup_{p \in P} D'_p = D'_P \end{cases}$$

D'_P définit un ordre partiel $\preceq_P$ sur l'ensemble des opérations O'.

Remarque 31 *L'ordre partiel $\preceq_P$ est inclus dans l'ordre partiel initial $\preceq$ puisque $\preceq_P$ est associé à l'ensemble D'_P qui est inclus dans l'ensemble D' des dépendances.*

Définition 45 *L'ensemble D'_r est l'ensemble des dépendances de données entre opérations dépendantes et distribuées sur des processeurs différents. Chacune de ces dépendances va donner lieu à un transfert de données sur chacune des liaisons composant la route r. Ce*

transfert de données sera modélisé lors de la distribution du graphe de l'algorithme sur le modèle développé de l'architecture.

$$D'_r = \bigcup_{p_k \in r} (O'_{p_k} \times \mathcal{P}(\bigcup_{p_l \in r, p_l \neq p_k} O'_{p_l}))$$

$$D'_r \subseteq D'$$

On note D'_R l'ensemble des dépendances entre opérations dépendantes et distribuées sur des processeurs différents.

Remarque 32 *Plusieurs dépendances d'_i appartenant à D'_r peuvent être distribuées sur la même route r de l'ensemble R. Ces dépendances seront séquentialisées lors de l'ordonnancement.*

$D'_r = \bigcup_i d'_i \;\; \forall r \in R$, avec $\gamma^{-1}(d'_i)$ distribué sur p un des processeurs composant la route r, et $\exists o'_j, o'_k, ... \in \gamma(d'_i)$ telles que o'_j distribuée sur $p_j \neq p$ et o'_k distribuée sur $p_k \neq p$.

$$d'_i = (o'_{p_{ik}}, \bigcup_{p_l \neq p_k, p_l \in r} o'_{p_{il}}), p_{ik} \in r$$

$\forall d'_i \in D'_r, \forall r \in R$, si $o'_j \in \gamma(d'_i)$ alors $\gamma^{-1}(d'_i) \preceq_r o'_j$ avec $\preceq_r =$ "est exécuté avant (sur des processeurs différents)".

$$\begin{cases} D'_{r_1} \cap D'_{r_2} = \emptyset \quad \forall\, r_1, r_2 \in R \\ \bigcup_{r \in R} D'_r = D'_R \end{cases}$$

D'_R définit un ordre partiel $\preceq_R$ sur l'ensemble des opérations O'.

Remarque 33 *L'ordre partiel $\preceq_R$ est inclus dans l'ordre partiel initial $\preceq$ puisque $\preceq_R$ est associé à l'ensemble D'_R qui est inclus dans l'ensemble D' des dépendances.*

$$\begin{cases} (\bigcup_{p \in P} D'_p) \bigcup (\bigcup_{r \in R} D'_r) = D' \\ D'_p \cap D'_r = \emptyset \;\; \forall p, r \in P \times R \end{cases}$$

Propriété 24 *Les arcs D'_P et D'_R forment une partition l'ensemble D.*

Chaque sous-graphe généré (O'_p, D'_p) est appelé région atomique par Zwiers et Janssen dans [89]. Cela permet d'encapsuler chaque sous-ensemble de sommets du graphe initial de l'algorithme en une région atomique. Les arcs inter-partition D'_R, c'est-à-dire les arcs dont l'émetteur et les récepteurs n'appartiennent pas à la même région, vont être distribués sur les différentes routes construites lors du routage de graphe de l'architecture.

Ainsi, un graphe G_{al} distribué sur un graphe encapsulé d'architecture routé G'_{ar} est un graphe $G_{partR}(.)$ où :

– l'ensemble des sommets est l'ensemble des sous-graphes $\bigcup_{p \in P} (O'_p, D'_p)$,

– l'ensemble des arcs est l'ensemble des dépendances D'_R.

Ce qui peut encore s'écrire :

$$G_{partR}(.) = (\bigcup_{p \in P} (O'_p, D'_p), \bigcup_{r \in R} D'_r)$$

Propriété 25 *Le partitionnement ne modifie ni le nombre de sommets ni le nombre d'arcs du graphe de l'algorithme : ainsi, les deux graphes avant et après partitionnement ont la même fonctionnalité.* En effet, on a : $D' = D'_P \cup D'_R$ et $O' = \bigcup_{p \in P} O'_p$. Le partitionnement est donc simplement une réécriture qui consiste à structurer le graphe en régions atomiques. Le nombre de décompositions en régions atomiques est fini, car le graphe de l'algorithme et le graphe de l'architecture possède un nombre fini de sommets et d'arcs.

Remarque 34 *L'ordre partiel initial $\preceq$ du graphe de l'algorithme est égal à l'union des ordres partiels du graphe distribué.*

$$\preceq = (\bigcup_{p \in P} \preceq_p) \cup (\bigcup_{r \in R} \preceq_r)$$

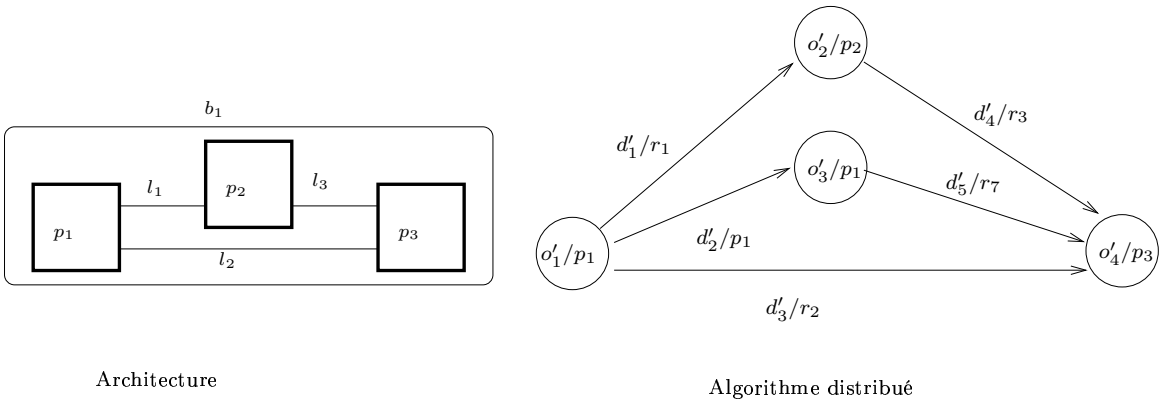


FIG. 3.2.3 – *Un partitionnement*

Si l'on considère l'exemple de la figure 3.2.3, on a effectué une partition de cardinal 3.

Dépendances intra-partition :

$O'_{p_1} = \{o'_1, o'_3\} \Rightarrow D'_{p_1} = \{d'_2\}$,

$O'_{p_2} = \{o'_2\} \Rightarrow D'_{p_2} = \emptyset$ car il y a une seule opération sur le processeur 2,

$O'_{p_3} = \{o'_4\} \Rightarrow D'_{p_3} = \emptyset$ pour la même raison.

Dépendances inter-partition : Une dépendance inter-partition :

d'_1 entre O'_{p_1} et O'_{p_2} qui est distribuée sur r_1, c'est-à-dire $D_{r_1} = D_{l_1} = \{d'_1\}$,

d'_4 entre O'_{p_2} et O'_{p_3} qui est distribuée sur r_2, c'est-à-dire $D'_{r_2} = D_{l_2} = \{d'_4\}$.

Deux dépendances inter-partition d'_3 et d'_5 entre p_1 et p_3 . Il existe plusieurs solutions pour distribuer $\{d'_3\}$ et $\{d'_5\}$. En voici deux :

(1) $\{d'_3\}$ et $\{d'_5\}$ sont exécutées en parallèle sur les chemins r_2 et r_7, c'est-à-dire $D'_{r_2} = D'_{l_2} = \{d'_3\}$,$D'_{r_7} = D'_{(l_1,l_3)} = \{d'_5\}$.

(2) $\{d'_3\}$ et $\{d'_5\}$ sont séquentialisées sur la route r_2, c'est-à-dire $D'_{r_2} = \{d'_3,d'_5\}$,$D'_{r_7} = \emptyset$

Propriété 26 *Étant donné qu'aucun arc n'a été supprimé ou ajouté durant le partitionnement, toutes les dépendances de données sont inchangées par rapport au graphe initial de l'algorithme. Ainsi, l'ordre partiel associé à ces dépendances est lui aussi inchangé. Ainsi, le partitionnement de l'algorithme est garanti sans inter-blocage par définition de l'ordre partiel associé à l'algorithme.* En effet, les arcs D'_P et D'_R forment une partition de l'ensemble des dépendances D'.

Propriété 27 *Soit $(G_{al},G'_{ar}) = ((O',D'),(P,R))$, un graphe d'algorithme et un graphe d'architecture routé. Alors il est possible de construire plusieurs graphes partitionnés $(G_{partR},G'_{ar})(.)$ associés à (G_{al},G'_{ar}), mais un nombre fini N borné par $(n_P)^{n'} * (n_R)^{k'}$. n_P est le nombre de processeurs, n' est le nombre d'opérations, n_R est le nombre de routes et k' est le nombre de dépendances.*

Preuve : Pour chaque opération o' de O', on a n_P distributions possibles, donc l'ensemble des distributions possibles des opérations sur les processeurs est égal à $(n_P)^{n'}$.

On considère une distribution donnée des opérations. Les arcs du graphe de l'algorithme inter-partition doivent être distribués sur les routes. Le nombre d'arcs à distribuer sur les routes est inférieur ou égal au nombre k' d'arcs du graphe initial. Pour chacun de ces arcs, on a un nombre de distributions inférieur ou égal au nombre n_R de routes du graphe. En effet, un arc inter-partition ne peut être distribué que sur les routes qui joignent les processeurs sur lesquels sont distribués les opérations émettrice et réceptrice(s) de l'arc. Donc, si on fait l'inventaire de toutes les distributions possibles des arcs sur les routes pour une distribution d'opérations donnée, on a un nombre de distributions inférieur ou égal à $(n_R)^{k'}$.

Si on calcule maintenant le nombre de distributions des opérations sur les processeurs et de distributions des arcs sur les liaisons, on a un nombre inférieur ou égal à $(n_P)^{n'} * (n_R)^{k'}$.

Remarque 35 *$(G_{al},G'_{ar})\mathcal{R}_{par}(G_{partR}(i),G'_{ar})$ avec $1 \leq i \leq n$ où n est le nombre de graphes partitionnés et $\mathcal{R}_{par}$ est la relation " a comme graphe partitionné ".*

Propriété 28 *Le partitionnement est une relation $\mathcal{R}_{par}$ de l'ensemble des couples de graphes orientés et non orientés vers l'ensemble des couples de graphes orientés et non orientés.* En

effet, le graphe de l'architecture n'est pas modifié et le nombre de décompositions du graphe de l'algorithme en régions atomiques (cf : remarque 25 page 62) est fini.

Nous allons maintenant présenter la relation *communication*, qui correspond à la distribution du graphe de l'algorithme sur le graphe de l'architecture développée. C'est un *one-to-one mapping* des régions atomiques sur les unités de calcul et un *one-to-one mapping* des communications entre régions atomiques sur les routes. Cela va permettre d'ordonnancer les communications inter-processeur sur les routes, comme décrit dans la section 3.3. Il est très important de prendre en compte aussi précisément que possible les communications, car dans la plupart des applications parallèles, les communications inter-processeur sont très coûteuses et donc détériorent le *speed-up*, c'est-à-dire le facteur d'accélération de la durée d'exécution d'une application quand on passe d'une architecture monoprocesseur à une architecture multiprocesseur. Si le graphe de l'architecture n'est pas suffisamment détaillé, l'optimisation de l'implantation va être de moins bonne qualité.

3.2.3 Communication

Introduction

La relation *communication* décrit les communications inter-processeur. Chaque élément de la partition du graphe de l'algorithme, distribué durant le partitionnement sur le processeur p du modèle encapsulé, va être distribué durant la communication à l'unité de calcul $S_{p,cal}$ du processeur p. Les arcs intra-partition D'_p, c'est-à-dire les arcs distribués sur le processeur p sont associés au bus RAM c_p intra-processeur défini dans le modèle d'architecture. Les dépendances de données D'_R entre régions atomiques du graphe de l'algorithme sont distribuées sur les routes du graphe de l'architecture construites lors du routage sur le modèle développé (cf : section 3.1.2). Plus précisément, lorsque deux opérations dépendantes sont distribuées sur deux unités de calcul différentes, la dépendance de données va donner lieu à un *transfert de données* sur chacun des média composant la route reliant les deux unités de calcul. L'ensemble des transferts de données associés à une même dépendance de données définit une *communication inter-processeur*, encore appelée *communication* entre deux opérations dépendantes.

Chaque transfert de données sur un média est représenté par deux sommets spéciaux ajoutés au graphe de l'algorithme et appelés *opérations de communication*. (On verra par la suite ce qu'est une telle opération.) Pour chaque dépendance de données, on insère entre les deux opérations dépendantes autant d'opérations de communication qu'il y a d'unités de communication utilisées pour ce transfert sur les média qui composent la route.

Formalisation

Soit O'', l'ensemble des opérations de communication ajoutées au graphe de l'algorithme.

$$O'' = \{o''_k\}_{1 \leq k \leq n''}$$

Soit O, l'ensemble des opérations de calcul et de communication du graphe de l'algorithme.

$$O = O' \cup O''$$

Soit O''_p, l'ensemble des opérations de communication distribuées sur la(les) unité(s) de communication du processeur p.

$$O''_p = \bigcup_{i=1}^{n''_p} o''_i \text{ avec } o''_i \in O'', \forall p \in P \text{ et } \sum_{p=1}^{\text{Card }(P)} n''_p = \text{Card}(O'') = n''$$

$$\begin{cases} O''_{p1} \cap O''_{p2} = \emptyset & \forall\, p_1,p_2 \in P \\ \bigcup_{p \in P} O''_p = O'' \end{cases}$$

Soit D'', l'ensemble des arcs entre opérations de communication distribuées sur des unités de communication de processeurs différents. Donc l'ensemble D'' sera distribué sur des liaisons inter-processeur.

$$D''_{p,q} \subseteq O''_p \times O''_q \; p \neq q$$

$$D'' \subseteq \bigcup_{p,q,p \neq q} O''_p \times O''_q$$

$$D'' = \bigcup_i d''_i = \bigcup_{p,q} D''_{p,q}$$

Soit $\preceq_h$, l'ordre partiel induit par l'ensemble D'' et soit $\preceq^*_h$, la fermeture transitive associée à cet ordre partiel.

$$\preceq_h = \{(o''_i,o''_j) \in O'' \times O'' \text{ tq } \exists d''_k \in D'' \text{ avec } o''_i = \gamma^{-1}(d''_k) \text{ et } \{o''_j\} = \gamma(d''_k)\}$$

Ceci peut encore s'écrire de la manière suivante :

$$\forall d''_k \in D'', \text{ si } \{o''_j\} = \gamma(d''_k) \text{ alors } \gamma^{-1}(d''_k) \preceq_h o''_j$$

Soit D^*_P, l'ensemble des arcs entre opérations (de calcul ou de communication) distribuées sur le même processeur. D^*_p sera distribué sur le RAM du processeur p.

$$D^*_p \subseteq (O'_p \times O''_p) \cup (O''_p \times O''_p) \cup (O''_p \times O'_p)$$

$$D^*_P = \bigcup_{p \in P} D^*_p$$

Soit $\preceq_c$ l'ordre partiel induit par l'ensemble D_P^* et soit $\preceq_c^*$ la fermeture transitive associée à cet ordre partiel.

$$\preceq_c = \{(o_i,o_j) \in O \times O \text{ tq } \exists d_k^* \in D_P^* \text{ avec } o_i = \gamma^{-1}(d_k^*) \text{ et } \{o_j\} = \gamma(d_k^*)\}$$

Ceci peut encore s'écrire de la manière suivante :

$$\forall d_k^* \in D_P^*, \text{ si } \{o_j\} = \gamma(d_k^*) \text{ alors } \gamma^{-1}(d_k^*) \preceq_c o_j$$

Soit d' appartenant à l'ensemble des dépendances D'_R. Chaque couple (o'_i,o'_j) tel que $o'_i = \gamma^{-1}(d')$ et $\{o'_j\} \subseteq \gamma(d')$ va être remplacé lors de la distribution par un graphe linéaire. On suppose que (o'_i,o'_j) est distribuée sur la route $r = (m_1,...,m_n)$. Lors de la distribution, la dépendance (o'_i,o'_j) est remplacée par le graphe linéaire suivant, sachant qu'à chaque média m_k, on associe les opérations de communication o''_{k1} et $o''_{(k+1)1}$ distribuées chacune sur les unités de communication des processeurs reliés par le média :

$(o'_i,o'_j) \overset{communication}{\longrightarrow} \left(d^*{}_1,o''_{11},d''_1,o''_{21},d^*{}_2,o''_{22},d''_2,...,d''_{n-1},o''_{n1},d^*{}_n\right)$ avec :

$$\begin{array}{lcl}
o'_i = \gamma^{-1}(d_1^*) = \gamma^{-1}(d') & \text{et} & \gamma(d_1^*) = o''_{11} = \gamma^{-1}(d''_1) \\
... & & ... \\
\gamma^{-1}(d_i^*) = o''_{i1} = \gamma(d''_{i-1}) & \text{et} & \gamma(d_i^*) = o''_{i2} = \gamma^{-1}(d''_i) \\
... & & ... \\
\gamma^{-1}(d_n^*) = o''_{n1} & \text{et} & \gamma(d_n^*) = o'_j \subseteq \gamma(d')
\end{array}$$

De même, on définit la transformation inverse $communication^{-1}$ par :

$(d^*{}_1,o''_{11},d''_1,o''_{21},d^*{}_2,o''_{22},d''_2,...,d''_{n-1},o''_{n1},d^*{}_n) \overset{communication^{-1}}{\longrightarrow} \left(o'_i,o'_j\right)$

L'arc d_k^* appartient à l'ensemble D_P^* et est distribué sur le bus RAM intra-processeur c_k.

L'arc d''_k appartient à l'ensemble D'' et est distribué sur la liaison inter-processeur h_k.

Remarque 36 *Les ensembles D'_P et D'' remplacent l'ensemble des dépendances D'_R.*

Ainsi, un graphe G_{al} distribué sur un graphe développé d'architecture G'_{ar} est un graphe $G_{comR}(.)$ où :

- l'ensemble des opérations O est l'ensemble des opérations de calcul O' et de communication O'', $|O| = n = n' + n''$
- l'ensemble des dépendances D est l'ensemble des dépendances D'_P précédemment construit, plus les ensembles D_P^* et D'' construits lors de la distribution sur le modèle développé. L'ensemble D'_P est un ensemble de dépendances uniquement entre opérations de calcul. L'ensemble D_P^* est un ensemble de dépendances intra-processeur entre opérations de calcul et de communication, ou bien entre opérations de communication uniquement. L'ensemble D'' est un ensemble de dépendances inter-processeur entre opérations de communication uniquement.

Ce qui peut encore s'écrire :

$$G_{comR}(.) = (O' \cup O'', D'_P \cup D^*_P \cup D'')$$

Propriété 29 *Chaque opération de communication $o'' \in O''$ a un et un seul successeur o_i appartenant à O et un et un seul prédécesseur o_j appartenant à O.*

$$\Gamma(o''_k) = \{o_i\} \text{ et } \Gamma^{-1}(o''_k) = \{o_j\}$$

Preuve : graphe linéaire.

Remarque 37 *Les prédécesseur et successeur d'une opération (de calcul ou de communication) sont soit de type calcul, soit de type communication.*

Propriété 30 *Entre deux opérations de communication situées sur une même unité de communication, il n'y a aucune dépendance de données directe, il n'y a que des dépendances transitives.*

Preuve : en effet, chaque opération de communication a un et un seul émetteur et un et un seul récepteur. Le prédécesseur (resp. successeur) est soit une opération de calcul, soit une opération de communication. Si c'est une opération de calcul, la dépendance entre les deux ne peut figurer sur l'unité de communication. Si c'est une opération de communication, elle est obligatoirement distribuée sur une autre unité de communication, sinon les opérations seraient confondues car, par construction, on distribue les opérations de communication pour une même dépendance de données sur des unités de communication différentes.

Définition 46 *Chaque opération de communication est une fonction identité.*

Remarque 38 *Une opération de communication est conditionnée par le même booléen que celui qui conditionne l'opération de calcul qui la précède. Ainsi, tout comme une opération de calcul, une opération de communication peut être conditionnée ou non.*

Si l'on considère l'exemple de la figure 3.2.4, on a 4 dépendances distribuées sur 4 routes :

$D'_{r_1} = \{d'_1\} = \{d^*_{11}, o''_1, d''_{11}, o''_2, d^*_{12}\}$
$D'_{r_2} = \{d'_3\} = \{d^*_{31}, o''_9, d''_{31}, o''_{10}, d^*_{32}\}$
$D'_{r_3} = \{d'_4\} = \{d^*_{41}, o''_3, d''_{41}, o''_4, d^*_{42}\}$
$D'_{r_7} = \{d'_5\} = \{d^*_{51}, o''_5, d''_{51}, o''_6, d^*_{52}, o''_7, d''_{52}, o''_8, d^*_{53}\}$

On a ajouté 10 opérations de communication représentées en grisé sur la figure. Chacune des dépendances d'_1, d'_3 et d'_4 est distribuée sur un seul média (resp. m_1, m_2 et m_3) donc deux opérations de communication sont ajoutées pour chacune de ces dépendances. En revanche, la dépendance d'_5 est distribuée sur la route r_7 composée de deux média m_1 et m_3, chacun des deux média supporte un transfert de données qui est modélisé par deux opérations

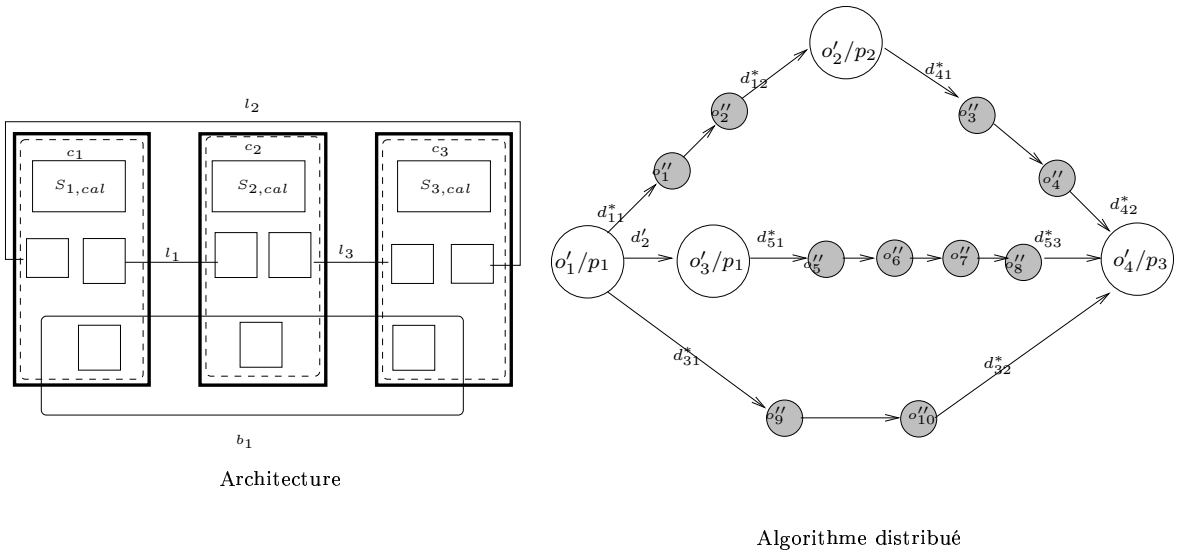


FIG. 3.2.4 – *Une communication.*

de communication, et donc la distribution de d'_5 engendre l'ajout de quatre opérations de communication.

Propriété 31 *L'ordre partiel $\preceq_P$ est inchangé durant la communication.*
Preuve : $\preceq_P$ est associé à l'ensemble D'_P des dépendances qui est inchangé durant la communication.

Propriété 32 *La relation de communication ne modifie pas l'ordre partiel du graphe de l'algorithme associé aux opérations de calcul du graphe.*

Preuve : soit $\preceq_{ch}=\preceq_c \cup \preceq_h$ l'union des ordres partiels construits lors de la communication, et soit $\preceq^*_{ch}$, la fermeture transitive associée à l'ordre partiel $\preceq_{ch}$. Supposons que $(o'_i,o'_j) \subseteq \preceq_R$. Par définition de la communication, (o'_i,o'_j) va être remplacé par un graphe linéaire tel que : $o'_i \preceq_{ch} o''_{11} \preceq_{ch} ... \preceq_{ch} o''_{i1} \preceq_{ch} o''_{i2} \preceq_{ch} ... \preceq_{ch} o''_{n1} \preceq_{ch} o'_j$. Alors, par transitivité, $o'_i \preceq_R o'_j \Leftrightarrow o'_i \preceq^*_{ch} o'_j$. Ainsi, si on appelle $\preceq^*_{ch/O'\times O'}$, la fermeture transitive définie précédemment et restreinte à l'ensemble des opérations de calcul, on a :

$$\preceq^*_{ch/O'\times O'}= \{(o'_i,o'_j) \in O' \times O' \ tq \ o'_i \preceq^*_{ch} o'_j\}$$

$$\preceq=\preceq_P \cup \preceq^*_{ch/O'\times O'}$$

Propriété 33 *L'ordre partiel à l'issue de la communication contient l'ordre partiel initial.*

$$\preceq \subseteq (\preceq_P \cup \preceq_c \cup \preceq_h)^*$$

Preuve : L'ordre partiel initial est égal à l'union de deux ordres partiels $\preceq_P$ et $\preceq_R$.

$$\preceq = \preceq_P \cup \preceq_R$$

L'ordre partiel $\preceq_P$ est inchangé durant la communication (cf. propriété 31).

Étudions maintenant l'ensemble des dépendances D'_R qui induit l'ordre partiel $\preceq_R$. Soit $\preceq_{Pch} = \preceq_P \cup \preceq_c \cup \preceq_h$, l'ordre partiel associé au graphe de l'algorithme à l'issue de la communication. Soit $d' \in D'_R$ la dépendance, qui, par construction est remplacée par un graphe linéaire $(d^*{}_1, o''_{11}, d''_1, o''_{21}, d^*{}_2, o''_{22}, d''_2, ..., d''_{n-1}, o''_{n1}, d^*{}_n)$. Ainsi

$$\gamma^{-1}(d') \preceq_R \gamma(d') \Rightarrow \begin{cases} \gamma^{-1}(d') \preceq_{Pch} \gamma(d^*_1) \\ \gamma(d^*_1) \preceq_{Pch} \gamma(d''_1) \\ \quad ... \\ \gamma^{-1}(d^*_n) \preceq_{Pch} \gamma(d'_i) \end{cases} \Rightarrow \gamma^{-1}(d') \preceq^*_{Pch} \gamma(d')$$

Ainsi, l'ordre partiel $\preceq_R$ est inclus dans l'ordre partiel $\preceq^*_{Pch}$ de l'algorithme distribué.

$$\Rightarrow (\preceq_P \cup \preceq_R) \subseteq \preceq^*_{Pch} \qquad \Leftrightarrow \qquad \preceq \subseteq \preceq^*_{Pch}$$

et donc l'ordre partiel initial est contenu dans l'ordre partiel construit lors de la communication.

Remarque 39 *Soit* $(G_{al}, G'_{ar}) = ((O', D'), (S, R_d))$, *un graphe d'algorithme et un graphe routé d'architecture. Il est possible de construire plusieurs, mais un nombre fini* n *de graphes distribués* $(G_{comR}, G'_{ar})(.)$ *sur le modèle développé de l'architecture, associés à* (G_{al}, G'_{ar}), *Ceci peut encore s'écrire :* $(G_{al}, G'_{ar})\, \mathcal{R}_{com}\, (G_{comR}(i), G'_{ar})$ *avec* $1 \leq i \leq n$ *où* n *est le nombre de graphes distribués sur le modèle développé et* $\mathcal{R}_{com}$ *est la relation "a pour graphe distribué sur le modèle développé".*

Propriété 34 *La communication est une relation* $\mathcal{R}_{com}$ *de l'ensemble des couples (graphe orienté, graphe non orienté) vers l'ensemble des couples de (graphe orienté, graphe non orienté).* Le graphe de l'architecture est inchangé et le nombre de décompositions du graphe de l'algorithme en régions atomiques (cf : remarque 25) est fini.

Distribution

Pour résumer, la distribution peut être vue :

- relativement au modèle d'architecture encapsulé comme le résultat du partitionnement, ce qui conduit à l'ensemble G_{partR}, ou
- comme le partitionnement suivi de la communication, ce qui conduit à l'ensemble G_{comR}. La distribution sur le modèle développé est plus précise car elle permet de modéliser les transferts de données sur les média.

On appelle G_{dR} le graphe de l'algorithme distribué sur le graphe de l'architecture routé et on appelle $\mathcal{R}_{dis}$, la relation qui associe à un couple (graphe d'algorithme, graphe d'architecture) l'ensemble des graphes (graphe d'algorithme distribué, graphe d'architecture routé). La relation $\mathcal{R}_{dis}$ est égale à la relation $\mathcal{R}_{par}$ si le modèle d'architecture est le modèle encapsulé et est égale à $\mathcal{R}_{com}$ si le modèle d'architecture est le modèle développé.

3.3 Ordonnancement

3.3.1 Introduction

On appelle *ordonnancement*, l'allocation temporelle du graphe distribué sur le modèle développé du graphe de l'architecture.

Un ordre partiel est associé à chaque sous-graphe obtenu durant la distribution. Chaque sous-graphe est distribué soit sur une unité de calcul soit sur une unité de communication. Comme chacune de ces unités est une machine à états finie séquentielle, l'ordre associé à chaque unité doit être renforcé, s'il n'est pas déjà total, en un ordre total. Afin d'éviter les inter-blocages, ce renforcement doit cependant être effectué de telle sorte qu'il respecte une contrainte d'ordonnancement que nous allons maintenant expliquer.

Plus précisément, prenons un transfert de données sur un média. Ce transfert est modélisé par l'ajout de deux opérations de communication au graphe de l'algorithme. La première opération de communication correspond à l'écriture dans la mémoire SAM de l'unité de communication émettrice du transfert et la seconde correspond à la lecture dans la mémoire SAM de l'unité de communication réceptrice. Ainsi, l'écriture d'une donnée s'effectue avant sa lecture. De plus, les mémoires mises en jeu lors du transfert étant des mémoires FIFO (cf. modèle d'architecture page 17), un message émis doit être reçu avant qu'un autre message ne soit émis, ceci pour éviter les inter-blocages. Autrement dit, on doit étudier les ordres totaux associés à chaque unité de communication.

Prenons l'exemple de la figure 3.3.5(0) où un média aurait à transférer deux données a, correspondant à la paire d'opérations de communication Ea et La, et b, correspondant à la paire d'opérations de communication Eb et Lb. Le transfert de la donnée a (resp. b), est modélisé par l'ajout de deux opérations de communication, écriture de a (Ea) (resp. écriture de b (Eb)) et lecture de a (La) (resp. lecture de b (Lb)). Ea et Eb sont distribuées sur une des deux unités de communication du média et La et Lb sont distribuées sur l'autre. Il n'y a aucun ordre entre Ea et Eb et entre La et Lb. Le média étant une ressource séquentielle, on doit avoir un ordre total sur chacune des deux unités de communication le composant, mais pas n'importe quel ordre total.

Si on étudie séparément l'ordre partiel associé aux deux unités de communication du

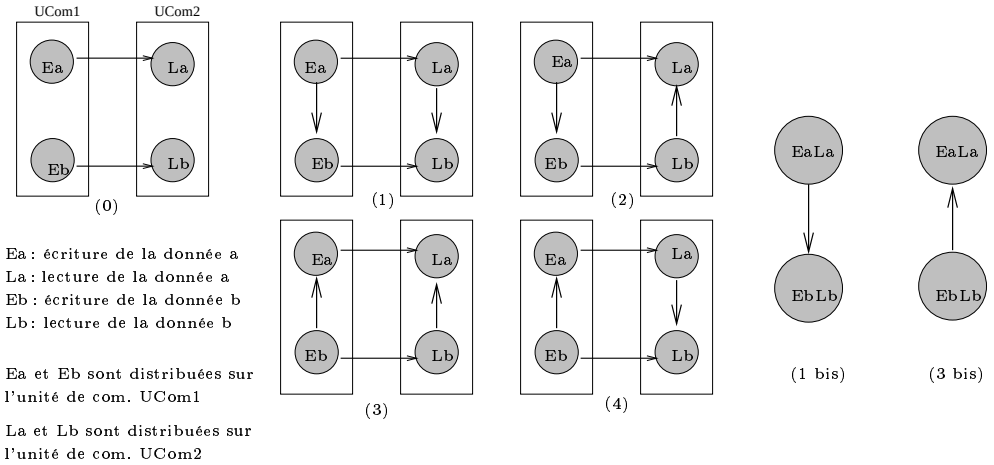


FIG. 3.3.5 – *Contrainte d'ordonnancement*

média, on a quatre ordres totaux possibles (1,2,3,4) dont deux sont valides (1) et (3) et les deux autres génèrent des inter-blocages (2) et (4). On vient de voir, en effet, que l'ordre d'écriture des données à transférer doit être le même que l'ordre de lecture, car les transferts de données ne sont supportés, par hypothèse sur le modèle d'architecture, que par des liaisons de type SAM. Si on regarde l'ordre (2) par exemple on a : Ea précède La, Eb précède Lb, Ea précède Eb et Lb précède La. Autrement dit l'ordre des écritures est Ea, Eb et cet ordre est différent de l'ordre des lectures qui est Lb, La. Ainsi, dès qu'un transfert commence, il doit s'achever avant qu'un autre ne débute sous peine de bloquer la liaison et c'est ce qui se passe pour les ordres (2) et (4).

Pour éviter les inter-blocages, on fusionne les deux opérations de communication de lecture et d'écriture d'une même donnée sur un média et on appelle cette nouvelle opération ainsi construite, *une opération de transfert*. Ainsi, on a la propriété suivante :

Propriété 35 *L'ordre d'écriture des transferts sur les média est égal à l'ordre de lecture.*
Preuve : par construction des opérations de transfert.

Si on reprend l'exemple de la figure 3.3.5 (1 bis, 3 bis), on a fusionné les opérations Ea et La, ce qui donne l'opération de transfert EaLa et on fait de même pour Eb et Lb ce qui donne l'opération EbLb. Donc les deux ordres totaux valides (1) et (3) correspondent aux ordres (1bis) et (3bis). On voit bien qu'en fusionnant les deux opérations on ne peut plus produire des ordres comme l'ordre (2) ou l'ordre (4), on a donc bien supprimé les inter-blocages.

Nous allons maintenant étudier comment s'effectue le renforcement de l'ordre partiel associé à chaque unité de calcul et à chaque média.

Premièrement, il existe un arc entre deux opérations de calcul ou deux opérations de transfert. Dans ce cas, il n'y a pas d'arc à rajouter, l'ordre est déjà total, donc il n'y a pas

d'ordonnancement à réaliser.

Deuxièmement, il n'y a pas d'arc entre deux opérations de calcul ou de transfert, alors :

- s'il est possible de construire une route qui joint deux opérations, alors ces opérations sont dites logiquement dépendantes (définition 19) [13], c'est-à-dire une doit être exécutée avant l'autre. La route doit être sans circuit et est obtenue par fermeture transitive de la relation $\preceq$ à la puissance correspondant au nombre d'arcs. Ainsi, l'ordonnancement est guidé par les dépendances du graphe de l'algorithme.
- s'il n'est pas possible de construire un chemin entre deux opérations, ces opérations sont dites *logiquement indépendantes* (définition 22) [13] ou *concurrentes* [89]. Cependant, comme ces opérations doivent être *causalement dépendantes*, une d'elles doit être contrainte à être exécutée avant l'autre. Ce choix est effectué par les heuristiques d'optimisation de la durée d'exécution de l'implantation en prenant en compte les durées d'exécution des opérations du graphe de l'algorithme.

3.3.2 Formalisation

Soit O''^t, l'ensemble des opérations de transfert associées au graphe de l'algorithme.

$$O''^t = \{o''_k\}_{1 \leq k \leq \frac{n''}{2}}$$

On appelle O, l'ensemble des opérations de calcul et des opérations de transfert.

$$O = O' \cup O''^t$$

Soit Π'', l'application qui, à chaque opération de transfert, associe le média sur lequel elle est distribuée.

$$\boxed{\begin{array}{llll} \Pi'' : & O''^t & \rightarrow & M \\ & o''_i & \mapsto & \Pi''(o''_i) = m_j \end{array}}$$

Soit Π''^{-1}, l'application réciproque de Π'', qui associe à chaque média, l'ensemble des opérations de transfert distribuées sur ce média.

$$\boxed{\begin{array}{llll} \Pi''^{-1} : & M & \rightarrow & \mathcal{P}(O''^t) \\ & m_j & \mapsto & \Pi''^{-1}(m_j) = \{o''_i \in O''^t \,/\, \Pi''(o''_i) = m_j\} \end{array}}$$

Remarque 40 *Les propriétés associées aux applications Π'' et Π''^{-1} sont étudiées dans la deuxième partie de la thèse, au chapitre Caractérisation page 85.*

Ainsi, $\Pi''^{-1}(m)$ correspond à l'ensemble des opérations distribuées sur le média m, on notera encore O''^{t}_{m} cet ensemble d'opérations.

$O''^{t}_{m} = \bigcup_{i=1}^{n''_m} o''_i$ avec $o''_i \in O''^{t}, \forall m \in M$ et $\sum_{m=1}^{\text{Card }(M)} n''_m = \text{Card}(O''^{t}) = \frac{n''}{2}$

$$\begin{cases} O''^{t}_{m1} \cap O''^{t}_{m2} = \emptyset \quad \forall\, m_1,m_2 \in M \\ \bigcup_{m \in M} O''^{t}_{m} = O''^{t} \end{cases}$$

Chaque graphe linéaire construit lors de la communication est transformé en un autre graphe linéaire dans lequel les opérations de communication associées à un même média sont regroupées en une seule opération, appelée opération de transfert. Soit :

$$d' \xrightarrow{com.} \left(d^*{}_1,o''_{11},d''_1,o''_{21},d^*{}_2,o''_{22},d''_2,...,d''_{n-1},o''_{n1},d^*{}_n\right) \xrightarrow{ordonnancement} (d^*{}_1,o''_1,d^*{}_2,o''_2,...,o''_n,d^*{}_n)$$

avec :

$$\begin{array}{lclcl} o'_i & = & \gamma^{-1}(d^*_1) & = & \gamma^{-1}(d') \\ \gamma(d^*_1) & = & o''_1 & = & \gamma^{-1}(d^*_2) \\ \gamma(d^*_2) & = & o''_2 & = & \gamma^{-1}(d^*_3) \\ ... & & ... & & ... \\ \gamma(d^*_{n-1}) & = & o''_{n-1} & = & \gamma^{-1}(d^*_n) \\ \gamma(d^*_n) & = & o'_j & \subseteq & \gamma(d') \end{array}$$

Le graphe linéaire $(o''_{11},d''_1,o''_{21})$ construit lors de la communication est remplacé lors de l'ordonnancement par l'opération de transfert o''_1 et cette opération de transfert est distribuée sur un média. Ainsi, l'ensemble D'' construit lors de la communication disparaît, car chaque dépendance d'' est "encapsulée" dans l'opération de transfert.

Soit D^*_P, l'ensemble des arcs entre opérations de calcul et de transfert ou de transfert seulement.

$$D^*_P \subseteq (O' \times O''^{t}) \cup (O''^{t} \times O''^{t}) \cup (O''^{t} \times O')$$

Soit $\preceq_c$, l'ordre partiel associé aux dépendances D^*_P.

Soit $\bar{D}''_m$ l'ensemble des dépendances associées au média m. Par construction, $\bar{D}''_m$ induit un ordre total sur l'ensemble des opérations de transfert distribuées sur ce média. On note $\prec_m$, l'ordre total associé au média m.

Soit $\bar{D}'_p$ l'ensemble des dépendances associées à l'unité de calcul du processeur p. $\bar{D}'_p$ induit un ordre total $\prec_p$ sur l'ensemble des opérations de calcul distribuées sur ce processeur.

$$D'_p \subseteq \bar{D}'_p \Leftrightarrow \preceq_p \subseteq \prec_p$$

Soit D, l'ensemble des dépendances du graphe de l'algorithme.

$$D = (\bigcup_{p \in P} \bar{D}'_p) \cup (\bigcup_{m \in M} \bar{D}''_m) \cup D^*_P$$

Soit $\trianglelefteq$, l'ordre partiel associé au graphe de l'algorithme distribué et ordonnancé.

$$\trianglelefteq = ((\bigcup_{p \in P} \prec_p) \cup (\bigcup_{m \in M} \prec_m) \cup (\bigcup_{c \in C} \preceq_c))^*$$

Remarque 41 *Par la suite, on utilisera le terme implantation pour désigner la distribution et l'ordonnancement, et on appellera graphe implanté un graphe distribué et ordonnancé.*

Propriété 36 *L'ordre partiel du graphe de l'algorithme implanté $\trianglelefteq$ contient l'ordre partiel initial $\preceq$.*

$$\trianglelefteq \ \supseteq \ \preceq$$

Preuve : l'ensemble des dépendances D' du graphe initial a été décomposé en deux sous-ensembles D'_P et D'_R qui forment une partition de D'.

$$\Rightarrow D' = D'_P \cup D'_R \quad \Leftrightarrow \quad \preceq = \preceq_P \cup \preceq_R \ \text{ avec } \ \preceq_P = \bigcup_{p \in P} \preceq_p$$

L'ensemble des dépendances D'_p associées à l'unité de calcul du processeur p induit un ordre partiel $\preceq_p$ sur l'ensemble des opérations. Chaque ordre partiel $\preceq_p$ est renforcé en un ordre total $\prec_p$. Chaque ordre total $\prec_p$ contient donc l'ordre partiel $\preceq_p$. Ainsi, l'ordre partiel associé à l'algorithme contient l'union des ordres partiels $\preceq_P$.

$$\preceq_P \subseteq \trianglelefteq$$

Étudions maintenant l'ensemble D'_R. Soit $d' \in D'_R$, par construction la dépendance d' est remplacée par un graphe linéaire $(d^*{}_1, o''_1, d^*{}_2, o''_2, ..., o''_n, d^*{}_n)$. Ainsi

$$\gamma^{-1}(d') \preceq_R \gamma(d') \Rightarrow \begin{cases} \gamma^{-1}(d') \trianglelefteq \gamma(d^*_1) = o''_1 \\ \gamma(d^*_1) \trianglelefteq \gamma(d^*_2) = o''_2 \\ \quad ... \\ \gamma^{-1}(d^*_n) \trianglelefteq \gamma(d^*_n) \end{cases} \Rightarrow \gamma^{-1}(d') \trianglelefteq \gamma(d')$$

Ainsi, l'ordre partiel $\preceq_R$ est inclus dans l'ordre partiel $\trianglelefteq$ de l'algorithme implanté :

$$\Rightarrow (\preceq_P \cup \preceq_R) \ \subseteq \trianglelefteq \qquad \Leftrightarrow \qquad \preceq \subseteq \trianglelefteq$$

et donc l'ordre partiel initial est contenu dans l'ordre partiel construit lors de l'ordonnancement.

Propriété 37 *L'algorithme implanté est garanti sans inter-blocage.*

Propriété 38 *Chaque opération de transfert $o'' \in O''^t$ a un et un seul successeur o_i appartenant à O et un et un seul prédécesseur o_j appartenant à O (par construction).*

$$\Gamma(o''_k) = \{o_i\} \ \textit{et} \ \Gamma^{-1}(o''_k) = \{o_j\}$$

Remarque 42 *Les prédécesseur et successeur d'une opération (de calcul ou de transfert) sont soit de type calcul, soit de type transfert.*

Propriété 39 *Entre deux opérations de transfert situées sur un même média, il n'y a aucune dépendance de données directe, il n'y a que des dépendances transitives.*
Preuve : en effet, chaque opération de transfert a un et un seul émetteur et un et un seul récepteur. Le prédécesseur (resp. successeur) est soit une opération de calcul, soit une opération de transfert. Si c'est une opération de calcul, la dépendance entre les deux ne peut figurer sur le média et si c'est une opération de transfert, elle est obligatoirement distribuée sur un autre média. Sinon les opérations seraient confondues car, par construction, on distribue les opérations de transfert pour une même dépendance de données sur des média différents.

Remarque 43 *Chaque opération de transfert est une fonction identité.*

Remarque 44 *Une opération de transfert est conditionnée par le même booléen que celui qui conditionne l'opération de calcul qui la précède. Ainsi, tout comme une opération de calcul, une opération de transfert peut être conditionnée ou non.*

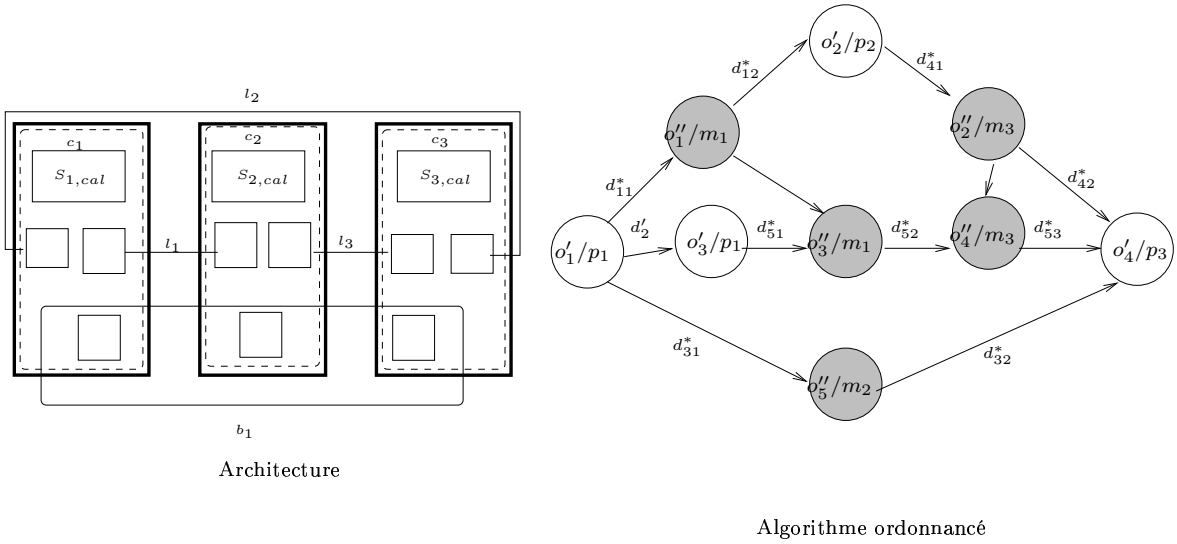


FIG. 3.3.6 – *Un ordonnancement*

On considère l'ordonnancement présenté dans la figure 3.3.6. Cet ordonnancement fait suite à la distribution présentée dans la figure 3.2.4. Chaque opération de transfert associée à un média remplace les deux opérations de communication associées aux unités de communication composant ce média lors de la communication. Ainsi, l'opération de transfert

o''_1/m_1 remplace o''_1 et o''_2, de même o''_3/m_1 remplace o''_5 et o''_6, o''_4/m_3 remplace o''_7 et o''_8, o''_2/m_3 remplace o''_3 et o''_4, enfin o''_5/m_2 remplace o''_9 et o''_{10}.

Étudions maintenant l'ordonnancement sur chacune des ressources de type unité de calcul ou de type média.

Sur les unités de calcul : Processeur 1: $O'_{p_1} = \{o'_1,o'_3\}$ et $D'_{p_1} = (o'_1,o'_3) \Longleftrightarrow o'_1 \preceq o'_3 \Longrightarrow$ Pas d'ordonnancement, Processeur 2: $O'_{p_2} = \{o'_2\}$ Une seule opération $\Longrightarrow$ Pas d'ordonnancement, Processeur 3: $O'_{p_3} = \{o'_3\}$ Même cas que (2).

Sur les média : Sur le média m_1 : $O''^t_{m_1} = \{o''_1,o''_3\}$ (*) o''_1 et o''_3 sont logiquement indépendantes. Elles doivent être rendues causalement dépendantes. On choisit d'ajouter l'arc (o''_1,o''_3). Sur le média m_2 : $O''^t_{m_2} = \{o''_5\} \Longrightarrow$ Pas d'ordonnancement. Sur le média m_3 : $O''^t_{m_3} = \{o''_2,o''_4\}$. Même cas que (*), on choisit d'ajouter l'arc (o''_2,o''_4).

Remarque 45 *Chaudhary et Aggarwal ont décrit dans [14], un* one-to-one mapping *garanti sans dead-lock des opérations du graphe de l'algorithme sur des pseudo-processeurs, qui peuvent être vus comme des unités de calcul, qui composent leur* extended host graph. *Leurs pseudo-processeurs sont numérotés pour chaque processeur, par des index temporels, ce qui conduit à un ordre total sur chaque processeur. Mais rien n'est dit concernant les média reliant ces processeurs.*

Remarque 46 *Soit* $(G_{dR}(i),G'_{ar})$, *le ième graphe distribué associé au graphe de l'algorithme* (O,D) *et au graphe de l'architecture routé* G'_{ar}. *Alors on peut construire plusieurs graphes ordonnancés* $(G_S(i,.),G'_{ar})$, *associés au couple* $(G_{dR}(i),G'_{ar})$, *mais leur nombre* N_i *est fini. Cela peut être réécrit :* $(G_{dR}(i),G'_{ar})\mathcal{R}_{ordo}(G_S(i,j),G'_{ar})$ *avec* $1 \leq j \leq N_i$ *où* N_i *est le nombre de graphes ordonnancés associés au graphe distribué* $G_{dR}(i)$ *et* $\mathcal{R}_{ordo}$ *est la relation "a pour graphe ordonnancé sur le modèle développé".*

Propriété 40 *L'ordonnancement est une relation* $\mathcal{R}_{ordo}$ *de l'ensemble des couples (graphe orienté, graphe non orienté) vers l'ensemble des couples (graphe orienté, graphe non orienté).*

De plus, le graphe de l'architecture est inchangé et le nombre de d'ordonnancements associés au graphe distribué est fini, car le nombre de sommets et d'arcs est fini.

Soit Π, l'application qui, à chaque opération de calcul (resp. de transfert) associe le processeur (resp. le média) sur lequel elle a été implantée.

$$\boxed{\begin{array}{lcl} \Pi : \quad O & \rightarrow & M \cup P \\ o_i & \mapsto & \Pi(o_i) = \begin{cases} \Pi'(o_i) \text{ si } o_i \in O' \\ \Pi''(o_i) \text{ si } o_i \in O'' \end{cases} \end{array}}$$

Soit Π^{-1}, l'application réciproque de Π, qui associe à chaque processeur (resp. chaque média), l'ensemble des opérations de calcul (resp. de transfert) implantées sur ce processeur (resp. ce média).

$$\boxed{\begin{array}{llll} \Pi^{-1}: & M \cup P & \rightarrow & \mathcal{P}(O) \\ & u_j & \mapsto & \Pi^{-1}(u_j) = \begin{cases} \{o''_i \in O'' \ / \ \Pi''(o''_i) = u_j\} \text{ si } u_j \in M \\ \{o'_i \in O' \ / \ \Pi'(o'_i) = u_j\} \text{ si } u_j \in P \end{cases} \end{array}}$$

Remarque 47 *Nous verrons dans la deuxième partie de la thèse, au chapitre Caractérisation page 85, les propriétés associées aux applications* Π *et* Π^{-1}.

3.4 Composition des relations

La composition des trois relations précédentes, $\mathcal{R}_{routage}$, $\mathcal{R}_{dis}$ et $\mathcal{R}_{ordo}$, donne l'ensemble des implantations valides possibles d'un algorithme donné sur une architecture donnée.

Pour un couple donné $G_{alar} = (G_{al},G_{ar})$, en posant $G_{Sar'}(i,j) = (G_S(i,j),G'_{ar})$, on a :

$$\boxed{\begin{array}{l} (\mathcal{R}_{ordo} \circ \mathcal{R}_{dis} \circ \mathcal{R}_{routage})(G_{alar}) = \{G_{Sar'}(1,1), \cdots ,G_{Sar'}(1,N_1), \cdots , \\ G_{Sar'}(i,1), \cdots ,G_{Sar'}(i,N_i), \cdots ,G_{Sar'}(n,1), \cdots ,G_{Sar'}(n,N_n)\} \end{array}}$$

Finalement, l'implantation peut être résumée comme dans la figure 3.4.7.

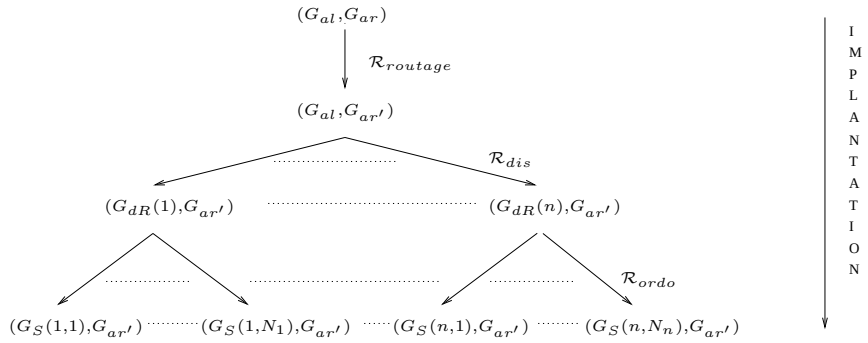


FIG. 3.4.7 – *Résumé de l'implantation*

3.5 Conclusion

Dans les applications complexes, impliquant des algorithmes irréguliers avec beaucoup de dépendances de données et des architectures parallèles, les communications inter-processeur représentent un point très important qui doit être considéré avec attention. Nous avons

modélisé en utilisant le même formalisme l'algorithme, l'architecture et l'ensemble de toutes les implantations valides de l'algorithme sur l'architecture. Cet ensemble est construit en composant trois relations : le routage, la distribution et l'ordonnancement.

Étant donné que les architectures réelles ne sont pas nécessairement complètement connectées, nous avons introduit le routage, qui permet non seulement les communications inter-processeur entre deux processeurs non directement connectés, mais en plus des communications en parallèle entre deux processeurs.

La distribution et l'ordonnancement sont d'habitude décrites sur le modèle d'architecture encapsulé. Ici, nous introduisons le modèle d'architecture développé qui permet de supporter chaque communication inter-processeur par une liaison plus deux unités de communication, chacune des unités appartenant à un processeur différent. Comme chaque unité de communication est une machine séquentielle à états finis, il est maintenant possible d'ordonnancer les communications inter-processeur distribuées dessus. Avec cette approche, étant donné que la distribution et l'ordonnancement sont plus précises au niveau des communications inter-processeur, nous sommes capables de faire un meilleur choix d'implantation parmi l'ensemble des implantations valides obtenues en composant les trois relations précédentes.

Deuxième partie

Optimisation et heuristique

1. Caractérisation

1.1 Introduction

A l'issue de la formalisation de l'implantation, on a associé à chaque itération du graphe de dépendances de données un instant logique et une relation d'ordre partiel $\trianglelefteq$, $o_i \trianglelefteq o_j \Leftrightarrow$ o_i "est exécuté avant" o_j. L'ensemble des opérations d'une itération donnée est associé à un instant logique. En ce sens ils sont simultanés.

L'ensemble des itérations du graphe de dépendances décrit un ensemble d'instants logiques. Cet ensemble discret, totalement ordonné, est appelé le temps logique.

En exploitant les caractéristiques de l'architecture sur laquelle le graphe de dépendances va être implanté, une nouvelle notion de temps intervient : *le temps physique*. L'exécution d'une opération de calcul o'_i a une durée (dans le temps physique) qui dépend du processeur p_j qui l'exécute, on notera cette durée $\Delta(o'_i,p_j)$. L'intervalle de temps physique pendant lequel o'_i occupe p_j, aussi appelé intervalle d'exécution de o'_i sur p_j a une date de début $S(o'_i)$ et une date de fin $E(o'_i)$. Ces deux dates sont liées par la relation suivante :

$$E(o'_i) = S(o'_i) + \Delta(o'_i,p_j)$$

De même, lorsque deux opérations dépendantes sont distribuées et ordonnancées sur deux processeurs différents, la dépendance de données va donner lieu à un transfert de données sur chacun des média composant la route reliant les deux processeurs. Chaque transfert de données sur un média est représenté par un sommet spécial ajouté au graphe initial de l'algorithme et appelé *opération de transfert*. Pour chaque dépendance de données, on insère entre les deux opérations dépendantes autant d'opérations de transfert qu'il y a de média sur la route. Ainsi, l'exécution d'une opération de transfert o''_k associée à la dépendance de données d_k a une durée (dans le temps physique) qui dépend du média m_j qui l'exécute, on notera cette durée $\Delta(o''_k,m_j)$. L'intervalle de temps physique pendant lequel une des opérations de transfert o''_k correspondant à la donnée d_k occupe le média de communication m_j a une date de début $S(o''_k)$ et une date de fin $E(o''_k)$. Ces dates sont liées par la relation suivante :

$$E(o''_k) = S(o''_k) + \Delta(o''_k,m_j)$$

L'ordre partiel entre les opérations (de type calcul ou de type transfert) dépendantes se

traduit en une inégalité sur les dates de ces opérations :

$$o_i \trianglelefteq o_j \Leftrightarrow E(o_i) \leq S(o_j)$$

Définition 47 *Soit U, l'ensemble des ressources. U est composé de l'ensemble des unités de calcul des processeurs P, appelés pour simplifier processeurs, et de l'ensemble des média M.*

$$U = P \cup M$$

Un instant logique correspond à l'indice t associé au motif répété $motif(t)$ défini dans le modèle d'algorithme à la page 48. Comme on l'a vu, à ce motif est associé un ordre partiel. L'exécution de chaque opération de ce motif occupe un certain intervalle de temps physique, donc l'exécution d'un motif occupe également un certain intervalle de temps physique. S'il y a plusieurs ressources disponibles, deux opérations indépendantes peuvent s'exécuter sur des ressources différentes au même instant physique.

Définition 48 *On appelle intervalle d'exécution associé à une ressource, l'intervalle de temps physique durant lequel des opérations s'exécutent sur la ressource.*

Lorsque deux opérations indépendantes s'exécutent au même instant physique sur des ressources différentes, on dit qu'il y a recouvrement d'intervalles d'exécution et que l'exécution est parallèle.

Caractéristique 1 *Au cours d'un même instant physique, plusieurs opérations peuvent s'exécuter simultanément mais il y a au plus une opération qui s'exécute sur chaque ressource composant l'architecture. Le parallélisme est inter-processeur (resp. inter-lien) et il n'y a pas de parallélisme entre opérations au niveau intra-processeur (resp. intra-lien).*

Nous allons tout d'abord définir la fonction Δ, puis nous dirons à quoi est égale cette fonction pour quelques modèles d'architecture, puis nous définirons S et E récursivement. Enfin nous définirons des fonctions de composition de S et E.

1.2 Définitions

Le graphe de l'algorithme est un graphe hypergraphe orienté flot de données et conditionné.

Soit $\Delta(o_i,u_j)$ la durée d'exécution de l'opération o_i s'exécutant sur la ressource u_j.

$$\boxed{\begin{array}{llll} \Delta : & O \times U & \rightarrow & \mathbb{R}^+ \\ & (o_i,u_j) & \mapsto & \Delta(o_i,u_j) \end{array}}$$

Domaine de définition $\mathcal{D}_\Delta$ de Δ :

$$\mathcal{D}_\Delta = \mathcal{D}' \cup \mathcal{D}''$$

Avec :

$$\mathcal{D}' = \{(o'_i,p_j) \in O' \times P \text{ tq } o'_i \text{ soit exécutable sur } p_j\}$$

En effet, il se peut que certaines opérations de calcul n'aient pas de réalisation sur certains types de processeurs. Par exemple, un opérateur spécialisé est incapable d'exécuter d'autres opérations de calcul que celle pour laquelle il a été conçu.

Remarque 48 *Si l'architecture est homogène au niveau des processeurs, on a :*

$$\Delta(o'_i,p_j) = \Delta(o'_i) \quad \forall p_j \in P$$

$$\mathcal{D}'' = \{(o''_k,m_j) \in O'' \times L \text{ tq } o''_k \text{ soit exécutable sur } m_j\}$$

En effet, il se peut que le même type de données ait un codage mémoire différent sur deux processeurs et qu'on n'ait pas réalisé (à cause d'un coût prohibitif ou pour toute autre raison) le transcodage que nécessiterait l'exécution d'une opération de transfert de ce type de données entre les deux processeurs.

Remarque 49 *Une dépendance de données intra-processeur, ne donnant lieu à aucun transfert de données, a une durée physique nulle.*

Définition 49 *Un graphe d'algorithme* étiqueté *est un graphe dont on connaît les durées d'exécution des opérations sur les différentes ressources où elles peuvent être exécutées.*

Voyons sur quelques exemples d'architecture quelles sont les valeurs de cette fonction.

1.2.1 Modèle PRAM homogène

Chaque opération de calcul du graphe de l'algorithme est étiquetée par une constante positive ou nulle correspondant à sa durée d'exécution sur un des processeurs de l'architecture.

$$\Rightarrow \Delta(o'_i,p_j) = \Delta(o'_i) \quad \forall p_j \in P \forall o'_i \in O'$$

De plus les processeurs dans ce modèle communiquent grâce à une mémoire globale partagée, donc les durées de communication inter-processeur sont nulles et il n'y a pas d'opération de transfert.

$$\Rightarrow \Delta(o''_k,m_j) \text{ n'est pas définie}$$

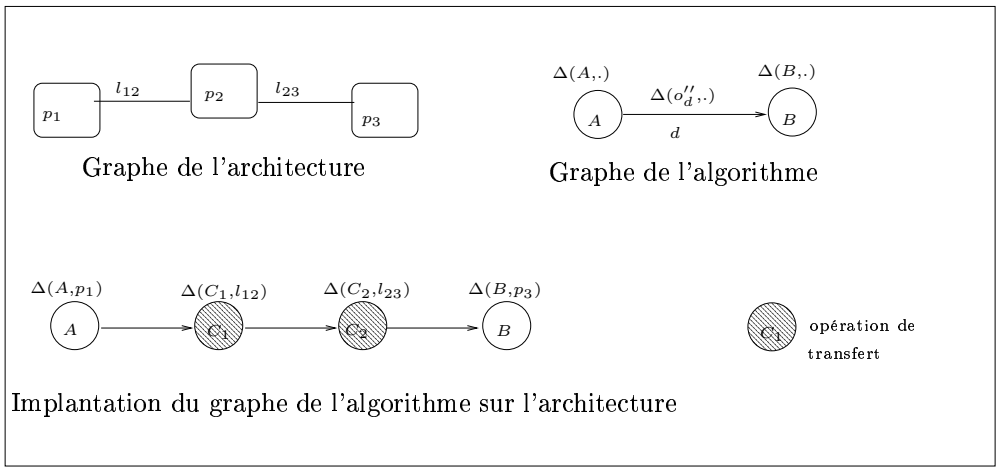


FIG. 1.2.1 – *Insertion d'opérations de transfert*

1.2.2 Modèle PRAM hétérogène

La durée d'exécution de chaque opération de calcul dépend du processeur sur lequel elle est implantée. Donc, la fonction Δ ne se simplifie pas pour ce type d'opération. Par contre, la durée d'exécution de chaque transfert de données est nulle comme dans le modèle PRAM homogène, et donc comme précédemment la fonction Δ n'est pas définie pour les opérations de transfert.

1.2.3 Modèle DRAM homogène complètement connecté

Comme dans le modèle PRAM homogène, chaque opération de calcul est étiquetée par sa durée d'exécution. Par contre, les processeurs communiquent entre eux par transferts de données entre leur mémoire, mais sans interférence entre les transferts. La durée d'une opération de transfert ne dépend ni du processeur émetteur ni du processeur récepteur, elle ne dépend que de la dépendance de donnée (c'est-à-dire en fait du volume de données à transférer).

$$\Rightarrow \Delta(o''_k,m_j) = \Delta_(o''_k) \quad \forall o''_k \in O''$$

1.2.4 Modèle DRAM hétérogène connexe et incomplètement connecté

Les fonctions ne peuvent être simplifiées. Une fois la distribution et l'ordonnancement effectués on a des nouvelles opérations dites opérations de transfert. Chaque opération de transfert modélise un transfert de données sur un média. On associe à cette opération, la durée du transfert de données sur le média (cf. figure 1.2.1).

C'est ce modèle que nous allons utiliser par la suite.

1.3 Dates associées à une implantation

Étant donnée une implantation et connaissant les durées des opérations, on cherche à définir les dates de début et de fin d'exécution de chaque opération (on verra dans le chapitre suivant comment à l'aide d'une heuristique, on construit une implantation). Ces dates peuvent être réparties en deux sous-ensembles :

- dates définies depuis le début,
- dates définies depuis la fin : l'axe des temps et son origine sont dans le sens opposé à celui des dates définies depuis le début (cf. figure 1.3.2).

Remarque 50 *Les dates définies depuis le début associées à une opération font intervenir les dates de ses prédécesseurs tandis que les dates définies depuis la fin font intervenir les dates de ses successeurs.*

Avant de définir ces dates, on rappelle la définition de la fonction Π donnée dans le modèle d'implantation à la page 76.

Soit Π, l'application qui, à chaque opération (de type calcul ou de type transfert) associe la ressource (de type processeur ou média) sur laquelle elle a été implantée. Soit Π^{-1}, l'application réciproque de Π, qui associe à chaque ressource, l'ensemble des opérations implantées sur cette ressource.

$$\boxed{\begin{array}{llll} \Pi: & O & \rightarrow & U \\ & o_i & \mapsto & \Pi(o_i) = u_j \end{array}} \quad \text{et} \quad \boxed{\begin{array}{llll} \Pi^{-1}: & U & \rightarrow & \mathcal{P}(O) \\ & u_j & \mapsto & \Pi^{-1}(u_j) = \{o_i \in O / \Pi(o_i) = u_j\} \end{array}}$$

Domaine de définition $\mathcal{D}_{\Pi^{-1}}$ de l'application Π^{-1} :

$$\mathcal{D}_{\Pi^{-1}} = \Pi(O)$$

Ainsi, l'opération o_i est implantée sur la ressource $\Pi(o_i)$.

Propriété 41 Π *est une fonction.* En effet, toute opération du graphe est implantée sur une et une seule ressource. $(\forall o_i \in O \;\; \exists! \; \Pi(o_i) \in U)$.

Propriété 42 Π *est non injective.* En effet, sur une ressource plusieurs opérations peuvent être implantées. $(\Pi(o_i) = \Pi(o_j) \not\Rightarrow o_i = o_j)$.

Propriété 43 Π *est non surjective.* En effet, sur certaines ressources composant l'architecture aucune opération ne peut être implantée (cas où le nombre de processeurs est supérieur au nombre d'opérations de calcul du graphe par exemple mais ce n'est pas le seul cas). L'ensemble $\Pi(O)$ est un sous-ensemble de U. $(\Pi(O) \subseteq U)$.

Propriété 44 Π^{-1} *est une injection.* En effet, toute opération est implantée sur une et une seule ressource. ($\forall\, u_1,u_2 \in \Pi(O) \times \Pi(O)$ $[\Pi^{-1}(u_1) = \Pi^{-1}(u_2) \Rightarrow u_1 = u_2]$).

Un exemple de calculs de dates est donné figure 1.3.2, l'architecture est un ensemble de deux processeurs PRAM homogènes (chaque opération de calcul a donc une et une seule durée d'exécution indiquée au-dessus de l'opération) communiquant par mémoire partagée donc il n'a pas de communication inter-processeur donc pas d'opération de transfert.

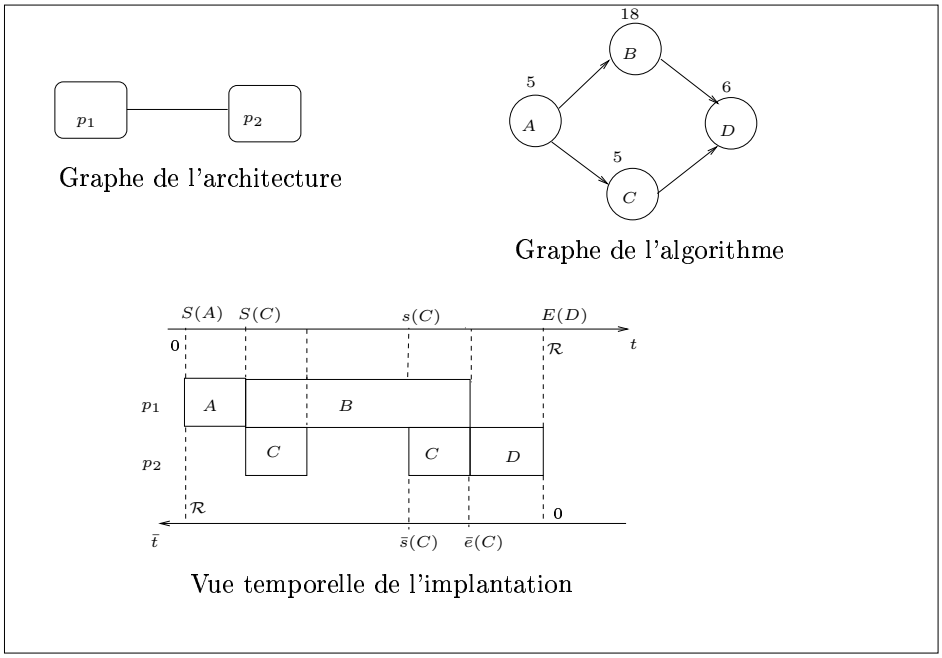


FIG. 1.3.2 – *Dates associées à une implantation*

1.3.1 Notations

Une majuscule (resp. une minuscule) désigne une date au plus tôt (resp. au plus tard).

Une lettre sans barre au-dessus (resp. avec barre) désigne une date définie depuis-le-début (resp. depuis-la-fin).

La lettre s (resp. e) désigne une date de début (resp. de fin).

On suppose que : $\max_{\emptyset}(.) = 0$.

1.3.2 Dates au-plus-tôt (définies depuis le début)

Soit $S(o_i)$, la date de début au-plus-tôt de o_i (aussi appelée habituellement $ASAP(o_i)$ "As-Soon-As-Possible", $AEST(o_i,\Pi(o_i))$ -Absolute Earliest Start Time- [48] ou $ST(o_i,\Pi(o_i))$ -Start Time-). Comme l'exécution d'une opération ne peut commencer qu'après la fin de

l'exécution de ses prédécesseurs, la date de début au-plus-tôt d'une opération est égale à la plus grande date de fin de ses prédécesseurs (de type calcul ou de type transfert).

Soit $E(o_i)$, la date de fin au-plus-tôt de o_i : (aussi appelée habituellement $FT(o_i,\Pi(o_i))$ "Finish Time").

Alors :

$$S(o_i) \geq E(o_j) \quad \forall\ o_j \trianglelefteq o_i \Rightarrow \begin{cases} S(o_i) = \max\limits_{o_j \trianglelefteq o_i} E(o_j) \\ E(o_j) = S(o_j) + \Delta(o_j,\Pi(o_j)) \end{cases}$$

Les dates de début et de fin au-plus-tôt (depuis le début) des opérations du graphe de l'algorithme de la figure 1.3.2 sont données dans la table 1.3.1.

	S	E
A	0	5
B	5	23
C	5	10
D	23	29

TAB. 1.3.1 – *Dates au-plus-tôt (depuis le début) associées à la figure 1.3.2*

1.3.3 Chemin critique & opérations critiques

Le chemin critique d'un graphe G_l donné est le plus long chemin de ce graphe en terme de coût. Pour les arcs et les sommets composant ce chemin, on parlera d'arcs critiques et de sommets critiques.

Définition 50 *La longueur $\mathcal{R}$ du chemin critique est égale à plus grande des dates de fin des opérations composant le graphe de l'algorithme.*

$$\mathcal{R} = \max_{o_k \in O} E(o_k)$$

1.3.4 Dates au-plus-tard (définies depuis la fin)

Soit $\bar{s}(o_i)$, la date de début au-plus-tard de o_i. Soit $\bar{e}(o_i)$, la date de fin au-plus-tard de o_i. Alors :

$$\bar{e}(o_i) \geq \bar{s}(o_j) \quad \forall\ o_i \trianglelefteq o_j \Rightarrow \begin{cases} \bar{e}(o_i) = \max\limits_{o_i \trianglelefteq o_j} \bar{s}(o_j) \\ \bar{s}(o_j) = \bar{e}(o_j) + \Delta(o_j,\Pi(o_j)) \end{cases}$$

Les dates de début et de fin au-plus-tard (depuis la fin) des opérations du graphe de l'algorithme de la figure 1.3.2 sont données dans la table 1.3.2.

	$\bar{s}$	$\bar{e}$
A	29	24
B	24	6
C	11	6
D	6	0

TAB. 1.3.2 – *Dates au-plus-tard (depuis la fin) associées à la figure 1.3.2*

Définition 51 *Si pour une opération o_i, $s(o_i) = S(o_i)$, alors on dit que o_i est une opération critique.*

1.3.5 Flexibilité d'ordonnancement d'une opération o_i : $F(o_i)$

La flexibilité d'ordonnancement d'une opération [58, 54] encore appelée mobilité ou marge d'ordonnancement est l'écart entre sa date de début au plus tard et sa date de début au plus tôt.

$$F(o_i) = s(o_i) - S(o_i) = e(o_i) - E(o_i)$$

Flexibilité relative de o_i : $MR(o_i)$

$$MR(o_i) = \frac{F(o_i)}{\Delta(o_i)}$$

2. Etat de l'art

2.1 Introduction

Les problèmes d'allocation de ressources peuvent se décomposer en deux catégories : *les problèmes de placement*, et *les problèmes d'ordonnancement*. Dans l'étude des systèmes distribués temps réel, on rencontre les deux problèmes simultanément. En effet, quand l'architecture est distribuée, on est obligé de faire du placement (ce que l'on a appelé dans la première partie de la thèse *distribution*) et de l'ordonnancement car chaque unité composant l'architecture est une ressource purement séquentielle (cf. modèle d'architecture). Par abus de langage, quand on parle de problème de placement sans ordonnancement, cela veut dire que l'on s'intéresse à un problème de placement-ordonnancement orienté placement, et que l'on suppose que l'on a un modèle de graphe d'algorithme non orienté où les dépendances de données entre opérations sont prises en compte mais pas les dépendances d'ordre d'exécution associées. Ainsi, l'ordonnancement de ce graphe est guidé non pas par les précédences du graphe mais par le respect de l'équilibrage de la charge sur chaque unité de calcul et sur chaque média composant le graphe de l'architecture. Ainsi, dans les problèmes de placement sans ordonnancement, le modèle d'algorithme est souvent un graphe non orienté et ce qu'on cherche à optimiser n'est plus la longueur d'ordonnancement mais plutôt l'équilibrage de la charge ou encore la minimisation des coûts de communication.

Nous nous intéressons ici au problème d'ordonnancement, c'est-à-dire au problème de placement-ordonnancement orienté ordonnancement. Plus précisément, on cherche à avoir une durée d'exécution du graphe de l'algorithme implanté qui soit inférieure ou égale à la contrainte temps réel du système que l'on étudie. Pour ce faire, l'algorithme d'application est placé (ce que nous avons appelé *distribué*) et ordonnancé sur les différentes ressources composant l'architecture en ayant le souci non plus uniquement de l'équilibrage de la charge comme c'est le cas dans les problèmes de placement-ordonnancement orienté placement, mais de la durée d'exécution totale du système. L'algorithme d'ordonnancement est cette fois-ci guidé par les précédences du graphe de l'algorithme d'application. Par la suite, on ne parlera que de problème d'ordonnancement plutôt que de dire problème de placement-ordonnancement orienté ordonnancement. Étant donné que ce type de problèmes fait l'objet de nombreuses recherches, on ne pourra pas toutes les passer en revue. C'est pourquoi, nous allons restreindre le cadre de cette étude en prenant des hypothèses qui correspondent à

celles des problèmes traités ici.

Après avoir défini les problèmes d'ordonnancement de manière générale et les problèmes de complexité engendrés, nous allons faire un inventaire des méthodes de résolution existantes, puis nous présenterons quelques méthodes approchées, d'une part les méthodes gloutonnes et d'autre part les méthodes de voisinage. Enfin nous détaillerons les heuristiques que nous proposons.

Pour finir, nous essaierons de comparer les heuristiques présentées avec celles que nous proposons.

2.2 Contexte de l'étude

L'algorithme d'ordonnancement est *statique* ou encore *hors ligne*, c'est-à-dire que l'ensemble des opérations du graphe de l'algorithme d'application et leurs durées sont supposées connues avant le début de l'exécution de l'implantation de l'algorithme d'application sur l'architecture. Ainsi, l'algorithme d'ordonnancement est exécuté lors de la compilation et pas à l'exécution. Cette hypothèse s'oppose aux algorithmes d'ordonnancement dits *dynamiques* ou encore *en ligne*. En effet, quand les durées des opérations du graphe de l'algorithme d'application ne sont connues qu'au moment de l'exécution, l'ordonnancement ne peut être effectué qu'au fur et à mesure du déroulement de l'algorithme d'application. Dans toute la suite de la thèse, on se place uniquement dans un contexte statique.

Les études sur le placement et l'ordonnancement étant très nombreuses, il est difficile d'en faire une classification exhaustive. En effet, elles diffèrent par : les modèles d'algorithme d'application, d'architecture et d'implantation et les contraintes qu'elles supposent et les objectifs qu'elles se proposent d'atteindre.

Graham a défini dans [31] un formalisme à trois champs $\alpha|\beta|\gamma$ permettant de préciser : le modèle d'architecture à l'aide du champ α, le modèle d'algorithme à l'aide du champ β et les contraintes et objectifs de la méthode de résolution à l'aide du champ γ. Chacun de ces trois champs possède plusieurs composantes, chaque composante représentant une caractéristique différente. Ce formalisme permet de comparer les heuristiques sur les modèles d'algorithme, et d'architecture qu'elles supposent. En revanche, le modèle d'implantation que nous avons détaillé dans la première partie de la thèse, n'est pas directement explicité avec ce formalisme comme on va le voir. Par exemple le fait de supposer que les liens inter-processeur ont une capacité infinie a pour conséquence sur le modèle d'implantation que la distribution des communications va être prise en compte mais pas nécessairement leur ordonnancement.

Nous allons décrire les hypothèses que nous avons prises en utilisant le formalisme de Graham. Puis nous dirons de manière explicite les conséquences de ces hypothèses sur le

modèle d'implantation choisi. Pour avoir plus de renseignements concernant les champs utilisés, puisqu'ici nous ne donnons que les valeurs des champs correspondant à nos hypothèses, on peut consulter la thèse de J.P. Beauvais [5].

Architecture

$\alpha_1 = R_m \Rightarrow$ les processeurs n'ont pas tous la même vitesse de traitement, ce qui revient à dire que les processeurs de l'architecture sont *hétérogènes*.

$$\Delta(o'_i,p_j) \neq \Delta(o'_i,p_k) \qquad p_j \neq p_k$$

$\alpha_2 = H^- \Rightarrow$ les média n'ont pas tous la même vitesse de traitement, les média de l'architecture sont *hétérogènes*.

$$\Delta(o''_i,m_j) \neq \Delta(o''_i,m_k) \qquad m_j \neq m_k$$

$\alpha_3 = C^- \Rightarrow$ le graphe de l'architecture n'est pas nécessairement complètement connecté.

$\alpha_4 = A^- \Rightarrow$ les média ont une capacité finie, ainsi si l'heuristique ne prend pas en compte l'ordonnancement des messages sur chaque média, elle va générer des inter-bloquages.

On suppose de plus que le modèle d'architecture est *déterministe*, c'est-à-dire que l'on connaît de manière exacte et non probabiliste les paramètres de l'architecture, que le coût de communications entre deux opérations dépendantes distribuées sur le même processeur est nul. On suppose enfin la possibilité de *recouvrement des calculs et des communications* sur un même processeur.

Remarque 51 *Dans la plupart des processeurs comme le TMS320C40 par exemple, il n'y a qu'un seul séquenceur d'instruction. Donc, quand une communication arrive ou part du le processeur, celui-ci doit interrompre son calcul, afin d'activer le DMA qui réalisera la communication, et ensuite, le calcul et la communication peuvent s'effectuer en parallèle. Comme l'interruption est de très courte durée, on ne la modélise pas et donc c'est raisonnable de supposer le recouvrement des calculs et des communications.*

Algorithme

$\beta_1 = \emptyset \Rightarrow$ chaque opération a une durée d'exécution quelconque.

$\beta_2 = \emptyset \Rightarrow$ les opérations n'ont pas de date de réveil.

$\beta_3 = \emptyset \Rightarrow$ les opérations n'ont pas de dates d'échéances (ou encore dead-line).

$\beta_4 = \tau_i \Rightarrow$ chaque opération est répétitive.

$\beta_5 = Com \Rightarrow$ les opérations communiquent entre elles.

$\beta_6 =<\Rightarrow$ les opérations du graphe de l'algorithme sont liées par des contraintes de précédences liées à l'ordre partiel initial associé à l'algorithme.

$\beta_7 = \emptyset \Rightarrow$ le graphe a une structure quelconque.

$\beta_8 = pmtn^- \Rightarrow$ l'exécution de chaque opération s'effectue sans préemption. Pratiquement, cela signifie que l'on ne peut interrompre l'exécution d'une opération donnée pour en traiter une plus prioritaire.

$\beta_9 = \emptyset \Rightarrow$ il n'y a pas d'opérations dupliquées que ce soit pour la communication ou pour la tolérance aux pannes.

$\beta_{10} = \emptyset \Rightarrow$ on ne modélise pas la quantité de mémoire nécessaire à une opération pour s'exécuter.

$\beta_{11} = \emptyset \Rightarrow$ il n'y a pas de relation d'exclusion d'exécution entre les opérations, c'est-à-dire de relation n'autorisant pas que telle et telle opérations ne s'exécutent sur le même processeur. Ce type de relation est utilisé dans le cas de la tolérance aux pannes.

$\beta_{12} = R_S \Rightarrow$ certaines opérations sont parfois contraintes à être exécutées sur tel ou tel processeur, c'est notamment le cas pour les entrées-sorties par exemple.

On suppose de plus que le modèle d'algorithme est déterministe, autrement dit qu'on peut associer à chaque opération une durée d'exécution sur un processeur et cette durée est définie de manière exacte et non en probabilité.

Contraintes et objectifs

Les trois premières contraintes γ_1,γ_2,γ_3 portent sur les contraintes que l'on cherche à respecter quand on fait du placement sans ordonnancement, limitation de la charge sur chaque processeur, équilibrage de la charge, donc ces contraintes n'existent pas pour notre problème. Il en est de même pour γ_4 et γ_6 car ils correspondent, d'une part à la minimisation de la somme des coûts d'exécution et de communication, ce qui est différent de la durée d'exécution du système, et à l'optimisation de la variance de la charge d'autre part, et aucun de ces deux phénomènes ne nous intéressent.

$\gamma_5 = TE2 \Rightarrow$ on cherche à minimiser la durée d'exécution du système. Pour être plus précis, comme on va le voir par la suite, on ne résout pas un problème d'optimisation mais un problème de recherche, c'est-à-dire que l'on cherche une solution respectant la contrainte temps réel du système et dès qu'on a trouvé une solution valide on s'arrête, on ne cherche pas à obtenir la solution optimale.

$\gamma_7 = MP \Rightarrow$ le nombre de processeurs de l'architecture est supposé fini.

Le tableau 2.2.1 résume les hypothèses que nous avons prises pour le problème d'ordonnancement. Ces hypothèses sont spécifiées quand cela est possible avec le formalisme de Graham et sinon elles sont ajoutées telles qu'elles à la suite du formalisme de Graham et sont précédées de la mention autres hypothèses.

Algorithme d'ordonnancement *statique*
Architecture $\alpha = R_p, H^-, C^-, A^-$ Autres hypothèses : modèle *déterministe*, *durée d'exécution nulle pour les communications intra-processeur*, *possibilité de recouvrement des calculs et des communications.* Algorithme $\beta = \tau_i, Com, <, R_S$ Autres hypothèses : modèle *déterministe*, Implantation Autres hypothèses : *routage des communications*, *distribution et ordonnancement des calculs*, *distribution et ordonnancement des communications.* Critères à optimiser $\gamma = TE, MP$

TAB. 2.2.1 – *Notre problème d'ordonnancement en utilisant la classification de Graham*

2.3 Présentation des problèmes de placement et d'ordonnancement

Les problèmes de placement et d'ordonnancement sont des problèmes d'optimisation combinatoire. Ils consistent en l'optimisation d'une fonction de coût (en général le temps d'exécution total), sous certains critères comme l'équilibrage de la charge de calcul, des communications, ou des critères portant sur la structure du graphe de l'algorithme.

2.3.1 Le placement

Le placement consiste à distribuer le graphe de l'algorithme sur le graphe de l'architecture en respectant les contraintes de dépendances de données symétriques et les contraintes liées aux durées d'exécution. Le graphe de l'algorithme et le graphe de l'architecture sont tous deux modélisés par des graphes non orientés.

Remarque 52 *En général, la plupart des méthodes de placement proposées dans la littérature utilisent des modèles de graphe non orienté pour modéliser le graphe de l'algorithme et le graphe matériel. Les dépendances de données entre opérations sont donc prises en compte, en revanche l'ordre partiel associé à l'algorithme ne l'est pas. C'est pourquoi, beaucoup de*

méthodes de placement ne peuvent être utilisables pour résoudre un problème d'ordonnancement.

2.3.2 L'ordonnancement

L'ordonnancement consiste à distribuer et à ordonnancer le graphe de l'algorithme sur le graphe de l'architecture en respectant les contraintes du graphe de l'algorithme liées à l'ordre partiel induit par les précédences et liées aux durées d'exécution de telle sorte que la durée d'exécution de l'application soit inférieure ou égale à la contrainte temps réel du système étudié. La formalisation de l'ordonnancement a été définie dans la première partie de la thèse. L'algorithme est modélisé par un hypergraphe orienté infiniment itéré et l'architecture est modélisée par un hypergraphe non orienté.

Remarque 53 *Le critère utilisé ici est la minimisation de la durée globale d'exécution de l'application car on se situe dans le cadre des applications temps réel mais d'autres critères (utilisation efficace de l'ensemble des ressources, respect du plus grand nombre de contraintes, minimisation du nombre d'interruptions,...) peuvent être utilisés.*

Le problème central d'ordonnancement est un problème d'ordonnancement sans contraintes de ressources, ou encore, ce qui revient au même, avec des contraintes de ressources illimitées. Les seules contraintes du problème central d'ordonnancement sont les contraintes temporelles induites par les arcs de dépendances de données du graphe de l'algorithme modélisant l'application que l'on souhaite ordonnancer.

2.3.3 Classifications des méthodes de résolution des problèmes d'ordonnancement

Il existe plusieurs manières de classifier les algorithmes d'ordonnancement. Nous allons en présenter deux, la première est la plus courante, elle consiste à classifier les méthodes de résolution suivant leur manière d'explorer l'espace des solutions. Plus précisément on se pose les questions suivantes : est-ce-que l'espace des solutions est entièrement exploré? Ou au contraire partiellement exploré? Ensuite, est-ce-que l'exploration partielle est effectuée "intelligemment"? ... La deuxième classification que nous allons présenter permet de classer les heuristiques selon les caractéristiques de l'implantation solution obtenue. Autrement dit, on ne cherche plus à savoir comment la solution a été obtenue mais plutôt quelles hypothèses sur les modèles d'algorithme, d'architecture et d'implantation il a été nécessaire de prendre pour arriver à une solution ayant les caractéristiques souhaitées. Cette dernière classification a l'avantage de préciser les caractéristiques (routage des communications, ...) de l'implantation que l'heuristique construit. Elle rejoint la classification de Graham présentée

à la section 2.2 *contexte de l'étude.*

Classification suivant l'exploration de l'espace des solutions

Une des manières les plus classiques de classifier les méthodes de résolution des problèmes d'ordonnancement est de distinguer les méthodes exactes des méthodes approchées. Les méthodes exactes sont des méthodes permettant de trouver la solution optimale d'un problème donné en faisant une énumération "intelligente" de l'espace des solutions. Autrement dit, elles trouvent toujours la solution exacte si on leur en laisse le temps. Ce sont des méthodes qui sont totalement inadaptées aux applications temps réel qui sont en général des problèmes de grande taille. En effet, prenons le cas d'un placement de 20 opérations sur 10 processeurs. Il y a 10^{20} placements possibles. En supposant grossièrement qu'un placement va prendre une instruction machine de l'ordre de la micro-seconde, il va falloir plusieurs siècles pour résoudre ce problème de manière optimale. Les méthodes approchées sont des méthodes permettant de trouver une solution approchée au problème posé. L'espace des solutions n'est pas entièrement visité. On trouvera une solution dite sous-optimale c'est-à-dire la meilleure relativement au critère choisi par l'utilisateur.

Autre classification suivant les caractéristiques de la solution

Certains algorithmes d'ordonnancement considèrent l'algorithme d'application indépendamment de l'architecture sur laquelle il sera implanté. Cela revient à résoudre le problème central de l'ordonnancement [11] et dans ce cas, les communications inter-processeur ne peuvent pas être modélisées et donc ce type d'algorithmes d'ordonnancement est adapté non pas aux systèmes distribués mais aux systèmes centralisés. D'autres algorithmes d'ordonnancement présupposent que l'architecture est constituée d'un ensemble de processeurs complètement connecté. Ces algorithmes distribuent les communications inter-processeur sur les liens mais ne les ordonnancent pas et ne prennent pas en compte le routage. Autrement dit, cela revient à modéliser l'architecture par un ensemble d'unités de calcul totalement connectées entre elles par des liens inter-processeur et cela suppose qu'un lien est toujours disponible pour accueillir une communication. Ahmad et Kwok dans [2] proposent de distinguer trois sortes d'algorithmes au sein de ce groupe : les BNP -Bounded Number Processors-, les UNC -Unbounded Number of Clusters- et les TDB -Task Duplication Based- . Les algorithmes BNP supposent que le nombre de processeurs est fini, que l'architecture est complètement connectée, et ils ne prennent en compte ni le routage ni l'ordonnancement des communications. Pour les algorithmes UNC, le nombre de processeurs est supposé infini, l'architecture est complètement connectée, et ni le routage ni l'ordonnancement des

communications inter-processeur ne sont effectués.

Remarque 54 *Les algorithmes de type TDB ont pour objectif de réduire les communications en allouant de manière redondante certaines tâches à plusieurs processeurs. Il existe différentes stratégies pour dupliquer les opérations du graphe. Certains algorithmes dupliquent seulement les prédécesseurs directs tandis que d'autres essaient de dupliquer tous les prédécesseurs possibles.*

Nous allons nous intéresser aux deux premiers types car nous n'utilisons pas la duplication des opérations comme mode de communication (cf. classification de Graham, on a choisi $\beta_9 = \emptyset$). Afin d'être plus réaliste, il est nécessaire de prendre en compte les délais de communication dans l'ordonnancement d'opérations sur un système multiprocesseur. Les algorithmes d'ordonnancement appartenant à la catégorie APN -Arbitrary Processors Network- décrite par Kwok et Ahmad dans [2] prennent en compte les caractéristiques spécifiques de l'architecture (caractéristiques telles que le nombre de processeurs ou la topologie du réseau supposée arbitraire). Ces algorithmes sont capables d'implanter les opérations de calcul sur les processeurs et soit (a) de router et distribuer les messages, appelés opérations de transfert dans la modélisation relationnelle, sur les média du réseau, soit (b) de router, distribuer et ordonnancer les messages sur les média. L'ordonnancement des messages peut dépendre de la stratégie de routage utilisée par le réseau.

2.4 Méthodes de résolution des problèmes d'ordonnancement

2.4.1 L'ordonnancement : un problème NP-difficile

Le problème consistant à trouver un ordonnancement d'un algorithme donné sur une architecture donnée ayant une durée inférieure ou égale à la contrainte temps réel du système est un problème de recherche reconnu comme un problème NP-difficile [56] dans sa forme la plus générale. Si, à chaque implantation solution construite lors de la composition des trois relations routage, distribution et ordonnancement de la modélisation relationnelle on associe une valeur, le problème d'ordonnancement qui consiste à rechercher parmi l'ensemble des solutions une solution de valeur optimale (minimale dans le cas où l'on cherche à minimiser la fonction durée d'exécution de l'algorithme d'application) devient un problème de recherche particulier : c'est un problème d'optimisation.

Le cadre de notre étude, comme on l'a vu jusqu'à présent, est le cadre des système temps réel. Ainsi, ce qui nous intéresse n'est pas d'avoir une durée d'exécution de l'algorithme d'application qui soit minimale mais plutôt qui respecte la contrainte temporelle temps réel du système. En effet, on a vu dans l'introduction de la thèse que la notion de temps réel n'est pas une notion portant sur la minimisation de la durée d'exécution du système mais plutôt

une notion de respect de délai du système qui interagit avec son environnement. Ainsi, notre problème d'ordonnancement n'est pas un problème d'optimisation au sens où l'on vient de le voir mais seulement un problème de recherche. On cherche une solution valide mais pas nécessairement optimale.

Définition 52 *Un problème est NP-difficile s'il existe un problème NP-complet se réduisant à ce problème par une réduction de Turing [11].*

Ce type de problème possède néanmoins, dans des conditions bien particulières des algorithmes d'ordonnancement polynômiaux. Par exemple, il existe un algorithme d'ordonnancement optimal dans [16] pour un graphe d'algorithme d'application ne comportant que des opérations de durée unitaire s'exécutant sur deux processeurs et sans communication.

2.4.2 Les méthodes exactes

Elles sont en général énumératives, c'est-à-dire que les algorithmes associés à ces méthodes, reposent sur une exploration (et comparaison) de toutes les solutions possibles. Ils fournissent une solution optimale au problème de placement mais ont une complexité exponentielle dans le pire des cas. Cependant, il est possible de réduire considérablement la complexité en moyenne grâce à l'emploi de stratégies adaptées. Les méthodes exactes peuvent être réparties en 3 classes principalement [19] :

1. les méthodes classiques liées à la théorie des graphes [30] :

   Nous allons présenter deux méthodes (potentiel-tâches et potentiel-étapes) permettant de trouver un ordonnancement optimal au problème central de l'ordonnancement [11]. Ces deux méthodes présupposent d'une part que les ressources soient illimitées et d'autre part qu'il n'y ait pas de communication entre les opérations.

   - Méthode du graphe potentiel-tâches

     Deux opérations fictives $\mathcal{I}$ et $\mathcal{O}$ de durée nulle sont rajoutées au graphe de l'algorithme (sans circuit) de départ. $\mathcal{I}$ et $\mathcal{O}$ vérifient les propriétés suivantes :

     Propriété 45

     - *L'opération $\mathcal{I}$ précède toutes les autres opérations de G_{al}. $\forall o_i \in O, \mathcal{I} < o_i$*
     - *L'opération $\mathcal{O}$ succède à toutes les autres opérations de G_{al}. $\forall o_i \in O, \mathcal{O} > o_i$*

     Le critère d'ordonnancement est ici la minimisation de la date de fin d'exécution de la dernière opération du graphe de l'algorithme. La méthode potentiel-tâches consiste à calculer dans un premier temps les dates de début au plus tôt $S(o_i)$ et au plus tard $s(o_i)$ de chaque opération o_i. Cela permet de distinguer les opérations dites critiques, i.e., situées sur le chemin critique du graphe étudié et donc qui ont

une marge de d'ordonnancement nulle. Autrement dit, si on tarde à ordonnancer ces opérations, la durée globale de l'ordonnancement s'en trouvera augmentée.

L'algorithme est l'algorithme de recherche du plus long chemin [30] :

Soit $\Pi(o_i)$, la longueur du chemin pour aller de $\mathcal{I}$ à o_i.

Soit $l_{o_i o_j}$, la durée d'exécution de o_i.

Soit O_{ordo}, l'ensemble des opérations implantées.

Soit O^C_{ordo}, l'ensemble des opérations non encore implantées. $O = O_{ordo} \cup O^C_{ordo}$

Alors :

(a) $\Pi(\mathcal{I}) = 0$ et $O_{ordo} = \{\mathcal{I}\}$

(b) Tant que $O^C_{ordo} \neq \emptyset$ faire :

Chercher un sommet $o_j \in O^C_{ordo}$ tq $\Gamma^{-1}_{o_j} \subset O_{ordo}$. (Autrement dit, trouver un sommet dont les prédécesseurs sont déjà implantés).

$\Pi(o_j) = \max\limits_{o_j \in \Gamma^{-1}_{o_i}} (\Pi(i) + l_{o_i o_j})$

$O_{ordo} \leftarrow O_{ordo} \cup o_j$

Fin de tant que

- Méthode du graphe potentiel-étapes

  La méthode du graphe potentiel-étapes (PERT)[1] consiste à calculer le dual du graphe de l'algorithme caractérisant notre application. Ainsi, les arcs du graphe potentiel-étapes sont les opérations du graphe de l'algorithme initial et ils sont donc valués par la durée des opérations. Les sommets sont les débuts et les fins des opérations.

- Conclusion sur ces deux méthodes

  Ce sont des méthodes qui donnent un ordonnancement optimal pour le critère de minimisation de la durée d'exécution. Néanmoins, elles semblent mal adaptées à notre cas car du fait qu'elles ne prennent pas en compte les communications, elles ne peuvent être considérées comme réalistes pour nous. Elles sont plus adaptées pour ordonnancer des opérations partiellement ordonnées mais sans dépendances de données comme celles qui composent la construction d'un bâtiment par exemple.

2. les méthodes de programmation mathématique :

   Ce type de méthodes est surtout utilisé pour résoudre des problèmes d'ordonnancement où les opérations n'ont pas de dépendances de données entre elles. Parmi les méthodes de programmation mathématique, on trouve les méthodes de programmation linéaire continu comme la méthode des points intérieurs [8], les méthodes de

1. PERT= Program Evaluation and Research Task

programmation linéaire en nombres entiers comme la relaxation lagrangienne [70] ou encore les méthodes de programmation dynamique.

3. les méthodes de séparation/évaluation (plus connu en anglais sous le nom de Branch&Bound [64]) :

   Ce sont des méthodes arborescentes qui consistent à placer progressivement les opérations sur les processeurs en explorant progressivement l'arbre de recherche décrivant, de manière exhaustive, toutes les solutions. En pratique, toutes les solutions possibles ne sont pas explicitement construites, des sous-ensembles entiers peuvent être abandonnés quand on peut être sûr qu'ils ne contiennent pas de solution optimale.

Les algorithmes d'ordonnancement implantant ces méthodes conduisent théoriquement à la solution optimale, mais sont très coûteux en pratique et donc utilisables uniquement pour résoudre des problèmes de petite taille. Pour des problèmes de grande taille, ce qui est le cas de la majeure partie des systèmes temps réel, la durée d'exécution de l'algorithme d'ordonnancement est trop longue. Les méthodes exactes sont donc mal adaptées à la résolution de systèmes temps réel. Il vaut mieux utiliser des heuristiques.

2.4.3 Les méthodes approchées ou heuristiques

Les méthodes approchées calculent une *implantation solution* de bonne qualité relativement au critère choisi. Une implantation solution est un graphe de type $G_S(.,.)$ défini dans le modèle d'implantation dans la première partie de la thèse. C'est donc un graphe dont la fermeture transitive de l'ordre partiel induit par les dépendances de données du graphe contient la fermeture transitive de l'ordre partiel initial. Les méthodes approchées peuvent fournir des algorithmes d'ordonnancement pouvant être rapides pour donner une solution sous-optimale. Avant de classifier les méthodes approchées, on va définir les notions de solution *partielle* et solution *complète* que l'on va utiliser par la suite.

Définition 53 *Une solution partielle est une implantation contenant un sous-ensemble d'opérations de calcul du graphe de l'algorithme initial, et une solution complète est une implantation contenant l'ensemble des opérations de calcul du graphe de l'algorithme initial.*

Parmi les méthodes approchées [67], on distingue :

1. les méthodes *gloutonnes* : ce sont des méthodes qui construisent une seule solution complète. A chaque étape de la méthode on complète une solution partielle en cherchant à faire le choix d'implantation le plus avantageux pour une opération de calcul donnée. Le choix d'implantation effectué à une étape donnée est définitif, on s'interdit donc de le remettre en cause au cours des étapes ultérieures. Il n'y a donc pas de retour-arrière

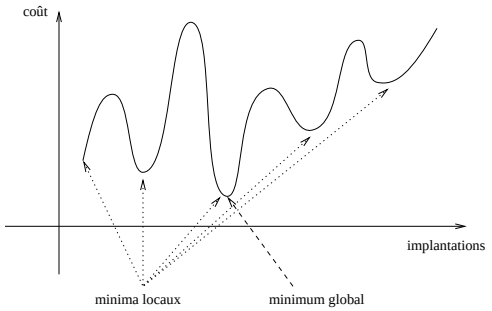


FIG. 2.4.1 – *Evolution du coût d'un ordonnancement*

(ou encore back-tracking). Le principe d'un algorithme d'ordonnancement glouton est donné dans la table 2.4.2.

Initialisation de l'implantation solution : solution partielle vide
`TantQue` l'implantation n'est pas une solution complète
♦ Sélection d'une opération de calcul et ordonnancement de cette opération et de ses éventuelles opérations de transfert associées
♦ Mise à jour de l'implantation solution
`FindeTantQue`

TAB. 2.4.2 – *Algorithme d'ordonnancement glouton*

2. les méthodes de *voisinage* : elles partent d'une solution initiale complète éventuellement calculée par une méthode gloutonne et elles cherchent à améliorer cette solution. Plus précisément, toute solution (une implantation choisie) est évaluée grâce à une fonction de coût. Ainsi, on peut construire la courbe décrivant les variations de la fonction de coût (cf. figure 2.4.1) en fonction de l'implantation associée. On observe une courbe possédant plusieurs minima locaux et un minimum global. Le but des méthodes de voisinage est de trouver ces minima. On distingue les méthodes :
   - de *recherche locale* encore appelées méthodes *non stochastiques*. Le principe de la recherche locale est d'améliorer une solution initiale en cherchant dans son voisinage une solution meilleure. Le retour-arrière et le Hill-Climbing sont des méthodes de recherche locale. Le principal inconvénient des algorithmes non stochastiques est de pouvoir générer des solutions non optimales qui correspondent à des optima locaux,
   - de *recherche globale* ou méthodes *stochastiques* ou encore *méta-heuristiques* qui permettent d'éviter de tomber dans un minimum local [63]. On s'autorise, à une itération donnée, avec une certaine probabilité, à choisir une solution moins

bonne que la précédente. Les méthodes Tabou, recuit simulé et les algorithmes génétiques appartiennent à cette dernière catégorie.

Le principe d'un algorithme de voisinage est donné à la table 2.4.3. La différence entre les méthodes de recherche locale et globale se situe à la fois au niveau de la construction d'une nouvelle implantation et à la fois dans le fait de retenir une implantation solution ou non. Si l'on applique une méthode de voisinage de type recherche locale, on risque d'atteindre un minimum local qui ne soit pas le minimum global. En effet, si la dernière solution trouvée est moins bonne que la précédente alors à coup sûr on ne la retient pas, alors que dans les méthodes de recherche globale, on s'autorise avec une certaine probabilité à accepter cette moins bonne solution, ceci pour éviter d'être piégé dans un minimum local. On peut alors espérer sortir d'un minimum local et parvenir à un autre minimum et ainsi de suite jusqu'à atteindre le minimum global.

Initialisation de l'implantation courante avec une implantation initiale
`TantQue` l'implantation courante ne convient pas
- Calcul d'une nouvelle implantation
- Mise à jour ou non de l'implantation courante avec la nouvelle implantation

`FindeTantQue`

TAB. 2.4.3 – *Algorithme d'ordonnancement de voisinage*

2.5 Importance du chemin critique

Beaucoup d'algorithmes d'ordonnancement utilisent le chemin critique pour décider comment placer les opérations. On verra par la suite que les opérations situées sur le chemin critique auront une très haute priorité. Nous allons tout d'abord définir le chemin critique et sa longueur et nous poursuivrons en donnant les différents moyens proposés dans la littérature pour identifier les opérations critiques.

Définition 54 *Le chemin critique d'un graphe d'algorithme G_{al} donné, est le plus long chemin de ce graphe. Pour les arcs et les opérations composant ce chemin, on parlera d'arcs critiques et d'opérations critiques.*

Cette définition suppose que l'on ait, au préalable, étiqueté le graphe afin de pouvoir calculer la longueur du chemin. L'étiquetage du graphe a été défini dans le chapitre 1 caractérisation. Plus précisément, une fois que l'on a caractérisé l'architecture cible, on associe une durée d'exécution à chaque opération de calcul de G_{al}. Avant que le graphe G_{al} ne soit

implanté, il n'y a pas d'opérations de transfert, donc les arcs sont étiquetés par le type et la quantité de données du transfert qu'ils modélisent.

Définition 55 *La longueur $\mathcal{R}$ du chemin critique est égale à :*

1. *la somme maximale des coûts de calculs et des coûts de communication le long d'un chemin allant d'une opération d'entrée à une opération de sortie,*
2. *la plus grande date de fin des opérations (de calcul et de transfert) à l'issue de l'ordonnancement.*

Avant que le graphe de l'algorithme ne soit implanté, on ne connaît pas la longueur du chemin critique. Par contre, on peut en calculer une estimation. On va définir les deux estimations les plus courantes : estimation *par défaut* et estimation *par excès*. Si on ne prend en compte que les coûts de calcul, on a une estimation de la longueur du chemin critique par défaut, en revanche si on prend en compte les coûts de calcul et les coûts de communication, on a une estimation par excès. Cette dernière estimation présuppose que les coûts des communication soient connus avant implantation, cela revient à dire que le coût de communication entre deux opérations dépendantes est constant et indépendant du nombre de liaisons utilisées pour effectuer le transfert. Cela veut dire que le routage n'est pas pris en compte, ou alors qu'il est pris en compte mais sans surcoût supplémentaire pour les communications routées sur plusieurs liens. Dans le chapitre Caractérisation, nous avons dit que nous prenions en compte le routage et le surcoût engendré par celui-ci au niveau des communications. Cela veut dire qu'avant ordonnancement, nous ne pouvons pas calculer d'estimation par excès de la longueur du chemin critique car le coût des communications n'est pas connu.

Soit $\mathcal{R}_{min}$, l'estimation par défaut de la longueur du chemin critique, soit $\mathcal{R}$, la longueur réelle du chemin critique après ordonnancement, et soit $\mathcal{R}_{max}$, l'estimation de la longueur du chemin critique par excès.

$\mathcal{R}_{min} =$ longueur maximale d'une opération d'entrée à une opération de sortie en ne prenant en compte que le coût des calculs.

$\mathcal{R}_{max} =$ longueur maximale d'une opération d'entrée à une opération de sortie en prenant en compte les durées de calcul et les durées des communications (présuppose un certain modèle d'architecture et d'implantation)

$$\mathcal{R}_{min} \leq \mathcal{R} \leq \mathcal{R}_{max}$$

La première estimation $\mathcal{R}_{min}$ revient à supposer que toutes les opérations dépendantes sont distribuées sur le même processeur, ce qui engendre une communication intra-processeur qui a un coût nul. En revanche, la seconde estimation $\mathcal{R}_{max}$ revient à supposer que toutes les opérations dépendantes sont distribuées sur des processeurs différents et donc chaque

dépendance engendre une communication inter-processeur qui a un coût constant pour une même dépendance.

Ainsi, si on a choisi l'estimation $\mathcal{R}_{min}$ comme approximation de la longueur du chemin critique, la longueur réelle ne peut être que supérieure à cette valeur et l'algorithme d'ordonnancement va avoir pour but de la faire augmenter le moins possible, tandis que dans le cas de l'estimation $\mathcal{R}_{max}$, l'algorithme d'ordonnancement va chercher à la faire diminuer le plus possible. Ainsi, on voit que d'une étape à une autre, la longueur du chemin critique va varier et une conséquence de cette variation de longueur est que ce ne sont pas toujours les mêmes opérations qui sont critiques. Donc il est intéressant de recalculer, à chaque étape, quelles sont les opérations critiques afin de savoir les opérations les plus urgentes à implanter. En effet, si on tarde à implanter une opération critique, la longueur d'ordonnancement risque d'augmenter, ce que l'on ne souhaite pas.

Nous allons définir au chapitre suivant des calculs de longueurs de chemin associées à chaque opération du graphe de l'algorithme. Ces opérations critiques sont identifiées par exemple en fonction d'informations définies plus loin à la page 106 telles que :

- la plus grande somme de *tlevel* et *blevel*,
- la même $AEST$ et $ALST$,
- la même $ASAP$ et $ALAP$, autrement dit une mobilité nulle,
- une pression d'ordonnancement nulle.

2.6 Méthodes gloutonnes

Comme on l'a vu précédemment dans la première partie de la thèse, l'implantation d'un algorithme sur une architecture consiste en, d'une part la distribution de l'algorithme sur une architecture, et d'autre part en l'ordonnancement de l'algorithme sur l'architecture. Si les deux problèmes sont résolus simultanément, on parle de méthode de résolution *one-stage* [88] et dans le cas où les problèmes sont résolus successivement on parle de méthodes *multiple-stage*. Les méthodes codées sous forme d'algorithmes de liste sont des méthodes one-stage tandis que les méthodes codées sous forme d'algorithmes de clustering sont des méthodes multiple-stage. Les algorithmes de liste et de clustering sont tous deux des algorithmes gloutons et les algorithmes de clustering reposent également souvent sur des principes d'algorithmes de liste comme on va le voir. Après avoir présenté les principes d'algorithmes de liste, puis d'algorithmes de clustering, nous allons donner quelques exemples de règles qui permettent de distribuer (et ou) d'ordonnancer les opérations sur les ressources puis nous finirons par quelques exemples d'algorithmes gloutons de liste et de clustering qui sont classés en utilisant la classification de Kwok et Ahmad présentée page 95.

2.6.1 Principe des algorithmes de liste

Dans un premier temps, une liste d'opérations est établie en utilisant une règle de priorité qui permet d'ordonner les opérations du graphe de l'algorithme par priorité décroissante.

Remarque 55 *La liste peut être qualifiée de* statique *ou de* dynamique. *Elle est dite statique si elle est inchangée au cours du processus d'ordonnancement autrement dit si on la construit une fois pour toutes au début du processus. En revanche, elle est dynamique si la priorité associée à une opération non encore ordonnancée peut varier d'une étape de l'heuristique d'ordonnancement à une autre étape. Le fait de recalculer les priorités des opérations non encore ordonnancées permet de connaître à chaque étape les opérations critiques. En effet, une opération peut être qualifiée de critique à une étape donnée et ne plus l'être à une étape suivante.*

Initialisation de la liste avec l'ensemble des opérations de calcul du graphe initial. Tri de ces opérations en utilisant la règle de priorité choisie `TantQue` la liste n'est pas vide ♦ Ordonnancer l'opération de calcul la plus prioritaire et ses opérations de transfert associées sur le *meilleur* processeur ♦ Retirer l'opération ordonnancée de la liste `FindeTantQue`

TAB. 2.6.4 – *Algorithme d'ordonnancement de liste statique*

Une fois que l'on s'est fixé la règle de priorité, l'algorithme d'ordonnancement réalise l'ordonnancement de l'algorithme d'application en commençant par implanter les opérations les plus prioritaires. Le tableau 2.6.4 donne le principe d'un algorithme de liste statique et le tableau 2.6.5 donne le principe d'un algorithme de liste dynamique. Le choix de la règle de priorité est déterminant pour la qualité de l'ordonnancement. En effet, une "mauvaise" règle de priorité peut faire que des opérations moins urgentes à ordonnancer que d'autres, soient ordonnancées avant les opérations urgentes.

Quand le contraire ne sera pas explicité, la liste sera initialisée avec l'ensemble des opérations sans prédécesseur du graphe de l'algorithme d'application.

Choisir le *meilleur* processeur pour une opération donnée dans un algorithme de liste dépend de la stratégie d'ordonnancement. On va voir dans la section suivante quelques exemples de choix de processeurs.

Initialisation de la liste.

Ordonner ces opérations en utilisant la règle de priorité choisie

`TantQue` la liste n'est pas vide

- Ordonnancer l'opération de calcul la plus prioritaire et ses opérations de transfert associées sur le *meilleur* processeur
- Ajouter les successeurs ordonnançables de l'opération ordonnancée, triés par priorité décroissante à la fin de la liste
- Recalculer ou calculer la priorité de chaque opération de la liste
- Retirer l'opération ordonnancée de la liste
- Réordonner éventuellement les opérations de la liste en utilisant la règle de priorité choisie

`FindeTantQue`

TAB. 2.6.5 – *Algorithme d'ordonnancement de liste dynamique*

2.6.2 Principe des algorithmes de "clustering"

Ce type d'algorithmes d'ordonnancement présuppose un modèle d'architecture composé d'un nombre de processeurs aussi grand que l'on veut et complètement connectés entre eux. Le modèle d'implantation associé à ces algorithmes est la distribution et l'ordonnancement des calculs et la distribution mais pas l'ordonnancement des communications. Les algorithmes réalisent dans un premier temps un partitionnement du graphe de l'algorithme en sous-graphes, appelés *clusters* ou *tas*, disjoints. Comme dans le cas d'un algorithme de liste, le clustering affecte une priorité à chaque opération. Plus précisément, chaque opération est supposée être un cluster unitaire et à chaque étape de l'algorithme de clustering, on cherche à assembler l'opération la plus prioritaire à un cluster bien choisi de telle sorte que la longueur totale de l'ordonnancement diminue. Le partitionnement peut être effectué de différentes manières, il est en tout cas construit de manière gloutonne, c'est-à-dire qu'à chaque étape du partitionnement, une opération est distribuée sur un cluster. On peut à chaque étape de l'algorithme de clustering avoir un ordre total sur chacun des clusters mais la différence avec l'algorithme de liste est que cet ordre total n'est pas fixe, c'est-à-dire qu'une opération candidate à la distribution sur un cluster peut être intercalée entre deux opérations déjà distribuées pourvu que l'ordre partiel du graphe initial soit conservé. Ou bien, on peut avoir sur chacun des clusters un ordre partiel et cet ordre partiel est renforcé seulement à l'issue de la distribution de tout le graphe de l'algorithme. Ainsi, l'ordonnancement d'une opération n'est véritablement effectué qu'à l'issue du processus d'ordonnancement et c'est ce qui fait la plus grande différence avec les algorithmes de liste définis précédemment. En

effet, la date de début d'une opération sur un cluster n'est pas fixée tant que le processus d'ordonnancement de toutes les opérations n'est pas achevé. A l'issue de l'algorithme de clustering, chaque cluster est distribué et ordonnancé sur un processeur et les dépendances entre les clusters deviennent des communications inter-processeur et sont distribuées mais pas ordonnancées sur les liens inter-processeur. Si le nombre de clusters est supérieur au nombre de processeurs disponibles, un assemblage est effectué jusqu'à avoir un nombre de clusters identiques au nombre de processeurs.

2.6.3 Règles de priorité

Voici des exemples de règles de priorité :

♦ $S(o_i)$: date de début au-plus-tôt de l'opération o_i, encore appelée $ASAP(o_i)$ As-Soon-As-Possible(o_i).

♦ $s(o_i)$: date de début au-plus-tard de o_i encore appelée $ALAP(o_i)$ As-Late-As-Possible(o_i) ou $LPST(o_i)$ Latest-Possible-Start-Time(o_i). La valeur de cette date est bornée par la longueur du chemin critique.

♦ $static\_level(o_i)$ [69] : niveau statique de o_i. C'est la longueur maximale entre o_i et une opération de sortie, durée de o_i incluse en supposant que les opérations dépendantes sont distribuées sur le même processeur. Cela revient à ne prendre en compte que les coûts de calcul. Cette valeur est égale à la date de début au-plus-tard (définie depuis la fin) et notée $\bar{s}$ que l'on va utiliser au chapitre suivant sur l'optimisation de la latence.

♦ $tlevel(o_i)$ [73] : niveau t de o_i et $blevel(o_i)$: niveau b de o_i. Le niveau t de o_i est la longueur maximale entre une opération d'entrée et o_i, la durée de o_i n'étant pas prise en compte. Le niveau b de o_i est la longueur maximale entre o_i et une opération de sortie, la durée de o_i étant prise en compte. Ces deux niveaux prennent en compte les durées des opérations mais aussi les durées des communications. Plus précisément, ces niveaux sont calculés en supposant que toutes les opérations dépendantes sont distribuées sur des processeurs différents et donc chaque dépendance engendre une communication qui a un certain coût et ce coût est pris en compte. Ces niveaux sont connus si on suppose une architecture complètement connectée et pas de routage, et ces niveaux ne sont pas connus si l'heuristique prend en compte le surcoût dû au routage des éventuelles communications inter-processeur.

♦ $Prio(o_i) = (tlevel + blevel)(o_i)$: priorité de o_i.

♦ $DL(o_i,p_j) = static\_level(o_i) - S(o_i,p_j)$: niveau dynamique de o_i sur p_j

♦ $DL(o_i) = (static\_level - tlevel)(o_i)$: niveau dynamique de o_i.

♦ $\sigma(o_i,p_j) = S(o_i,p_j) + \bar{s}(o_i) - \mathcal{R}$ [54] : pression d'ordonnancement de o_i sur p_j sachant que $\mathcal{R}$ est la longueur du chemin critique. Cela peut encore s'écrire : $\sigma(o_i,p_j) = S(o_i,p_j) +$

	S_{def}	S_{exces}	s_{def}	s_{exces}	*static _level*	*tlevel*	*blevel*	*Prio*	*DL*	M_{def}	M_{exces}
o_1	0	0	0	0	2.5	0	12.5	12.5	2.5	0	0
o_2	1	6	3.5	6	1.5	6	6.5	12.5	-4.5	2.5	0
o_3	1	1.5	4.5	11.5	0.5	1.5	2.5	4	-1	3.5	10
o_4	1	3	1	8.5	0	3	0	3	-3	0	5.5
o_5	2.5	12.5	5	12.5	0	12.5	0	12.5	-12.5	2.5	0

TAB. 2.6.6 – *Calcul des valeurs des priorités associées au graphe de l'algorithme de la figure 2.6.2*

$static\_level(o_i) - \mathcal{R}$

♦ $M(o_i) = (ALAP - ASAP)(o_i)$: mobilité, flexibilité, ou encore marge d'ordonnancement de o_i, encore notée $F(o_i)$.

♦ $MR(o_i) = \frac{M(o_i)}{\Delta(o_i)}$: mobilité relative de o_i.

♦ $DCP(o_i,p_j) = s(o_i,p_j) - S(o_i,p_j)$: différence entre date de début au-plus-tard et date de début au-plus-tôt de o_i sur p_j. $s(o_i,p_j)$ est encore appelée $ALST(o_i)$ Absolute Latest Start Time(o_i). $S(o_i,p_j)$ est encore appelée $AEST(o_i)$ Absolute Earliest Start Time(o_i).

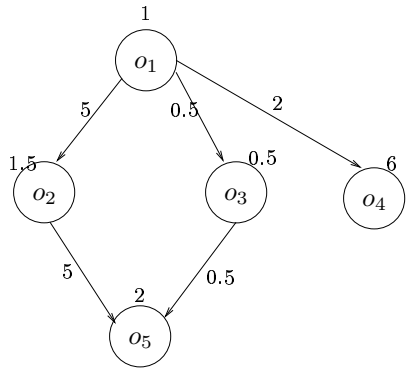


FIG. 2.6.2 – *Graphe d'algorithme étiqueté*

Ainsi, si on décide de trier les opérations du graphe de la figure 2.6.2 par *static_level* décroissant, la liste des opérations de la plus prioritaire à la moins prioritaire est o_1,o_2,o_3,o_4,o_5 d'après la table 2.6.6. De même, si on trie la liste par *Prio* décroissante, cela donne la liste suivante o_1,o_2,o_5,o_3,o_4.

2.6.4 Algorithmes gloutons

Avant de donner des exemples d'algorithmes d'ordonnancement de liste, nous allons définir des fonctions qui permettent de sélectionner le *meilleur* processeur pour une opération donnée.

- ♦ $S(o_i,p_j)$: date de début de o_i sur le processeur p_j, encore appelée $ST(o_i)$ Start Time(o_i).
- ♦ $FT(o_i,p_j)$: date de fin de o_i sur p_j, encore appelée $FT(o_i)$ Finish Time(o_i).

Algorithmes de liste

BNP : nombre de processeurs limité

1. ETF (Earliest Task First) [41, 15] L'opération la plus prioritaire est celle qui a la plus petite date de début. C'est un algorithme statique. Quand deux opérations ont la même date, on choisit celle qui a le *static_level* le plus élevé. Soit $\mathcal{R}_{ETF}$, la longueur du chemin critique calculée par l'algorithme ETF. Soit Λ_G, la somme maximale, sur tous les chemins du graphe de l'algorithme implanté, des durées de communication. La longueur du chemin critique de l'implantation solution, calculée par ETF, est bornée par [41] :
$$\mathcal{R}_{ETF} \leq (2 - \frac{1}{m})\mathcal{R}_{min} + \Lambda_G$$
où m est le nombre de processeurs.
2. DLS (Dynamic Level Scheduling) L'algorithme DLS [73] est un algorithme de liste dynamique qui trie les opérations par niveau dynamique (DL(.,.) précédemment défini) décroissant. Autrement dit, à chaque étape de l'ordonnancement, l'algorithme calcule le DL de chaque opération de la liste, sur tous les processeurs. La paire (opération,processeur) choisie est celle qui a le DL le plus élevé. On remarque que pour cet algorithme le meilleur processeur pour une opération donnée est choisi avec la règle de priorité et non avec une autre fonction comme c'est le cas pour d'autres algorithmes d'ordonnancement.

APN : prise en compte de la topologie du réseau

1. L'algorithme DLS (Dynamic Level Scheduling)
Cet algorithme appartient à l'ensemble BNP défini précédemment mais peut également appartenir à cette catégorie si le routage des messages peut être supporté par l'utilisateur.
2. L'algorithme BU (Bottom Up) L'algorithme BU [60] distribue et ordonnance les opérations critiques sur un même processeur et distribue et ordonnance les opérations non

critiques sur les autres processeurs, tout en optimisant l'équilibrage de la charge sur chacun de ces processeurs. C'est un algorithme statique.

3. L'algorithme MH (Mapping Heuristic)
   Cet algorithme [69] a une règle de priorité statique. La liste des opérations est triée par ordre décroissant de *static_level*. L'opération la plus prioritaire est ordonnancée sur le processeur donnant la plus petite date de début. Les successeurs ordonnançables de l'opération, ordonnés par priorité décroissante, sont ajoutés à la fin de la liste, et la liste n'est pas retriée, ceci pour pouvoir respecter les précédences du graphe de l'algorithme initial.

4. L'algorithme BSA (Bubble Scheduling and Allocation)
   Le principe de l'algorithme BSA [46, 47] est le suivant :

   (a) *Injection sérielle*
   Le processus d'injection sérielle consiste dans un premier temps à séquentialiser toutes les opérations sur le même processeur, appelé processeur pivot ; c'est celui qui possède le plus de liens inter-processeur.

   Définition 56 *Les opérations In-Branch (IBN) sont les opérations à partir desquelles il existe un chemin permettant d'atteindre une opération critique, et les opérations Out-Branch (OBN) sont les opérations du graphe qui ne sont ni critiques ni IBN.*

   La séquentialisation des opérations est effectuée en distribuant d'abord les opérations critiques, puis les opérations IBN et enfin les opérations OBN.

   (b) *Migration des opérations*
   Ensuite, la migration de certaines opérations vers d'autres processeurs est organisée afin de diminuer la longueur de l'ordonnancement.

   Définition 57 *La DAT - Data Arrival Time- d'une opération est la date à laquelle le dernier message de ses opérations parents lui parvient. Cette date prend en compte les communications inter-processeur qui sont routées et ordonnancées sur les liens.*

   L'opération sélectionnée pour migrer est celle qui a sa DAT supérieure à sa date de début sur le processeur pivot. Le processeur choisi pour cette opération est celui qui réduit le plus sa date de début. Le processus s'arrête quand toutes les opérations distribuées sur le processeur pivot ont été étudiées.

Algorithmes de clustering

UNC : nombre illimité de tas

1. L'algorithme DSC -Dominant Sequence Clustering
   Gérasoulis et Yang ont développé dans [87] l'algorithme DSC. Cet algorithme est inspiré de celui de Sarkar EZ -Edge Zeroing- cité dans [2, 87]. La complexité de ces algorithmes est de $O(n_d(n_d + n_o))$, où n_d est le nombre d'arcs du graphe de l'algorithme et n_o le nombre de ses sommets. L'algorithme cherche à distribuer, sur le même processeur, les opérations qui donnent lieu à la communication la plus importante, de telle sorte que la longueur partielle de l'ordonnancement n'augmente pas. La liste est composée des opérations *libres*, c'est-à-dire celles dont les prédécesseurs sont déjà implantés. Les opérations libres sont ordonnées par $Prio(.)$ (somme de *blevel* et *tlevel*) décroissant. La description de l'algorithme est donnée dans la table 2.6.7.

Notations
Soit OE, l'ensemble des Opérations Examinées. Soit ONE, l'ensemble des Opérations Non encore Examinées.
Algorithme
Calcul du *blevel* pour tous les nœuds et $tlevel = 0$ pour les nœuds d'entrée. Chaque opération est considérée comme non examinée et est vue comme un tas unitaire. `TantQue` toutes les opérations n'ont pas été examinées ♦ Trouver l'opération *libre* la plus prioritaire parmi les ONE ♦ Assembler si possible cette opération au cluster contenant un de ces prédécesseurs de telle sorte que le *tlevel* de ce nœud diminue le plus. Si toutes les mises à 0 possibles augmentent le *tlevel*, alors le nœud sélectionné forme un cluster unitaire ♦ Mise à jour des priorités des successeurs ordonnançables de l'opération précédente ♦ Mise à jour de ONE, OE `FindeTantQue`

TAB. 2.6.7 – *Algorithme de clustering DSC*

2. L'algorithme DCP (Dynamic Critical Path)
   L'algorithme DCP [48] est un algorithme de liste dynamique. L'ordonnancement de chaque processeur évolue à chaque étape et les dates de début des opérations ne sont pas fixées tant que toutes les opérations n'ont pas été implantées. Les opérations sont simplement regroupées ensemble en "cluster" selon un ordre total construit à chaque étape de l'ordonnancement. Des arcs de dépendances d'ordonnancement peuvent être

ajoutés sur certains clusters pour garantir la linéarité sur chacun des processeurs. Le fait que les dates ne soient pas fixées permet d'insérer une opération prise en considération dans une étape ultérieure, dans un intervalle de temps plus tôt s'il y a "assez de place" entre les opérations. Pour respecter les contraintes de précédence, une opération ne peut pas être insérée dans un intervalle de temps après (resp. avant) une opération successeur (resp. prédécesseur) ordonnancée. Tous les processeurs ne sont pas systématiquement examinés : seuls sont regardés ceux qui ont des opérations qui communiquent avec l'opération à implanter.

La liste des candidats à l'implantation est constituée des opérations appartenant au DCP -Dynamic Critical Path-. Ce sont les opérations pour lesquelles DCP(.) (défini précédemment) est nul. Ces opérations n'ont pas obligatoirement tous leurs prédécesseurs implantés.

Sélection du processeur

Soit o_i, l'opération que l'on cherche à implanter. Soit o_e, le successeur de o_i qui a le plus petit écart entre AEST et ALST. o_i est ordonnancé sur le processeur p_j qui donne la somme suivante $S(o_i,p_j) + S(o_e,p_j)$ la plus faible. La description de l'algorithme est donnée à la table 2.6.8.

Notations
Soit OE, l'ensemble des Opérations Examinées Soit ONE, l'ensemble des Opérations Non encore Examinées
Algorithme
Calcul de AEST et ALST pour toutes les opérations `TantQue` toutes les opérations n'ont pas été examinées ♦ Trouver l'opération *libre* la plus prioritaire parmi les ONE ♦ Sélectionner le *meilleur* processeur ♦ Mise à jour des priorités de toutes les opérations ♦ Mise à jour de ONE, OE `FindeTantQue`

TAB. 2.6.8 – *Algorithme de clustering DCP*

3. L'algorithme LC (Linear Clustering)

   Cet algorithme est un algorithme dynamique [43]. Il assemble de manière itérative les opérations pour former un seul tas sur le chemin critique. Les opérations assemblées sont supprimées de la liste et le processus d'assemblage est répété.

4. L'algorithme MD (Mobility Directed)

L'algorithme MD [85] est un algorithme ayant une règle de priorité dynamique. L'opération qui a la plus petite mobilité relative est sélectionnée et cette opération est ordonnancée sur le processeur qui a un intervalle de temps libre suffisant pour contenir la durée d'exécution de cette opération. On constate que cet algorithme ne cherche pas à minimiser la date de début des opérations. Cette stratégie peut dégrader de manière significative la longueur finale de l'ordonnancement. Une fois que l'opération est ordonnancée, toutes les mobilités relatives des opérations figurant encore dans la liste sont mises à jour.

2.6.5 Évaluation de la qualité de l'implantation solution construite

Il existe plusieurs façons d'évaluer une implantation. L'implantation construite peut être évaluée comme implantation optimale ou encore comme étant à une certaine distance de l'implantation optimale calculée exactement ou empiriquement.

Implantation optimale

Une première évaluation repose sur l'optimalité de certains algorithmes d'ordonnancement pour certaines structures d'algorithme d'application ou d'architecture.

Théorème 1 *En supposant que les opérations sont toutes de durée unitaire et que les communications ont des durées nulles, il existe une implantation de durée minimale [40] obtenue par une heuristique de liste, si le graphe d'algorithme est une anti-arborescence, et si la règle de priorité consiste à ordonner les opérations par blevel décroissant.*

Théorème 2 *Si le graphe de l'algorithme a une structure quelconque et que le graphe de l'architecture est composé de deux processeurs, alors il existe un algorithme d'ordonnancement optimal [16] qui utilise comme règle de priorité le blevel et qui classe ensuite les opérations équivalentes en fonction des successeurs.*

Théorème 3 *Dans le cas où le graphe de l'algorithme est une anti-arborescence et où les coûts de calcul et de communication sont supposées de durée unitaire, il existe des algorithmes d'ordonnancement [81] qui donnent la solution optimale.*

L'inconvénient de cette évaluation, comme on vient de le voir dans les théorèmes précédents, est qu'il faut supposer des structures particulières de graphes d'algorithme et d'architecture et un étiquetage du graphe de l'algorithme également particulier. Ainsi, si on a un graphe avec une structure quelconque et si on utilise une de ces heuristiques qui construit une solution optimale pour un graphe d'algorithme particulier, on ne pourra rien dire sur la qualité de l'implantation associée à ce graphe.

Distance exacte par rapport à l'implantation optimale

En supposant un modèle d'algorithme avec des coûts de calcul quelconques et des coûts de communication nuls, Coffman [17] a montré le théorème suivant :

Théorème 4 *Tout algorithme de liste construit une implantation solution qui se situe à un facteur inférieur ou égal à* $2 - \frac{1}{n_P}$ *de la solution optimale dans le cas où l'architecture est composée de* n_P *processeurs.*

$$\mathcal{R} \leq (2 - \frac{1}{n_P})\mathcal{R}^*$$

$\mathcal{R}^*$ *représentant la longueur du chemin critique dans le cas d'un ordonnancement optimal.*

Cela revient à dire que tout algorithme d'ordonnancement de liste qui ne prend pas en compte les communications inter-processeur se situe à 50 % de l'implantation optimale dans le pire des cas. Ce type de majoration n'est plus valable quand on prend en compte les durées des communications.

Théorème 5 *En présence de coûts de calcul et de communication unitaires, la longueur du chemin critique* $\mathcal{R}$ *calculée par une heuristique de liste est majorée par la valeur suivante : (cf. [68])*

$$\mathcal{R} \leq (3 - \frac{2}{n_P})\mathcal{R}^* - (1 - \frac{1}{n_P})$$

$\mathcal{R}^*$ *représentant la longueur du chemin critique dans le cas d'un ordonnancement optimal et* n_P *étant le nombre de processeurs qui composent l'architecture et* n_P *étant supposé supérieur ou égal à 3.*

Ainsi, si on a un grand nombre de processeurs, la longueur du chemin critique calculée par une heuristique est dans le pire des cas égale à trois fois la longueur optimale, ce qui n'est pas une très bonne performance. Les inconvénients de ce type d'évaluation sont, d'une part que l'on suppose un étiquetage particulier du graphe de l'algorithme et, d'autre part que l'on évalue toujours la plus mauvaise implantation que l'heuristique pourrait construire pour un graphe donné par rapport à l'implantation optimale. Il se peut très bien en effet, que l'heuristique ne construise jamais de telles mauvaises solutions.

Distance empirique par rapport à l'implantation optimale

Pour remédier à ce problème, Adam et al. dans [1] comparent de manière empirique les longueurs d'ordonnancement obtenues avec telle ou telle heuristique pour un graphe d'algorithme d'application donné. En supposant des coûts de communication nuls, ils ont montré de manière empirique, que l'algorithme d'ordonnancement de liste reposant sur le chemin critique se situe à 5 % de la solution optimale dans 90% des cas.

Cette technique empirique permet d'évaluer une implantation d'un graphe ayant une structure et des coûts de calcul quelconques. Ainsi, plutôt que d'évaluer la qualité d'une heuristique en évaluant à combien se trouve la solution calculée par l'heuristique de la solution optimale dans le pire des cas, on peut tester l'heuristique sur un grand nombre d'exemples générés au hasard et regarder les performances de l'heuristique sur ces exemples.

2.7 Méthodes de voisinage

2.7.1 Méthodes de recherche locale

Le retour-arrière

Le retour-arrière, encore appelé *back-tracking* est une méthode de recherche locale qui part d'une solution complète et qui cherche à améliorer cette solution complète. La première implantation est en général calculée par une méthode gloutonne qui peut être implantée sous forme d'algorithme de liste. Si, au cours d'une des étapes de l'heuristique de liste, la fonction de coût utilisée pour ordonner la liste n'a pas pu départager deux opérations (cf. définition 58), l'heuristique étant gloutonne, un choix parmi ces opérations est fait de manière arbitraire.

Définition 58 *Soit f une fonction de coût. On dit que deux opérations o'_i et o'_j ne peuvent être départagées par la fonction de coût f si et seulement si :*

$$|f(o'_i) - f(o'_j)| \leq \epsilon$$

ϵ étant donné par l'utilisateur et proche de zéro.

Le retour-arrière permet de revenir sur de telles étapes, de faire un autre choix et d'évaluer une nouvelle implantation solution. En effet, une opération mal distribuée peut détériorer la qualité de l'implantation obtenue, et un retour-arrière permet d'éviter cette détérioration. Il existe plusieurs niveaux de back-tracking. Tout d'abord, il se peut qu'il y ait plusieurs étapes lors de la construction de la première implantation pour lesquelles il y ait des opérations équivalentes, de plus en recalculant une nouvelle solution, on peut à nouveau avoir des opérations équivalentes... Pour éviter l'explosion combinatoire, différentes stratégies peuvent être retenues : ou bien on peut décider d'explorer l'ensemble de toutes les solutions que l'on peut construire en retournant en arrière jusqu'à une profondeur donnée, ou bien, étant donnée l'implantation initiale, on peut calculer une seule solution sans retour-arrière, par opération équivalente détectée sur la première solution. Cette dernière stratégie a l'avantage de pouvoir dénombrer exactement le nombre de solutions implantées et évite donc l'explosion combinatoire et si une seule opération est mal distribuée ce type de retour-arrière suffit.

Nous verrons dans le chapitre suivant la stratégie que nous avons utilisée pour notre heuristique et nous donnerons les algorithmes associés.

FAST

Les opérations du chemin critique ont la plus haute priorité. Cependant, elles ne peuvent être implantées tant que leurs prédécesseurs ne l'ont pas été. Pour remédier à ce problème Kwok, Ahmad et Gu proposent dans [49] de séparer les opérations du graphe de l'algorithme en deux catégories : les opérations IBN et les opérations OBN définies page 109. Les IBNs d'une opération critique donnée vont être implantées dans la liste avant celle-ci, tandis que les OBNs vont figurer en bout de liste car ce sont les opérations les moins urgentes à implanter. De plus, pour ordonner les IBNs entre elles, les opérations sont ordonnées dans l'ordre décroissant de leur *blevel*.

Une fois la liste obtenue, l'opération prête est ordonnancée sur le premier processeur disponible.

Dans un problème d'ordonnancement, une méthode pour améliorer la longueur de l'ordonnancement est de transférer une opération dit *bloquante* d'un processeur à un autre.

Définition 59 *Une opération est dite bloquante si lorsqu'on l'enlève d'un processeur, les opérations qui lui succèdent sur ce même processeur peuvent commencer plus tôt.*

En particulier, il est intéressant de détecter les opérations qui bloquent les opérations critiques. Dans cette approche, une liste d'opérations susceptibles de bloquer les opérations critiques est élaborée de manière statique, autrement dit avant le début du processus d'ordonnancement. Cette liste correspond aux opérations IBN et OBN précédemment définies. Ainsi, ces opérations bloquantes vont constituer le voisinage que le processus de recherche va explorer.

2.7.2 Méthodes de recherche globale

Parmi les méthodes de recherche globale classiques on distingue la méthode du recuit simulé [19, 67], la méthode Tabou [67] et les méthodes reposant sur des algorithmes génétiques [67, 22]. Nous allons présenter rapidement les principes du recuit simulé et de la méthode Tabou.

Le recuit simulé

Le recuit simulé est une méthode stochastique inventée en 1983 par les physiciens Kirkpatrick, Gelatt et Vecchi. Elle repose sur le principe physique du recuit : « un métal qui est

refroidi lentement aura une structure très ordonnée et cette structure correspond à l'énergie minimale du métal» . Inversement, un métal que l'on refroidit trop vite aura une structure peu ordonnée, autrement dit présentera des défauts. La méthode du recuit simulé [67, 19] consiste à transposer ce procédé à la résolution d'un problème d'optimisation combinatoire. Elle part d'une solution réalisable. A chaque itération, on tire au sort une configuration voisine et on évalue son coût. S'il est meilleur que celui de la solution courante, alors on garde cette solution, sinon on calcule une probabilité d'acceptation. On choisit de rejeter ou d'accepter la nouvelle configuration (même si elle dégrade le coût) en réalisant un tirage aléatoire. L'algorithme du recuit simulé est donné à la table 2.7.9. Ainsi cette méthode permet de sortir d'un minimum local afin d'atteindre éventuellement le minimum global. Les résultats obtenus avec ce type de calcul sont satisfaisants même si l'exécution est gourmande en temps de calcul. L'avantage du recuit simulé est qu'il peut être appliqué à une grande variété de problèmes d'optimisation combinatoire.

La méthode Tabou

Contrairement à la méthode du recuit simulé, la méthode Tabou [67] ne possède aucun caractère aléatoire. A chaque itération, le voisinage de la solution examinée est totalement visité et on choisit parmi ce voisinage une solution meilleure. Pour éviter de tomber dans des pièges correspondant à des minima locaux, on utilise une liste tabou pour stocker les valeurs pièges.

2.8 Conclusion

Les heuristiques de liste basées sur le calcul du chemin critique sont les mieux adaptées pour le prototypage rapide optimisé tout en permettant de raccourcir le cycle de développement dans lequel il est incontournable d'essayer plusieurs versions de l'algorithme et de l'architecture. L'inconvénient de ces méthodes est qu'elles font des optimisations locales à chaque itération de l'algorithme d'ordonnancement et on n'est pas certain sauf dans des cas particuliers (structures de graphes particulières, étiquetage particulier) d'obtenir un minimum global. En revanche les méthodes de voisinage de type tabou ou recuit simulé, plus difficiles à mettre en œuvre, en particulier à cause du réglage des paramètres, fournissent des meilleurs résultats au prix d'un temps d'exécution de l'algorithme d'ordonnancement plus long. Une étude a été menée par L. Phelippeau-Gelineau dans sa thèse [67] sur les résultats obtenus par des méthodes de liste et tabou. Elle a comparé ces méthodes en générant des graphes disposant de structures particulières ou quelconques. Ses conclusions à l'issue de ces expérimentations sont que les heuristiques de liste basées sur le calcul du chemin critique

Notations
Soit s, une solution et $V(s)$, le voisinage de cette solution. Soit T, la température. Soit $f(s)$, le coût de la solution s. Soit $\Delta(f)$, la variation de coût entre deux solutions s et s'. $\Delta(f) = f(s') - f(s)$. Soit $N_{maxiter}$, le nombre maximum d'itérations. Soit ϵ, le seuil de température qui fait stopper l'algorithme.
Algorithme
Calcul de la solution initiale s. $N_{maxiter}$ =..., ϵ =..., $n_{iter} = 1$ `TantQue` $n_{iter} \neq N_{maxiter}$ `ou` $T \neq \epsilon$ ♦ Tirer au sort une solution s' appartenant à $V(s)$ ♦ $n_{iter} \leftarrow n_{iter} + 1$ ♦ `Si` $\Delta(f) \leq 0$ `alors` $s \leftarrow s'$ ♦ `Sinon` ◊ Calcul d'une probabilité d'acceptation $a = e^{-\frac{\Delta f}{T}}$ ◊ Tirage au sort de p dans [0,1] ◊ `Si` $p \leq a$ `alors` $s \leftarrow s'$ ◊ `Sinon` s est conservée. ◊ `FindeSi` ♦ `FindeSi` ♦ $T \leftarrow kT$ avec $k < 1$ `/*`Baisse de la température `*/` `FindeTantQue`

TAB. 2.7.9 – *Algorithme du recuit simulé*

fournissent rapidement des solutions de bonne qualité. Si on dispose de plus de temps sur la construction d'une solution alors la méthode tabou fournit les meilleurs résultats et améliore la qualité des ordonnancements de 4%. Dans [58] une étude a été menée sur la comparaison d'heuristiques de liste et de méthodes de recuit simulé. Cet article montre que dans 70 % des cas, les heuristiques de liste fournissent d'aussi bons résultats que les méthodes de recuit simulé au prix d'un temps d'exécution 100 fois plus petit et ceci dans le cas où les opérations du graphe de l'algorithme ne sont pas contraintes sur des processeurs.

3. Heuristiques d'optimisation de la latence

3.1 Introduction

Le terme latence désigne ici la durée totale d'exécution d'une seule itération du graphe de l'algorithme sur le graphe de l'architecture. Le graphe de l'algorithme est un hypergraphe orienté conditionné infiniment itéré (cf. modèle d'algorithme page 29), et le graphe de l'architecture est un hypergraphe non orienté (cf. modèle d'architecture page 15). Associés à un couple d'algorithme et d'architecture donnés, on a construit l'ensemble des implantations valides de cet algorithme sur cette architecture en composant trois relations : le routage, la distribution et l'ordonnancement. Chaque implantation de cet ensemble, encore appelée graphe d'algorithme implanté, est dite valide car, par construction, l'ordre partiel d'exécution associé inclut l'ordre partiel du graphe de l'algorithme initial. Nous allons maintenant chercher parmi ces implantations, à l'aide d'une heuristique, une implantation dont la longueur du chemin critique soit inférieure ou égale à la contrainte temps réel du problème étudié.

L'ordre d'exécution associé à chacun des sous-graphes distribués et ordonnancés sur chaque ressource de type calcul ou média, d'un graphe implanté est total car comme nous l'avons vu dans le modèle d'architecture, chaque ressource est une machine purement séquentielle. L'ordre d'exécution d'un graphe implanté est un ordre partiel égal à l'union des ordres totaux associés à chacune des ressources composant l'architecture et de l'ordre partiel associé aux dépendances entre opérations de calcul et de transfert.

Les heuristiques proposées prennent en compte aussi bien les durées des opérations de calcul que les durées des transferts de données grâce aux opérations de transfert définies dans le modèle d'implantation. Contrairement à la modélisation qui effectue d'abord la distribution de tout le graphe de l'algorithme puis l'ordonnancement du graphe, chaque heuristique effectue successivement pour chaque opération de calcul, la distribution et l'ordonnancement de cette opération, et les éventuels distribution et ordonnancement des opérations de transfert associées, avant de passer à la distribution et l'ordonnancement de l'opération de calcul suivante. Ceci est réalisé en respectant l'ordre partiel induit par les précédences du graphe de l'algorithme d'application. Ainsi, ces heuristiques font partie des méthodes *multiple-stage*

définies dans le chapitre état de l'art à la page 103.

Les heuristiques choisies sont statiques : les choix de distribution et d'ordonnancement sont réalisés à la compilation et non à l'exécution. Ceci est possible, car la structure du graphe de l'algorithme est connue a priori et ce graphe est étiqueté par les durées d'exécution des opérations et par les transferts de données (cf. chapitre caractérisation page 81). Ensuite on peut générer à l'aide du logiciel d'aide à l'implantation SynDEx [53][1] supportant la méthodologie A^3 [75], un exécutif [34] principalement statique et donc à faible coût.

Nous allons présenter deux méthodes approchées de résolution du problème de distribution et d'ordonnancement : tout d'abord une heuristique gloutonne [54, 74] implantée sous forme d'algorithme de liste [88] dynamique, puis une méthode de voisinage de recherche locale, plus précisément avec retour-arrière. Pour chacune de ces méthodes, on étudie le cas où l'algorithme est un hypergraphe orienté non conditionné et le cas où l'hypergraphe est conditionné. Cette heuristique gloutonne construit rapidement une implantation de bonne qualité [58]. Cette implantation est ensuite améliorée par une méthode avec retour-arrière [83] en recherchant dans son voisinage une meilleure implantation. Afin d'évaluer la qualité de l'implantation obtenue avec notre heuristique gloutonne, nous allons la comparer sur quelques exemples d'application à d'autres heuristiques du même type.

3.2 Dates mises en jeu dans les heuristiques

3.2.1 Avant-propos sur un graphe partiellement implanté

Lors de la caractérisation, nous avons défini des dates associées à une implantation. Ici l'heuristique construit progressivement une implantation, c'est-à-dire que l'on passe d'un graphe non implanté, à un graphe partiellement implanté, pour arriver à un graphe implanté. Nous avons besoin de calculer les dates pour les opérations aussi bien pour le graphe initial que pour le graphe partiellement implanté. Le problème est que pour chaque opération non implantée, il y a autant de durées d'exécution que de ressources pouvant exécuter cette opération. Afin de limiter l'explosion combinatoire, on associe à chaque opération non implantée, une approximation de sa durée d'exécution. Nous avons choisi de prendre comme approximation, la moyenne empirique des durées d'exécution. Soit $\Delta_{app}(o_i)$, l'approximation de la durée d'exécution de l'opération o_i.

$$\Delta_{app}(o_i) = \frac{1}{n_i} \sum_{j=1}^{n_i} \Delta(o_i,p_j)$$

et n_i désigne le nombre de processeurs pouvant exécuter l'opération o_i.

1. http://www-rocq.inria.fr/syndex

Sur un graphe partiellement implanté, la fonction Π est partiellement définie. Soit $Dom(\Pi)$, le domaine de définition de Π ; il est égal à l'ensemble des opérations implantées. Alors la durée d'exécution $\Delta(o_i)$ d'une opération o_i sur un graphe partiellement implanté est définie par :

$$\Delta(o_i) = \begin{cases} \Delta(o_i,\Pi(o_i)) & si \quad o_i \in Dom(\Pi) \\ \Delta_{app}(o_i) & si \quad o_i \notin Dom(\Pi) \end{cases}$$

A la $n$ième étape de l'heuristique, on a une implantation partielle construite entre les étapes 1 et $n-1$. Et on cherche à compléter cette implantation à l'étape n. Pour ce faire, des tentatives d'implantation sont effectuées séparément pour chacune des opérations candidates sur chacune des ressources où les opérations candidates peuvent s'exécuter. Si a est le nombre de candidats à l'étape n et si n_i est le nombre de ressources où l'opération o_i peut s'exécuter, il y aura à l'étape n, $\sum_{i=1}^{a} n_i$ implantations partielles à cette étape. Seule une implantation sera retenue, en utilisant un certain critère, à cette étape et servira d'implantation partielle pour l'étape $n+1$. Les autres implantations construites seront abandonnées.

Soit $\lessdot$, l'ordre partiel associé à un graphe partiellement implanté.

Propriété 46

$$\preceq \subseteq \lessdot \subseteq \trianglelefteq$$

où $\preceq$ est l'ordre partiel associé au graphe de l'algorithme initial non implanté, et $\trianglelefteq$ est l'ordre partiel associé au graphe de l'algorithme implanté. Si $\lessdot = \preceq$, le graphe n'est pas implanté, et si $\lessdot = \trianglelefteq$, le graphe est implanté.

Pour tout graphe partiellement implanté on a :

$$E(o_j) \leq S(o_i) \quad \forall\ o_j \lessdot o_i \Rightarrow \begin{cases} S(o_i) = \max\limits_{o_j \lessdot o_i} E(o_j) \\ E(o_j) = S(o_j) + \Delta(o_j) \end{cases}$$

$$\bar{e}(o_i) \geq \bar{s}(o_j) \quad \forall\ o_i \lessdot o_j \Rightarrow \begin{cases} \bar{e}(o_i) = \max\limits_{o_i \lessdot o_j} \bar{s}(o_j) \\ \bar{s}(o_j) = \bar{e}(o_j) + \Delta(o_j) \end{cases}$$

Voyons maintenant de façon plus précise à quoi sont égales la relation $\lessdot$ et les dates associées pour les opérations candidates à l'étape n.

Définition 60 *Une opération est* candidate à l'implantation *ou encore* implantable, *lorsque tous ses prédécesseurs ont été implantés. Si cette opération est une opération de type retard, alors ses successeurs doivent aussi avoir été implantés, ceci afin de respecter l'ordre partiel de l'algorithme.*

3.2.2 Définitions

Dates au-plus-tôt depuis le début

Soit $(\Gamma^{-1})^*(o_i)$, l'ensemble des prédécesseurs de l'opération o_i comprenant à la fois les opérations de type calcul et les opérations de transfert.

Soit $S^{(n)}(o_i,u_j)$, la date de début au-plus-tôt (depuis le début) de l'opération o_i sur la ressource u_j à l'étape n de l'heuristique. Pour que l'opération o_i, candidate à l'étape n, puisse débuter sur la ressource u_j, il faut que les deux conditions suivantes soient réunies :

1. *Que les entrées de l'opération o_i soient disponibles à cette étape*, c'est-à-dire que les prédécesseurs de l'opération correspondant à l'ensemble $(\Gamma^{-1})^*(o_i)$, aient terminé leur exécution.

   Remarque 56 *Si le graphe de l'algorithme d'application est non conditionné, les entrées d'une opération sont de type données uniquement alors que, dans le cas contraire, les entrées peuvent être non seulement de type données mais également de type conditionnement. Une entrée de type conditionnement est un booléen de conditionnement qui conditionne l'exécution de l'opération dont il est l'entrée d'activation, comme on l'a vu dans le modèle d'algorithme.*

2. *Que la ressource u_j soit disponible (terme défini ci-après) à l'étape n pour l'opération o_i.* Soit $X_2^{(n)}(o_i,u_j)$, l'ensemble des opérations implantées jusqu'à l'étape n sur u_j et dont l'exécution doit être achevée avant que celle de o_i commence. La date de disponibilité d'une ressource u_j pour l'opération o_i à l'étape n de l'heuristique est égale à la plus grande des dates de fin des opérations de l'ensemble $X_2^{(n)}(o_i,u_j)$.

Remarque 57 *Les ensembles $(\Gamma^{-1})^*(o_i)$ et $X_2^{(n)}(o_i,u_j)$ peuvent être confondus ou avoir une intersection non vide. En effet, si l'architecture est une architecture monoprocesseur, les deux ensembles sont confondus.*

Soit $\lessdot^{(n)}(o_i,u_j)$, l'ensemble des prédécesseurs à l'étape n de la candidate o_i quand on tente de l'implanter sur la ressource u_j : $\lessdot^{(n)}(o_i,u_j) = \{o_j \in O / o_j \lessdot o_i\}$.

D'après ce qui précède, on a : $\lessdot^{(n)}(o_i,u_j) = (\Gamma^{-1})^*(o_i) \cup X_2^{(n)}(o_i,u_j)$.

Propriété 47 *Une fois que o_i est implantée, l'ensemble $\lessdot^{(n)}(o_i,u_j)$ ne varie plus, il en est de même pour $S^{(n)}(o_i,u_j)$ et $S^{(n)}(o_i,u_j) = S(o_i,\Pi(o_i))$.* En effet, dès que o_i est implantée définitivement, l'ensemble de ses prédécesseurs ne varie plus, ce qui n'est pas le cas avant à cause de l'éventuel ajout d'opérations de transfert.

Dates au-plus-tard depuis la fin

Soit $\bar{s}(o_i,u_j)$ la date de début au-plus-tard (depuis la fin) de l'opération o_i sur la ressource u_j. Soit $\succ^{(n)}(o_i)$, l'ensemble des successeurs à l'étape n de la candidate o_i quand on tente de l'implanter sur la ressource u_j : $\succ^{(n)}(o_i) = \{o_j \in O / o_i \prec o_j\}$.

Quand on cherche à implanter une opération candidate, ses successeurs n'ont pas encore été implantés par définition. Ainsi, l'ensemble des successeurs d'une opération candidate est indépendant de la ressource sur laquelle on tente de l'implanter ; cet ensemble est égal à l'ensemble de ses successeurs du graphe de l'algorithme initial. D'après ce qui précède, on a : $\succ^{(n)}(o_i) = \Gamma(o_i)$. On remarque donc que l'ensemble $\Gamma(o_i)$ est indépendant de n et il est de plus composé uniquement d'opérations de calcul.

Propriété 48 *$\bar{s}(o_i,p_j)$ et $\bar{e}(o_i,p_j)$ sont identiques sur toutes les ressources et sont inchangés tant qu'aucun des successeurs de o_i n'a été implanté.*

$$\bar{s}(o_i,p_j) = \bar{s}(o_i) \quad , \quad \bar{e}(o_i,p_j) = \bar{e}(o_i) \quad \forall p_j \in P$$

En effet, les dates calculées depuis la fin associées à une opération o_i dépendent uniquement des successeurs de cette opération. Donc tant qu'aucun successeur de o_i n'a été ordonnancé, les dates depuis la fin de l'opération o_i sont invariables, par contre une fois les successeurs de o_i placés, des opérations de transfert peuvent s'intercaler entre o_i et ses successeurs, donc le calage à droite est modifié et aussi la date de début définie depuis la fin.

Relation entre les deux types de dates

Sauf si le contraire est explicitement précisé, les dates sont exprimées depuis le début. Relation entre deux dates T au-plus-tard (resp. au-plus-tôt)(définie depuis le début) et $\bar{T}$ au-plus-tard (resp. au-plus-tôt)(définie depuis la fin) : $T(o_i) = \mathcal{R} - \bar{T}(o_i)\ \forall o_i \in O$. Pour être plus précis :

$$T^{(n)}(o_i) = \mathcal{R}_i^{(n)} - \bar{T}^{(n)}(o_i) \tag{3.2.1}$$

où n est le numéro de l'étape de l'heuristique. En fait, tant qu'aucun successeur de o_i n'est implanté, il ne peut pas y avoir d'opérations de transfert insérées entre les deux et donc o_i conserve sa date définie depuis la fin jusqu'à ce moment (cf. propriété 48). Donc :

$$T^{(n)}(o_i) = \mathcal{R}_i^{(n)} - \bar{T}(o_i) \text{ tant qu'aucun successeur de } o_i \text{ n'a été placé}$$

Et donc la différence $T^{(.)}(o_i) - \mathcal{R}_i^{(.)}(o_i)$ est constante tant qu'aucun successeur de o_i n'a été placé, et ce quelle que soit l'étape de l'ordonnancement considérée. On se rend compte ainsi que le calcul du chemin critique a la même complexité que celui de date définie depuis le début.

Nous allons maintenant voir à quoi est égal $X_2^{(n)}(o_i,u_j)$ dans les deux versions de l'heuristique.

3.2.3 Calculs de dates sans conditionnement

L'ensemble des opérations distribuées et ordonnancées sur u_j dont l'exécution doit être achevée avant que celle de o_i ne commence est égal à l'ensemble de toutes les opérations implantées sur u_j jusqu'à l'étape n incluse.

$$X_2^{(n)}(o_i,u_j) = \Pi^{-1}(u_j)$$

On constate que $X_2^{(n)}(o_i,u_j)$ est indépendant de l'opération o_i. On en déduit la propriété suivante :

Propriété 49

$$X_2^{(n)}(o_i,u_j) = X_2^{(n)}(u_j) \quad \forall o_i \in O$$

Et donc cela revient à dire que lorsqu'on ne prend pas en compte le conditionnement, on a une et une seule date de disponibilité pour une ressource donnée à une étape donnée.

3.2.4 Calculs de dates avec conditionnement

L'ensemble des opérations distribuées et ordonnancées sur u_j dont l'exécution doit être achevée avant que celle de o_i ne commence est égal à l'ensemble des opérations implantées sur u_j et qui n'appartiennent pas à l'ensemble des opérations exclusives de o_i.

$$X_2^{(n)}(o_i,u_j) = \Pi^{-1}(u_j) \setminus \hat{\hat{\Gamma}}_c[\phi^{-1} \circ \bar{\phi}(o_i)]$$

Remarque 58 *L'ensemble $\hat{\hat{\Gamma}}_c[\phi^{-1} \circ \bar{\phi}(o_i)]$, défini dans le modèle d'algorithme dans la première partie de la thèse à la page 45, est l'ensemble des descendants conditionnés par $\bar{\phi}(o_i)$, c'est-à-dire le booléen complémentaire de $\phi(o_i)$ qui lui, conditionne l'opération o_i. Autrement dit l'ensemble $\hat{\hat{\Gamma}}_c[\phi^{-1} \circ \bar{\phi}(o_i)]$ est l'ensemble des opérations exclusives de l'opération o_i.*

3.2.5 Conclusions

On remarque que les entrées d'une opération sont toujours égales aux prédécesseurs (de type calcul ou de type transfert) quel que soit le modèle d'algorithme implanté. Et donc la seule chose qui change quand on passe du modèle non conditionné au modèle conditionné est la notion de disponibilité d'une ressource.

Si l'on considère un graphe d'algorithme implanté non conditionné, la disponibilité d'une ressource est égale à la date de fin de la dernière opération ordonnancée sur cette ressource.

Si l'on considère un graphe d'algorithme implanté conditionné, la date de disponibilité d'une ressource est variable suivant les opérations. Autrement dit, à une ressource donnée correspond plusieurs dates de disponibilité possibles.

Ceci vient du fait que l'ordre partiel sur une même ressource entre opérations exclusives est possible lors de l'ordonnancement, car on sait qu'à l'exécution par définition les opérations exclusives ne peuvent être simultanément exécutées au cours de l'exécution du même instant logique.

3.3 Algorithme de l'heuristique gloutonne

Définition 61 *Soit f une fonction de coût. On dit que deux opérations o'_i et o'_j ne peuvent être départagées par la fonction de coût f si et seulement si :*

$$|f(o'_i) - f(o'_j)| \leq \epsilon$$

ϵ étant donné par l'utilisateur et petit devant $f(.)$.

A chaque étape de l'heuristique, une liste d'opérations implantables auxquelles on a assigné des priorités est établie. Cette liste est ordonnée par priorité décroissante. La règle de priorité choisie repose sur une fonction de coût, la pression d'ordonnancement [54], qui vise à minimiser la longueur du chemin critique et à exploiter la marge d'ordonnancement de chaque opération. Pour ce faire, on calcule pour chaque opération autant de pressions d'ordonnancement qu'il y a de processeurs sur lesquels elle peut s'exécuter. On conserve pour chaque opération uniquement la pression d'ordonnancement la plus faible qui constitue la priorité de l'opération et on conserve également le processeur qui induit cette pression. Une fois que l'on a associé à chaque opération sa pression d'ordonnancement la plus faible et le processeur correspondant, on ordonne la liste par pression d'ordonnancement décroissante. Ensuite, l'opération la plus prioritaire est ordonnancée sur le processeur associé à la date à laquelle il est disponible pour cette opération. Si des opérations ont la même priorité, autrement dit elles ne peuvent être départagées par la fonction de coût, alors une de ces opérations est ordonnancée et les autres opérations sont marquées et serviront de voisinage pour la méthode approchée de recherche locale avec retour-arrière (cf. section 3.5). L'opération ordonnancée est retirée de la liste et ses successeurs candidats sont ajoutés à cette liste. Le processus est répété jusqu'à ce que la liste soit vide. On a ainsi construit une implantation solution du problème de distribution ordonnancement.

Plus précisément, l'algorithme d'ordonnancement est décrit à la table 3.3.1.

Initialisation de la liste des candidats avec les opérations sans prédécesseur (cf. section 3.3.1 équation 3.3.1)

`TantQue` la liste n'est pas vide

- Calcul de la pression d'ordonnancement de chaque candidat (cf. section 3.3.2 équation 3.3.2)
- Restriction de l'ensemble des candidats (cf. section 3.3.3 équation 3.3.3)
- Sélection du meilleur candidat et implantation de ce candidat sur le meilleur processeur qui lui est associé en implantant également les différentes opérations de transfert associées à ce meilleur candidat (cf. section 3.3.4 équations 3.3.4 et 3.3.5) et marquage des éventuelles opérations non départagées
- Ajout à la liste, des successeurs ordonnançables de ce candidat (cf. section 3.3.5 équation 3.3.6)
- Suppression de la liste, du candidat implanté (cf. section 3.3.6 équations 3.3.7)

`FindeTantQue`

TAB. 3.3.1 – *Algorithme d'ordonnancement de liste dynamique*

Notations

L'exposant entre parenthèses désigne l'étape de l'heuristique.

Soit $O_{cand}^{(n)}$, l'ensemble des opérations implantables à l'étape n de l'heuristique. Étant donné que l'heuristique est gloutonne, et que l'on implante une opération de calcul, le nombre d'étapes est égal au cardinal de l'ensemble des opérations calcul de G_{al}.

Soit $O_{ordo}^{(n)}$, l'ensemble des opérations implantées avant l'étape n.

$$O_{ordo}^{(n)} = \{o_i \in O \quad / \quad \exists! \quad u_j \in U \quad avec \quad \Pi(o_i) = u_j\}$$

Remarque 59 *Les seules opérations implantables au sens où on l'a défini sont des opérations de calcul, car quand on ajoute des opérations de transfert au graphe de l'algorithme, elles sont aussitôt implantées. Et donc il n'existe pas sur le graphe de l'algorithme des opérations de transfert qui ne soient pas implantées.*

3.3.1 Initialisation de la liste des candidats

L'ensemble des opérations implantables à l'étape 1 de l'heuristique est l'ensemble des opérations de calcul sans prédécesseur.

$$O_{cand}^{(1)} = \{o'_i \in O' \quad / \quad \Gamma^{-1}(o'_i) = \emptyset\} \tag{3.3.1}$$

3.3.2 Calcul de la pression d'ordonnancement

Pour chaque candidat o'_i appartenant à l'ensemble $O_{cand}^{(n)}$, on calcule sa pression d'ordonnancement $\sigma(o'_i,p_j)$ sur chacun des processeurs p_j de l'ensemble des processeurs capables d'exécuter l'opération o'_i. Cet ensemble peut être restreint au seul processeur sur lequel l'utilisateur a contraint l'opération o'_i à s'exécuter.

La pression d'ordonnancement mesure à la fois la marge d'ordonnancement ou l'allongement du chemin critique, et induit une priorité d'ordonnancement pour chaque opération.

Soit $P^{(n)}(o'_i,p_j)$ la pénalité d'ordonnancement engendrée par l'opération o'_i lorsqu'on tente de l'implanter sur le processeur p_j.

$$P^{(n)}(o'_i,p_j) = \mathcal{R}_{ij}^{(n)} - \mathcal{R}^{(n-1)}$$

où $\mathcal{R}_{ij}^{(n)}$ est la longueur d'ordonnancement à l'étape n quand on tente d'implanter l'opération o'_i sur le processeur p_j. La pénalité d'ordonnancement permet de mesurer l'allongement du chemin critique.

Soit $F^{(n)}(o'_i,p_j)$ la flexibilité ou marge d'ordonnancement engendrée par l'opération o'_i lorsqu'on tente de l'implanter sur le processeur p_j.

$$F^{(n)}(o'_i,p_j) = s^{(n)}(o'_i,p_j) - S^{(n)}(o'_i,p_j)$$

où $s^{(n)}(o'_i,p_j)$ (resp. $S^{(n)}(o'_i,p_j)$) est la date de début au-plus-tard (resp. au-plus-tôt) (depuis le début) de o'_i sur p_j à la $n$ième étape de l'heuristique.

La pression d'ordonnancement est donnée par :

$$\boxed{\sigma^{(n)}(o'_i,p_j) = P^{(n)}(o'_i,p_j) - F^{(n)}(o'_i,p_j)}$$

Étude des variations de σ

- Tant que la marge d'ordonnancement d'une opération est positive, le chemin critique ne s'allonge pas, la pénalité d'ordonnancement est donc nulle.
  $F^{(n)}(o'_i,p_j) \geq 0 \Rightarrow P^{(n)}(o'_i,p_j) \equiv 0 \Rightarrow \sigma^{(n)}(o'_i,p_j) = -F^{(n)}(o'_i,p_j) \leq 0$
  Donc si la pression d'ordonnancement est négative, la date de début au-plus-tôt de l'opération est inférieure à la date début au-plus-tard. Cela veut donc dire que l'opération a une certaine flexibilité d'ordonnancement égale à la valeur absolue de la pression d'ordonnancement.

- A partir du moment où la marge d'ordonnancement est nulle, l'opération est critique (sa date au-plus-tôt est égale à sa date au-plus-tard) et la longueur du chemin critique augmente.
  $F^{(n)}(o'_i,p_j) \equiv 0 \Rightarrow P^{(n)}(o'_i,p_j) \geq 0 \Rightarrow \sigma^{(n)}(o'_i,p_j) = P^{(n)}(o'_i,p_j) \geq 0$
  Si la pression d'ordonnancement est positive, l'opération est critique et engendre une pénalité d'ordonnancement égale à la pression d'ordonnancement.

Ainsi, plus la pression d'ordonnancement augmente, plus il est urgent d'implanter cette opération.

Simplification de la formule de σ

Pour calculer la pression d'ordonnancement de chaque candidat, des calculs de dates sont nécessaires. Les dates sont calculées pour chaque tentative d'implantation de chaque candidat sur chaque processeur et donc ces dates prennent en compte les éventuelles opérations de transfert créées à cette occasion et ordonnancées au-plus-tôt sur les média du graphe de l'architecture.

$\bar{s}(o'_i,p_j)$ est la date de début au-plus-tard (depuis la fin) de o'_i sur p_j. $\mathcal{R}^{(n-1)}$ est la longueur du chemin critique à l'étape $n-1$.

$\sigma^{(n)}(o'_i,p_j) = P^{(n)}(o'_i,p_j) - F^{(n)}(o'_i,p_j)$

en remplaçant $P^{(n)}(o'_i,p_j)$ et $F^{(n)}(o'_i,p_j)$ par leur valeur, on a :

$\sigma^{(n)}(o'_i,p_j) = \mathcal{R}_i^{(n)} - \mathcal{R}^{(n-1)} - (s^{(n)}(o'_i,p_j) - S^{(n)}(o'_i,p_j))$

D'après l'équation 3.2.1 définie page 123 : $s^{(n)}(o'_i,p_j) = \mathcal{R}_i^{(n)} - \bar{s}^{(n)}(o'_i,p_j)$. En remplaçant, on obtient,

$$\sigma^{(n)}(o'_i,p_j) = S^{(n)}(o'_i,p_j) + \bar{s}^{(n)}(o'_i,p_j) - \mathcal{R}^{(n-1)}$$

En utilisant la propriété 48 page 123, on a :

$$\boxed{\sigma^{(n)}(o'_i,p_j) = S^{(n)}(o'_i,p_j) + \bar{s}(o'_i) - \mathcal{R}^{(n-1)}}$$

Ainsi, $\sigma^{(n)}(o'_i,p_j) = S^{(n)}(o'_i,p_j) +$ constante , et donc la pression d'ordonnancement est une fonction affine de S.

Comparaison de complexité entre mobilité (flexibilité) et pression d'ordonnancement

Dans la pratique, la longueur du chemin critique $\mathcal{R}$ n'est jamais calculée lors du calcul de la pression d'ordonnancement. En effet, quand on compare deux pressions d'ordonnancement

de deux opérations différentes, les chemins critiques s'en vont.

$$\begin{array}{rrcl} & \sigma^{(n)}(o_i) & = & S^{(n)}(o_i) + \bar{s}(o_i) - \mathcal{R}^{(n-1)} \\ - & \sigma^{(n)}(o_j) & = & S^{(n)}(o_j) + \bar{s}(o_j) - \mathcal{R}^{(n-1)} \\ \hline & \delta\sigma(o_i,o_j) & = & (S^{(n)}(o_i) - S^{(n)}(o_j)) + (\bar{s}(o_i) - \bar{s}(o_j)) - (\mathcal{R}^{(n-1)} - \mathcal{R}^{(n-1)}) \end{array}$$

En revanche, dans le cas de la mobilité définie dans le chapitre précédent à la page 107, les chemins critiques ne s'annulent pas. La variation de mobilité fait intervenir deux longueurs de chemin critiques différentes, donc elles ne s'annulent pas, ce qui nécessite leurs calculs et donc augmente considérablement la complexité de l'algorithme.

$$\begin{array}{rrcl} & M^{(n)}(o_i) & = & \mathcal{R}_i^{(n)} - S^{(n)}(o_i) - \bar{s}(o_i) \\ - & M^{(n)}(o_j) & = & \mathcal{R}_j^{(n)} - S^{(n)}(o_j) - \bar{s}(o_j) \\ \hline & \delta M(o_i,o_j) & = & (\mathcal{R}_i^{(n)} - \mathcal{R}_j^{(n)}) - (S^{(n)}(o_i) - S^{(n)}(o_j)) - (\bar{s}(o_i) - \bar{s}(o_j)) \end{array}$$

La mobilité à l'étape n fait intervenir la longueur du chemin critique de l'étape n, c'est-à-dire l'étape courante, contrairement à la pression d'ordonnancement qui fait intervenir la longueur du chemin critique de l'étape précédente. Ainsi, la longueur du chemin critique de l'étape n calculée à l'étape n dépend de l'opération que l'on essaie d'implanter et donc cette longueur est variable d'un essai d'implantation à un autre.

Utilisation de la pression pour l'heuristique

Pour chaque candidat, on calcule la meilleure pression d'ordonnancement qui est donc celle qui induit la plus grande flexibilité d'ordonnancement et la plus faible pénalité d'ordonnancement, autrement dit la plus faible des pressions parmi celles calculées.

Pour chaque o'_i, on recherche le meilleur processeur p_i^{best} : $\forall o'_i \in O_{cand}^{(n)} \quad \sigma_{opt}^{(n)}(o'_i,p_i^{best}) = \min_{p_k \in P} \sigma(o_i,p_k)$. On obtient alors un couple $= (o_i,p_i^{best})$.

Ce qui s'écrit pour l'étape n de l'heuristique :

$$\boxed{\forall o'_j \in O_{cand}^{(n)} \quad \sigma_{opt}^{(n)}(o'_j) = \min_{p_k \in P} \sigma^{(n)}(o'_j,p_k)} \tag{3.3.2}$$

A cette pression $\sigma_{opt}(o'_i)$ correspond une date de début au-plus-tôt que l'on notera $S^{(n)}(o'_i)$, un processeur que l'on notera $\Pi(o'_i)$, et des opérations de transfert implantées sur des médias.

$$S^{(n)}(o'_i) = S^{(n)}(o'_i,\Pi(o'_i)) \quad tq \quad \sigma^{(n)}(o'_i,\Pi(o'_i)) = \sigma_{opt}^{(n)}(o'_i)$$

Remarque 60 *Soit c, le nombre d'opérations candidates à l'implantation sur p processeurs à l'étape n de l'heuristique. Alors, il y a au plus $(c*p)$ tentatives d'implantations d'opérations de calcul à cette étape.*

3.3.3 Restriction de l'ensemble des candidats

Parmi l'ensemble des candidats à l'étape n, seuls certains candidats sont déclarés implantables : on désigne par $O_{candI}^{(n)}$, le sous-ensemble des *cand*idats *i*mplantables à l'étape n. Voici comment se construit ce sous-ensemble à l'étape n de l'heuristique.

Soit $S_{min}^{(n)}$, la plus petite date de début des opérations de l'ensemble $O_{cand}^{(n)}$ à l'étape n.

$$S_{min}^{(n)} = \min_{o'_i \in O_{cand}^{(n)}} S^{(n)}(o'_i)$$

On note l'opération o'_{min}, l'opération dont la date de début est $S_{min}^{(n)}$.

Soit $E_{min}^{(n)}$, la date de fin au-plus-tôt de o'_{min}.

$$E_{min}^{(n)} = S_{min}^{(n)} + \Delta(o'_{min})$$

avec $\Delta(o'_{min}) = \frac{1}{n_P} \sum_{j=1}^{n_P} \Delta(o'_{min}, p_j)$ avec $p_j \in P$ et $\mathrm{Card}P = n_P$

Les opérations retenues dans le sous-ensemble $O_{candI}^{(n)}$ sont les opérations dont la date de début au-plus-tôt est strictement inférieure à la date de fin au-plus-tôt E_{min} précédemment définie.

$$O_{candI}^{(n)} = \{o'_i \in O_{cand}^{(n)} \quad / \quad S^{(n)}(o'_i) < E_{min}^{(n)}\} \tag{3.3.3}$$

L'intérêt de restreindre l'ensemble des candidats à une étape donnée est d'éviter d'avoir un ordonnancement avec des attentes sur les processeurs dans lesquelles on pourrait insérer des opérations, ce qui est illustré par l'exemple suivant.

On suppose pour simplifier l'exemple que les durées des communications inter-processeur sont nulles. Soient I,J,A,B quatre opérations de calcul à implanter sur deux processeurs p_0 et p_1 connectés par un média m (cf. figure 3.3.1). I et J sont implantées sur p_0 et p_1 respectivement et A et B ne peuvent être implantées que sur p_0. On suppose que les durées des opérations I,A,B sont unitaires et que J a une durée de 2,1. Calculons la pression associée à chacune des opérations A et B. $\sigma(A) = S(A) + \bar{s}(A) - \mathcal{R}$, avec $S(A) = E(I) = 1$, $\bar{s}(A) = 1$, $\mathcal{R} = 3{,}1$ soit $\sigma(A) = -1{,}1$.

$\sigma(B) = S(B) + \bar{s}(B) - \mathcal{R}$ avec $S(B) = \max(E(I),E(J)) = E(J) = 2{,}1$ et $\bar{s}(B) = 1$ soit $\sigma(B) = O$

L'opération la plus urgente à ordonnancer est celle qui a la pression la plus élevée (cf. équation 3.3.4) soit ici B. L'opération B est ordonnancée sur p_0 à la date 2,1 et ensuite l'opération A est ordonnancée à la date 3,1. On voit que le processeur p_0 est libre entre les dates 1 et 2,1 ce qui est suffisant pour accueillir l'opération A qui a une durée unitaire, on obtient dans ce cas un ordonnancement à "trous." En revanche, en restreignant les candidats

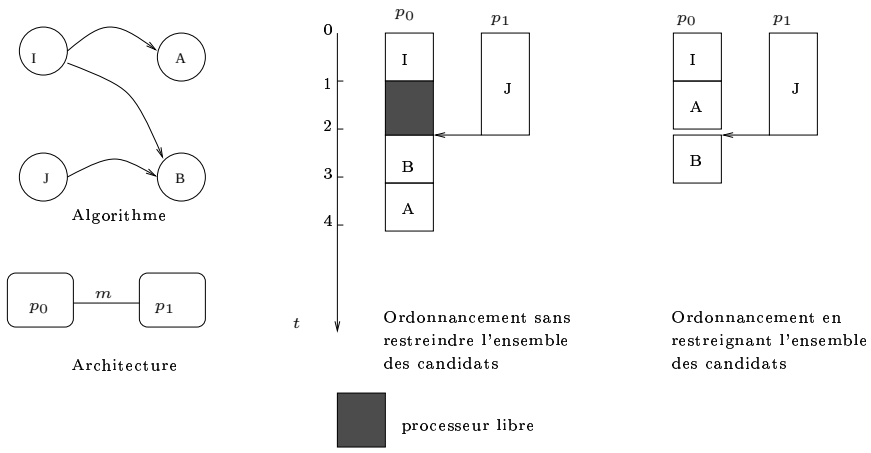


FIG. 3.3.1 – *Intérêt de restreindre l'ensemble des candidats*

tel qu'indiqué plus haut, seul A est candidat car la date E_{min} est égale à $S(A)+\Delta(A)$ donc égal à 2 et $S(B)$ qui est égale à 2,1 est supérieure à E_{min} donc l'opération B est retirée des candidats implantables à cette étape. Ainsi, A est ordonnancée à la date 1 sur p_0 et B à la date 2,1 (ce qui est la même date que dans l'ordonnancement précédent). Cette fois-ci l'ordonnancement est dit "sans trous". Les deux ordonnancements sont donnés figure 3.3.1.

3.3.4 Sélection du meilleur candidat et implantation

Sélection

Parmi les meilleures paires opération/processeur, on choisit celle qui a la pression la plus élevée, autrement dit celle qui risque d'allonger le plus le chemin critique. Ainsi, on implantera à l'étape n l'opération $o'^{(n)}_{elue}$ vérifiant :

$$\boxed{\sigma^{(n)}_{best}(o'^{(n)}_{elue}) = \max_{o'_j \in O^{(n)}_{candI}} \sigma^{(n)}_{opt}(o'_j)} \tag{3.3.4}$$

S'il existe plusieurs couples (o_i,p_i^{best}) tels que $\sigma^{(n)}_{opt}(o_i,p_i^{best}) = \sigma^{(n)}_{best}(o_.,p_.^{best})$, alors un couple est choisi au hasard parmi les couples équivalents pour être ordonnancé à l'étape n. Les autres couples sont mémorisés et serviront de voisinage pour la méthode avec retour-arrière.

Implantation

L'opération $o'^{(n)}_{elue}$ est implantée à l'étape n sur le processeur $\Pi(o'^{(n)}_{elue})$. Alors

$$O^{(n+1)}_{ordo} = O^{(n)}_{ordo} \cup o'^{(n)}_{elue} \cup \{\text{opérations de transfert associées implantées sur les média}\} \tag{3.3.5}$$

3.3.5 Ajout des successeurs ordonnançables

On ajoute à la liste des candidats les successeurs implantables du candidat implanté. C'est donc l'ensemble des successeurs de $o'^{(n)}_{elue}$ et dont tous les prédécesseurs ont déjà été implantés. Soit $\Gamma^{(n+1)}_{ordo}(o'^{(n)}_{elue})$, l'ensemble des successeurs implantables de $o'^{(n)}_{elue}$.

$$\Gamma^{(n+1)}_{ordo}(o'^{(n)}_{elue}) = \Gamma(o'^{(n)}_{elue}) \ \setminus \ \{o'_j \in \Gamma(o'^{(n)}_{elue}) \quad / \quad \Gamma^{-1}(o'_j) \not\subset O^{(n+1)}_{ordo}\}$$

$$O^{(n+1)}_{cand} = O^{(n)}_{cand} \cup \Gamma^{(n+1)}_{ordo}(o'^{(n)}_{elue}) \tag{3.3.6}$$

Comme l'heuristique est gloutonne, elle réalise à chaque étape une implantation partielle et définitive. Ceci se traduit par la série d'inclusions suivantes :

$$O^{(1)}_{ordo} \subset O^{(2)}_{ordo} \subset O^{(3)}_{ordo} \subset ..... \subset O^{(n)}_{ordo} .... \subset O$$

De plus, l'intersection entre les ensembles $O^{(n)}_{ordo}$ et $O^{(n+1)}_{ordo}$ est égale à l'opération de calcul implantée à l'étape $n+1$ plus l'ensemble des opérations de transfert ajoutées lors de l'implantation de cette opération.

3.3.6 Suppression du candidat implanté

Dans la liste constituée des futurs candidats à l'étape suivante, c'est-à-dire à l'étape $n+1$, on retire l'opération implantée à l'étape n.

$$O^{(n+1)}_{cand} = O^{(n+1)}_{cand} \setminus \{o'^{(n)}_{elue}\} \tag{3.3.7}$$

Des équations 3.3.6 et 3.3.7, on déduit la propriété suivante :

Propriété 50 *A une étape n donnée, une opération ne peut être à la fois implantable et implantée : $O^{(n)}_{cand} \cap O^{(n)}_{ordo} = \emptyset$.*

Le procédé est itéré jusqu'à ce que toutes les opérations de G_{al} aient été distribuées et ordonnancées.

3.4 Exemple

Nous allons détailler sur un exemple le déroulement de l'heuristique. On considère le graphe de l'algorithme défini dans la figure 3.4.2. Il est composé de quatre opérations o_1,o_2,o_3,o_4. On cherche à implanter cet algorithme sur une architecture décrite figure 3.4.2 composée de deux processeurs p_1 et p_2 connectés par un média m.

Pour simplifier cet exemple, on suppose d'une part que les durées des communications inter-processeur sont identiques et égales à 6 et d'autre part que les processeurs ont les mêmes

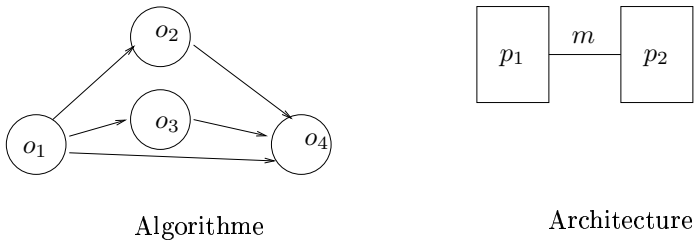


FIG. 3.4.2 – *Exemple*

	Δ	$S^{(1)}(.)$	$E^{(1)}(.)$	$\bar{e}(.)$	$\bar{s}(.)$
o_1	5	0	5	20	25
o_2	15	5	20	5	20
o_3	5	5	10	5	10
o_4	5	20	25	0	5

TAB. 3.4.2 – *Tableau des durées et des différentes estimations par défaut des dates des opérations de la figure 3.4.2*

caractéristiques. Par conséquent, chaque opération a la même durée sur tous les processeurs. Le tableau 3.4.2 donne les valeurs des durées de chaque opération et des estimations par défaut avant implantation des dates associées aux opérations. Ces dates ne prennent pas en compte les durées associées aux transferts de données.

Voici les différentes étapes du déroulement de l'algorithme d'ordonnancement :

Première étape

Une seule opération est candidate : $O^{(1)}_{cand} = \{o_1\}$. On choisit arbitrairement d'implanter o_1 sur p_1. On doit également y être contraint à cause des contraintes de placement des entrées/sorties matérielles. La longueur $\mathcal{R}^{(1)}$ du chemin critique est égale à 25.

Deuxième étape

Deux opérations sont candidates : $O^{(2)}_{cand} = \{o_2,o_3\}$.

$\sigma^{(2)}(o_i,p_j) = S^{(2)}(o_i,p_j) + \bar{s}(o_i,p_j) - \mathcal{R}^{(1)}$ pour $i = 1,2$ et $j = 1,2$.

- opération o_2

  $S^{(2)}(o_2,p_1) = E^{(1)}(o_1)$

  $S^{(2)}(o_2,p_2) = E^{(2)}(o''_{12})$ où o''_{12} est l'opération de transfert implantée sur le média m, insérée entre o_1 et o_2, pour modéliser le transfert de données entre o_1 et o_2.

  $S^{(2)}(o''_{12},m) = E^{(1)}(o_1)$

- opération o_3

  $S^{(2)}(o_3,p_1) = E^{(1)}(o_1)$

	Δ	$S^{(2)}(.,p_1)$	$S^{(2)}(.,p_2)$	$S^{(2)}(.,m)$
o_2	15	5	11	$\times$
o_3	5	5	11	$\times$
o''_{12}	6	$\times$	$\times$	5
o''_{13}	6	$\times$	$\times$	5

TAB. 3.4.3 – *Dates réelles de début au-plus-tôt des opérations o_2 et o_3 et des opérations de transfert associées*

$S^{(2)}(o_3,p_2) = E^{(2)}(o''_{13})$ où o''_{13} est l'opération de transfert implantée sur le média m, insérée entre o_1 et o_3, pour modéliser le transfert de données entre o_1 et o_3.
$S^{(2)}(o''_{13},m) = E^{(1)}(o_1)$

Les dates réelles de début au-plus-tôt des opérations de calcul o_2 et o_3 et des opérations de transfert associées o''_{12} et o''_{13} sont données dans le tableau 3.4. Toutes les cases du tableau ne sont pas remplies car les opérations de calcul ne peuvent s'exécuter que sur les processeurs et les opérations de transfert que sur les média. La droite de pression d'ordonnancement (cf figure 3.4.3) représente tous les cas d'implantation. Suivant l'implantation que l'on effectue, on a une pression correspondante que l'on notera $\sigma^{(2)}(o_i,p_j)$.

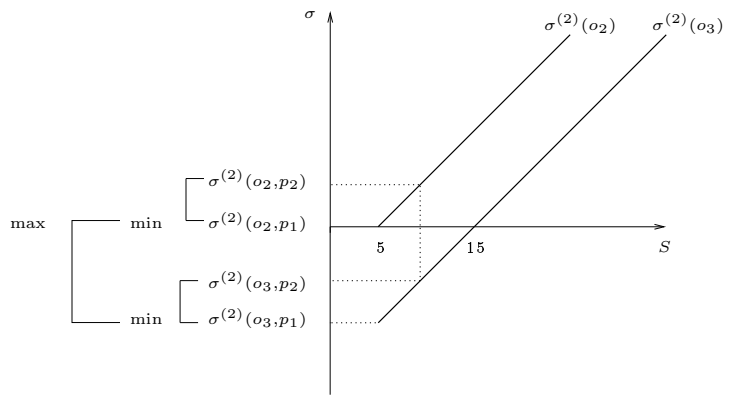


FIG. 3.4.3 – *Pression d'ordonnancement des opérations o_2 et o_3 à l'étape 2 de l'algorithme d'ordonnancement*

$$\Longrightarrow \left\{ \begin{array}{lllll} \sigma^{(2)}_{opt}(o_2) & = & \min(\sigma^{(2)}(o_2,p_1),\sigma^{(2)}(o_2,p_2)) & = & \sigma^{(2)}(o_2,p_1) \\ \sigma^{(2)}_{opt}(o_3) & = & \min(\sigma^{(2)}(o_3,p_1),\sigma^{(2)}(o_3,p_2)) & = & \sigma^{(2)}(o_3,p_1) \end{array} \right.$$

Remarque 61 *On constate dans le tableau 3.4 que les deux opérations de transfert ont la même date de début sur le média m. Les deux opérations de transfert ne seront pas présentes*

	Δ	$S^{(3)}(.,p_1)$	$S^{(3)}(.,p_2)$	$S^{(3)}(.,m)$
o_3	5	20	11	$\times$
o''_{13}	6	$\times$	$\times$	5

TAB. 3.4.4 – *Dates réelles de début au-plus-tôt de l'opération o_3 et de l'opération de transfert associée*

simultanément car à chaque étape une et une seule opération de calcul est ordonnancée, ici soit o_2 soit o_3 et donc comme une opération de transfert est associée à o_2 et l'autre à o_3 elles ne seront pas toutes les deux présentes dans le graphe final implanté.

Candidat élu : $\sigma^{(2)}(o_{elue}) = \max\limits_{O^{(2)}_{cand}}(\sigma(o_2,p_1),\sigma(o_3,p_1)) = \sigma(o_2,p_1)$

On implante o_2 sur p_1, il n'y a pas d'opérations de transfert ajoutées, le graphe de l'algorithme d'application est donc inchangé et la longueur $\mathcal{R}^{(2)}$ du chemin critique est égale à 25.

Troisième étape

Une seule opération est candidate à cette étape : $O^{(3)}_{cand} = \{o_3\}$.

$\sigma^{(3)}(o_3,p_j) = S^{(3)}(o_3,p_j) + \bar{s}(o_3) - \mathcal{R}^{(2)}$ pour $j = 1,2$.

$S^{(3)}(o_3,p_1) = E^{(2)}(o_2)$

$S^{(3)}(o_3,p_2) = E^{(3)}(o''_{13})$ où o''_{13} est l'opération de transfert implantée sur le média m pour modéliser le transfert de données entre o_1 et o_3.

$S^{(3)}(o''_{13},m) = E^{(1)}(o_1)$

Les valeurs de ces dates sont données dans le tableau 3.4.4. La pression d'ordonnancement associée à o_3 est donnée figure 3.4.4.

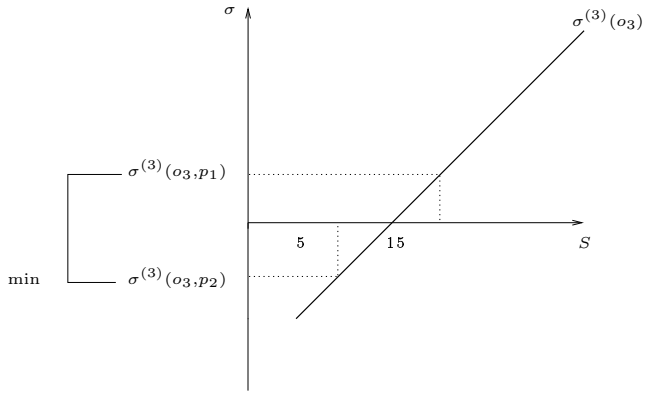


FIG. 3.4.4 – *Pression d'ordonnancement de l'opération o_3 à l'étape 3 de l'algorithme d'ordonnancement*

	Δ	$S^{(4)}(.,p_1)$	$S^{(4)}(.,p_2)$	$S^{(4)}(.,m)$
o_4	5	22	26	$\times$
o''_{14}	6	$\times$	$\times$	11
o''_{24}	6	$\times$	$\times$	20
o''_{34}	6	$\times$	$\times$	16

TAB. 3.4.5 – *Dates réelles de début au-plus-tôt de l'opération o_4 et des opération de transfert associées*

$\Longrightarrow \sigma^{(3)}_{opt}(o_3) = \min(\sigma^{(3)}(o_3,p_1),\sigma^{(3)}(o_3,p_2)) = \sigma^{(3)}(o_3,p_2)$

Candidat élu : $\sigma^{(3)}(o_{elue}) = \sigma^{(3)}_{opt}(o_3)$ puisqu'il n'y a qu'un candidat, on implante o_3 sur p_2 et on implante l'opération de transfert o''_{13} sur le média m à la date 5. Le graphe de l'algorithme est donc modifié et la longueur $\mathcal{R}^{(3)}$ du chemin critique est égale à 25.

Quatrième étape

Il n'y a plus qu'une seule opération à implanter : $O^{(4)}_{cand} = \{o_4\}$.

$\sigma^{(4)}(o_4,p_j) = S^{(4)}(o_4,p_j) + \bar{s}(o_4) - \mathcal{R}^{(3)}$ pour $j = 1,2$.

$S^{(4)}(o_4,p_1) = \max(E^{(1)}(o_1),E^{(2)}(o_2),E^{(4)}(o''_{34})) = E^{(2)}(o_2)$

$S^{(4)}(o_4,p_2) = \max(E^{(3)}(o_3),E^{(4)}(o''_{14}),E^{(4)}(o''_{24})) = E^{(4)}(o''_{24}))$

$S^{(4)}(o''_{14},m) = E^{(1)}(o_1)$

$S^{(4)}(o''_{24},m) = E^{(2)}(o_2)$

$S^{(4)}(o''_{34},m) = E^{(3)}(o_3)$

Les valeurs de ces dates sont données dans le tableau 3.4.5. La pression d'ordonnancement de l'opération o_4 est donnée figure 3.4.5.

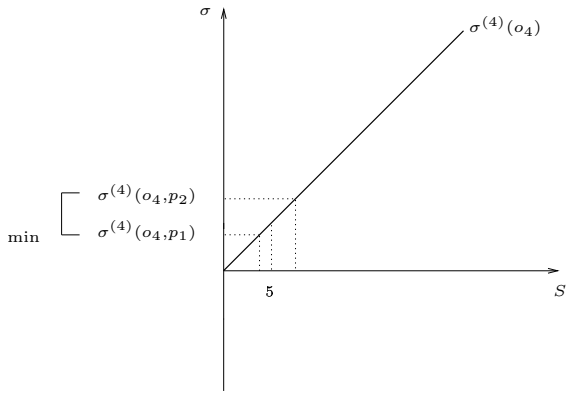
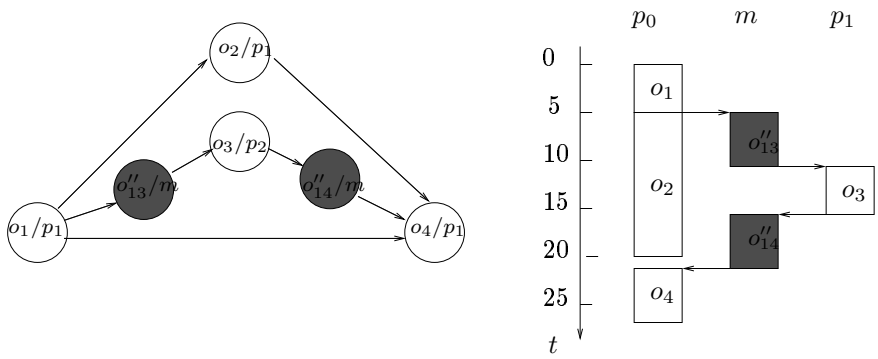


FIG. 3.4.5 – *Pression d'ordonnancement*

$\Longrightarrow \sigma^{(4)}_{opt}(o_4) = \min(\sigma^{(4)}(o_4,p_1),\sigma^{(4)}(o_4,p_2)) = \sigma^{(4)}(o_4,p_1)$

Candidat élu : $\sigma^{(4)}(o_{elue}) = \sigma^{(4)}_{opt}(o_4)$ puisqu'il n'y a qu'un candidat et on implante o_4 sur p_1 et l' opération de transfert o''_{34} sur le média m à la date 16 ; la longueur $\mathcal{R}^{(4)}$ du chemin critique est égale à 27.

La figure 3.4.6 donne le graphe de l'algorithme implanté obtenu par l'heuristique. Ce graphe contient deux opérations de transfert o''_{13} et o''_{34} ajoutées au graphe de l'algorithme initial et implantées sur le média m. Ce graphe correspond au graphe implanté décrit dans la première partie de la thèse. Sur chacune des ressources p_o,m,p_1 composant l'architecture, l'ordre d'exécution est total et contient l'ordre partiel initial. Le graphe de l'algorithme implanté étiqueté correspond à l'étiquetage du graphe de l'algorithme implanté défini au chapitre Caractérisation.



Graphe d'algorithme implanté

Vue temporelle du graphe d'algorithme implanté

FIG. 3.4.6 – *Graphes d'algorithmes implantés*

3.5 Algorithme de l'heuristique avec retour-arrière

3.5.1 Principes

L'algorithme d'ordonnancement glouton a réalisé une première implantation qu'on appelle solution initiale. A une étape donnée de l'heuristique gloutonne, il peut exister plusieurs solutions partielles équivalentes que la fonction de coût, c'est-à-dire la pression d'ordonnancement, n'a pas permis de départager. Un choix aléatoire est alors effectué à chaque étape contenant des opérations équivalentes et ce sont ces choix qui conduisent à une solution initiale. L'implantation solution de l'algorithme d'application initial est ensuite améliorée par une méthode de retour-arrière, qui est une méthode de voisinage de recherche locale. Cette méthode consiste à désordonnancer les opérations ordonnancées entre la dernière étape de

l'heuristique et une étape où un choix aléatoire a été fait, puis à ordonnancer une autre opération parmi les opérations équivalentes, et enfin à construire une autre implantation avec la même méthode que précédemment. Le retour arrière permet de supprimer une partie de l'aléatoire dans l'algorithme d'implantation en évaluant, en termes de durée d'exécution, des solutions initiales construites à partir de chacune des opérations équivalentes. Ces dernières vont constituer le voisinage que le processus de retour-arrière va explorer. Le voisinage est ici déterminé au fur et à mesure du déroulement du processus d'ordonnancement, ce qui n'est pas le cas de l'algorithme FAST décrit dans [49] qui détermine avant l'ordonnancement les nœuds sur lesquels il va falloir revenir.

La méthode avec retour-arrière a l'avantage de pouvoir différencier des opérations qui n'avaient pas pu l'être par l'heuristique gloutonne car elles avaient des valeurs voisines de fonction de coût. Pour cette méthode, elles ont toujours même valeur de fonction de coût mais ce qui change, c'est la longueur du chemin critique qu'elles engendrent. Ainsi, la fonction de coût de la méthode avec retour-arrière permet de départager les implantations entre elles et non les opérations comme c'était le cas pour une implantation gloutonne.

Définition 62 *Une étape de l'heuristique est une* profondeur *pour la méthode avec retour-arrière si et seulement si il existe plusieurs opérations équivalentes, c'est-à-dire non départagées par la fonction de coût à cette étape.*

Il se peut qu'il y ait plusieurs profondeurs associés à une implantation. Dans ce cas, quelle(s) profondeur(s) choisir? On peut choisir la plus grande, la plus petite, une profondeur moyenne... De plus, quand on construit une nouvelle implantation on peut de nouveau avoir d'autres profondeurs, ce qui accroît la complexité de la méthode. Nous avons choisi de n'explorer que les voisinages construits lors de la première implantation, ceci afin de limiter l'explosion combinatoire de la méthode mais par contre, on a choisi de les explorer tous.

3.5.2 Construction de l'implantation initiale

L'implantation initiale est construite en utilisant les mêmes principes que l'heuristique gloutonne, en mémorisant en plus, pour chaque profondeur, l'ensemble des opérations équivalentes. Le tableau 3.5.6 donne plus précisément l'algorithme de construction de l'implantation initiale.

3.5.3 Amélioration de cette implantation

Une fois l'implantation initiale construite, on cherche dans son voisinage une meilleure implantation. Pour ce faire, on désordonnance les opérations implantées entre les étapes `nb_etapes` et la première profondeur non encore explorée rencontrée. On sélectionne à l'étape

Construction de l'implantation initiale
Initialisation de la liste courante des candidats avec les opérations sans prédécesseur Initialisation du nombre d'étapes de l'heuristique :`nb_etapes=0` `TantQue` la liste courante n'est pas vide ♦ Incrémentation du nombre d'étapes ♦ Création d'une liste contenant la liste des candidats équivalents à l'étape considérée ♦ Implantation du premier candidat de cette nouvelle liste ♦ Suppression de cette opération de la liste courante et ajout à la liste courante de ses successeurs ordonnançables `FindeTantQue` Initialisation du meilleur ordonnancement avec la longueur du chemin critique de l'implantation calculée et du nombre d'étapes `nb_etapes`

TAB. 3.5.6 – *Algorithme d'ordonnancement avec retour-arrière : construction de l'implantation initiale*

égale à la profondeur, une nouvelle opération non encore implantée à cette étape, parmi les opérations équivalentes. Une nouvelle liste d'opérations est ensuite construite avec l'ensemble des opérations équivalentes privé de celle qui a été sélectionnée et avec l'ensemble des successeurs ordonnançables de l'opération, puis tant que cette liste n'est pas vide on ordonnance une à une les opérations en suivant le même principe qu'une heuristique gloutonne. Le détail de l'algorithme est donné dans la table 3.5.7.

Après chaque nouvelle implantation construite, on teste si la longueur de son chemin critique est inférieure ou égale au meilleur chemin critique mémorisé et si tel est le cas, cette nouvelle implantation est mémorisée et servira de référence pour le retour-arrière suivant, si ce n'est pas le cas, on garde comme meilleure implantation, l'implantation précédente. Ainsi, on refuse d'accepter une implantation qui serait moins bonne que celle mémorisée.

3.5.4 Exemples

L'égaliseur

Le meilleur ordonnancement de l'égaliseur (cf. figures 3.5.7, 3.5.4) est égal à 70 et est obtenu par la méthode de retour-arrière en remontant jusqu'à une profondeur égale à deux et en choisissant le deuxième candidat équivalent. Le retour-arrière construit deux implantations seulement et améliore la longueur d'ordonnancement de 6%.

Recherche des étapes où le retour-arrière est possible, c'est-à-dire les étapes, encore appelées profondeurs, où le nombre de candidats équivalents est plus grand que un.

`Pour` toutes les étapes où le retour-arrière est possible :

- ♦ `Pour` tous les candidats équivalents associés à une profondeur donnée
  - ◊ Désordonnancement des opérations implantées entre les étapes `nb_etapes` et `profondeur`
  - ◊ Ordonnancement d'un candidat équivalent
  - ◊ Création d'une nouvelle liste de candidats : ensemble des candidats équivalents à la profondeur étudié et différent du candidat précédemment ordonnancé, plus les successeurs ordonnançables de ce candidat qui ont été implantés entre les étapes `nb_etapes` et `profondeur+1` .
  - ◊ `TantQue` la liste n'est pas vide
    - △ Implantation d'une opération de la liste sur le meilleur processeur
    - △ Suppression de cette opération et ajout de ses successeurs ordonnançables
  - ◊ `FindeTantQue`
  - ◊ `Si` la longueur du chemin critique de l'implantation courante est inférieure à la meilleure longueur mémorisée `alors`
    - △ Mise à jour de la meilleure longueur, de la profondeur du retour-arrière associée et du numéro du candidat équivalent correspondant
- ♦ `FindePour`

`FindePour`

TAB. 3.5.7 – *Algorithme d'ordonnancement avec retour-arrière : amélioration de l'implantation*

Le "gmdf"

Le tableau 3.5.8 détaille le déroulement de la méthode de recherche locale avec retour-arrière appliquée au graphe de la figure 3.5.9. Chaque case de la première colonne du tableau désigne une profondeur du retour-arrière. Si on considère par exemple la profondeur 17 de l'heuristique, il y a 2 opérations équivalentes et pour l'étape 26, il y en a 4. La valeur de la case [profondeur][numéro de candidat] du tableau est égale à la longueur du chemin critique de l'implantation obtenue. L'implantation initiale a été construite en prenant à chaque profondeur le candidat 1 pour l'ordonnancement. C'est pourquoi, les longueurs de chemin critique données dans la deuxième colonne du tableau sont toutes égales. Ensuite, pour les autres implantations, de l'étape 1 à la profondeur-1 choisie, les choix d'implantation effectués sont identiques à ceux de l'implantation initiale, puis à l'étape égale à la profondeur

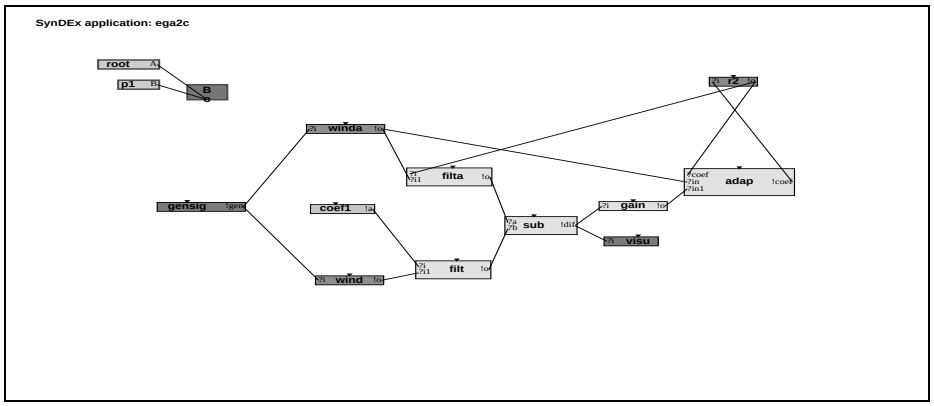
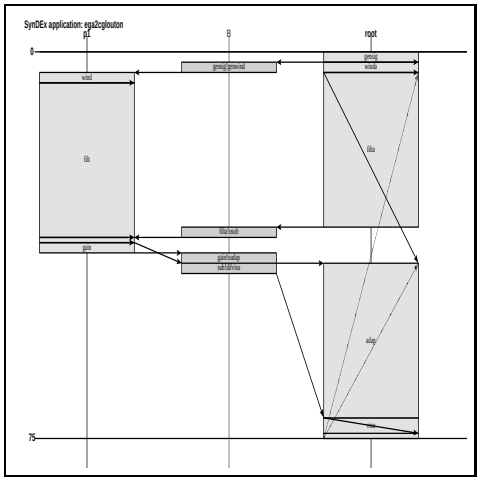


FIG. 3.5.7 – *Graphes des algorithme et architecture de l'égaliseur*



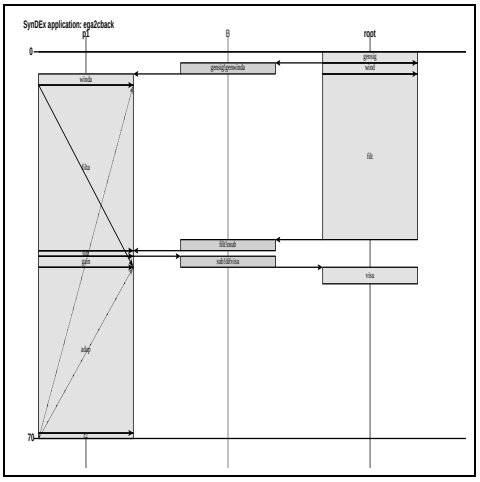


FIG. 3.5.8 – *Graphes de l'égaliseur implantés construits avec une méthode gloutonne puis avec une méthode de retour-arrière*

une autre opération parmi les 1, 2, 3 ou 4 est sélectionnée pour être ordonnancée à cette étape et on regarde à quelle implantation on arrive.

Le meilleur ordonnancement égal à 39262 est obtenu à la profondeur 29 sur le candidat numéro : 2. Le nombre d'implantations construites est égal à 31, ce qui correspond au nombre de longueurs de chemin critique figurant dans le tableau, et la méthode améliore la longueur d'ordonnancement de 7,25%.

Etapes de l'heuristique	Numéros de candidats équivalents			
	1	2	3	4
5	42334	40414	40414	
6	42334	40414		
17	42334	40414		
26	42334	42462	42462	40414
28	42334	41310	39262	
29	42334	*39262*		
32	42334	41310	41310	
33	42334	41310		
36	42334	41310	41310	
37	42334	41310	41310	
38	42334	41310		
40	42334	41310		

TAB. 3.5.8 – *Influence du retour-arrière sur la longueur d'ordonnancement du graphe gmdf décrit dans la figure 3.5.9*

3.6 Comparaison d'heuristiques sur des exemples d'application

Nous allons comparer sur des exemples [84] notre heuristique avec celles décrites dans la littérature et présentées brièvement dans le chapitre état de l'art de la page 108 à la page 112. Bien que certaines n'appartiennent pas tout à fait à la même classe que la nôtre, nous restreindrons nos hypothèses pour se retrouver dans le même cas qu'elles.

Soit σ, notre heuristique initiale et $\sigma+RA$, l'heuristique avec retour arrière. Pour éviter la confusion entre durée d'exécution de l'algorithme d'application et durée d'exécution de l'algorithme de distribution/ordonnancement, nous utiliserons dans la suite le terme longueur d'ordonnancement pour désigner la durée d'exécution de l'algorithme d'application.

Les résultats des longueurs d'ordonnancement du graphe de l'algorithme d'application de la figure 3.6.12 en utilisant les données de [48] sont donnés dans la table 3.6.9. Les algorithmes d'ordonnancement comparés sont de catégorie BNP et UNC, ils supposent donc une architecture complètement connectée et ne prennent pas en compte l'ordonnancement et le routage des communications. Le nombre de processeurs est égal à deux.

On constate que notre heuristique donne la plus petite longueur d'ordonnancement pour cet exemple.

Les algorithmes d'ordonnancement comparés dans [48] appartiennent aux deux premières

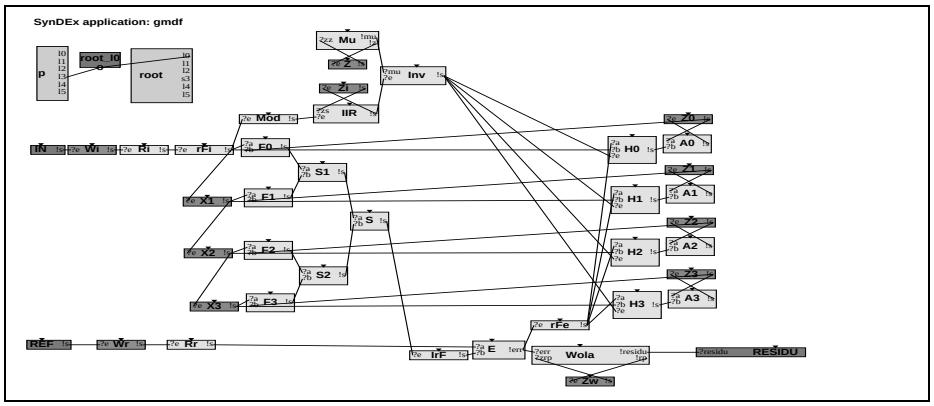


FIG. 3.5.9 – *Graphes des algorithme et architecture de gmdf*

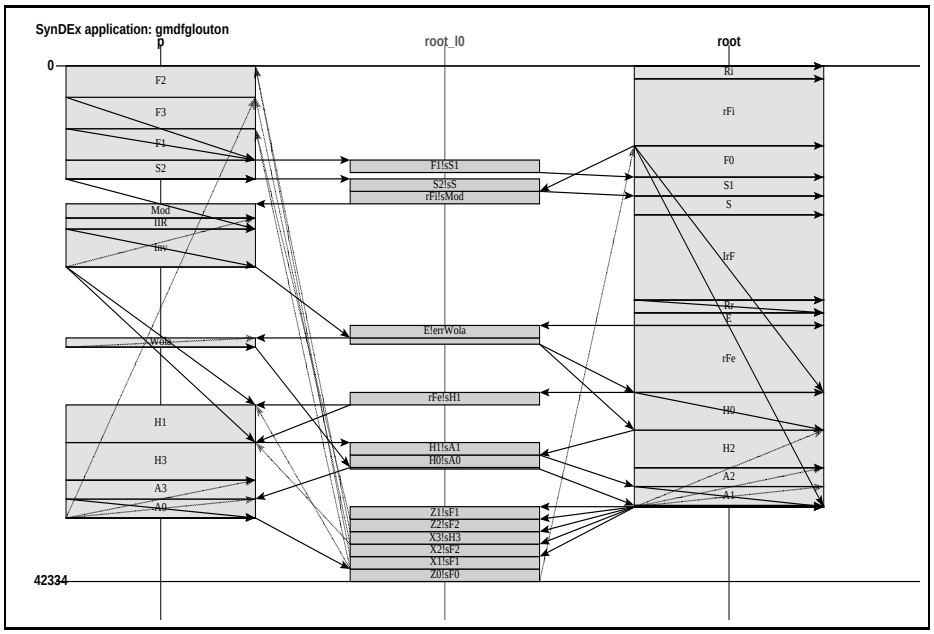


FIG. 3.5.10 – *Graphe de gmdf implanté construit avec une méthode gloutonne*

catégories : BNP et UNC et sont testés sur le graphe d'algorithme de la figure 3.6.13. Ni le routage, ni l'ordonnancement des communications ne sont donc effectués. L'architecture est un réseau de quatre processeurs, elle est complètement connectée. Pour ne pas être pénalisés par le routage et l'ordonnancement des communications qu'effectuent σ, nous avons choisi une architecture avec deux liens entre chaque processeur. Les résultats sont donnés dans la table 3.6.10.

Dans les conditions ainsi définies, notre algorithme d'ordonnancement arrive en deuxième

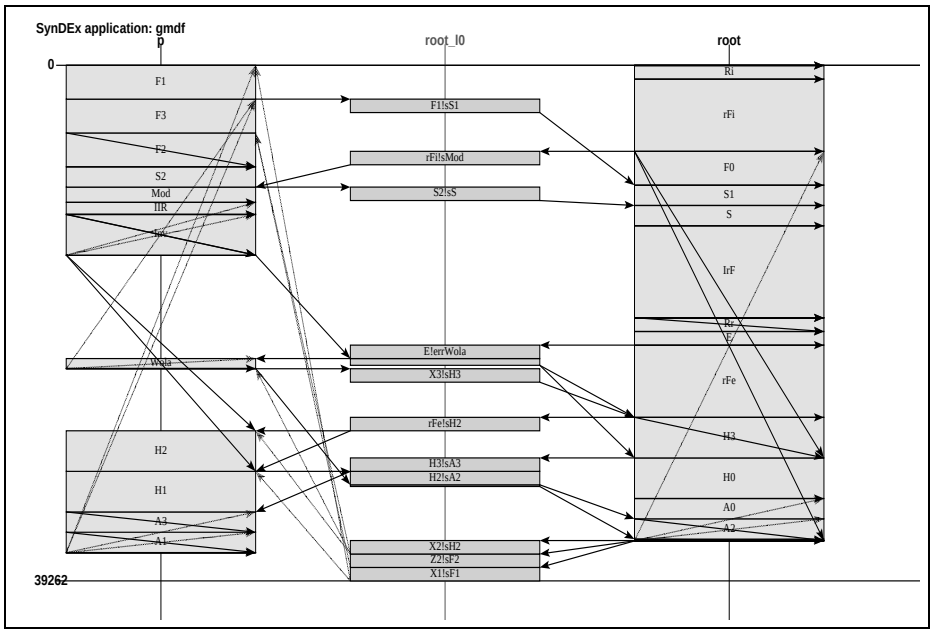


FIG. 3.5.11 – *Graphe de gmdf implanté construit avec une méthode de retour-arrière*

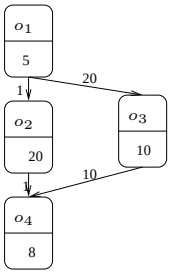


FIG. 3.6.12 – *Graphe d'algorithme 1*

position pour cet exemple.

Les algorithmes d'ordonnancement comparés dans l'exemple de la figure 3.6.14 appartiennent à la catégorie APN : l'architecture n'est pas forcément complètement connectée, les communications peuvent donc être routées et, enfin, les communications sont ordonnancées. La topologie choisie dans cet exemple présenté dans [46, 47] est un anneau à quatre processeurs. Le routage est donc nécessaire mais, contrairement à l'algorithme σ, les trois autres algorithmes d'ordonnancement comparés BSA, BU et BH, supposent des coûts de transfert constants quel que soit le nombre de liens utilisés pour ce transfert. Les résultats issus de [47, 46] sont donnés dans la table 3.6.11.

BSA est le meilleur algorithme pour la minimisation de la longueur de l'ordonnancement

Type d'algorithmes	ETF	EZ, MD	σ
longueur d'ordonnancement	43	35	34

TAB. 3.6.9 – *Comparaisons d'algorithmes d'ordonnancement*

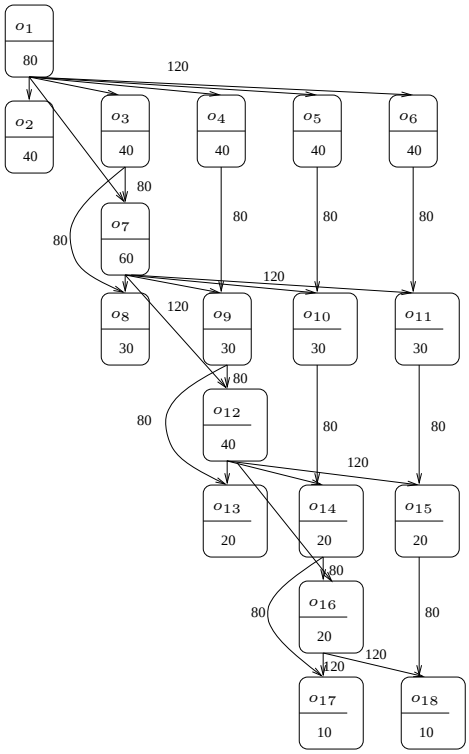


FIG. 3.6.13 – *Graphe d'algorithme 2*

de l'exemple de la figure 3.6.14, avec une longueur de 16. $\sigma + RA$ donne une longueur d'ordonnancement égale à 18, ce qui lui confère la place de deuxième. Si l'on compare les algorithmes d'ordonnancement non pas sur la longueur d'ordonnancement mais sur la somme de la longueur d'ordonnancement et des durées des communications comme le propose Kwok et Ahmad dans [47, 46], l'algorithme $\sigma + RA$ est le meilleur pour cet exemple (cf. schéma ci-après). On peut aussi constater que l'algorithme σ sans retour arrière donne des résultats honorables. En effet, il se place en troisième position pour la longueur de l'ordonnancement, il se place également troisième pour la somme de la longueur d'ordonnancement et des durées des communications.

Conclusion

Grâce à une modélisation fine des modèles d'algorithme d'application et d'architecture, nous avons obtenu un modèle d'implantation où à la fois les calculs et les communications

Type d'algorithmes	EZ	ETF	MD	DCP	σ
longueur d'ordonnancement	600	520	460	440	450

TAB. 3.6.10 – *Comparaisons d'algorithmes d'ordonnancement*

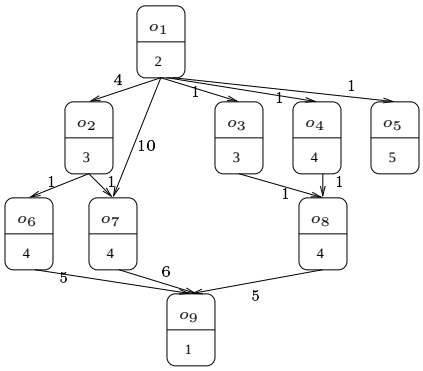


FIG. 3.6.14 – *Graphe d'algorithme 3*

sont distribués et ordonnancés (cf. première partie de la thèse modélisation relationnelle). Le modèle d'implantation ainsi obtenu prend en compte des architectures réalistes, conduisant à des optimisations plus précises. De plus, cela est important pour la génération d'exécutif temps réel qui peut découler de l'implantation. A l'issue de la formalisation, nous avons construit l'ensemble des implantations valides d'un algorithme d'application donné sur une architecture donnée. Pour faire un choix d'implantation parmi cet ensemble, nous avons proposé une extension avec retour-arrière d'une heuristique gloutonne rapide. La rapidité de l'heuristique gloutonne permet de faire du prototypage rapide optimisé d'applications soumises à des contraintes temps réel. Les performances des heuristiques de distribution/ordonnancement avec et sans retour-arrière ont été évaluées en comparant, avec d'autres heuristiques de distribution/ordonnancement, les longueurs d'ordonnancement obtenues sur quelques exemples d'algorithmes d'application. Ces performances sont satisfaisantes sur ces exemples, il reste maintenant à valider statistiquement ces résultats à un grand nombre de graphes d'application.

Type d'algorithmes	BSA	BU	MH	σ	$\sigma + RA$
longueur d'ordonnancement	16	24	20	20	18
Somme des durées des communications	11	27	16	14	8
Total	27	51	36	34	26

TAB. 3.6.11 – *Comparaisons d'algorithmes d'ordonnancement*

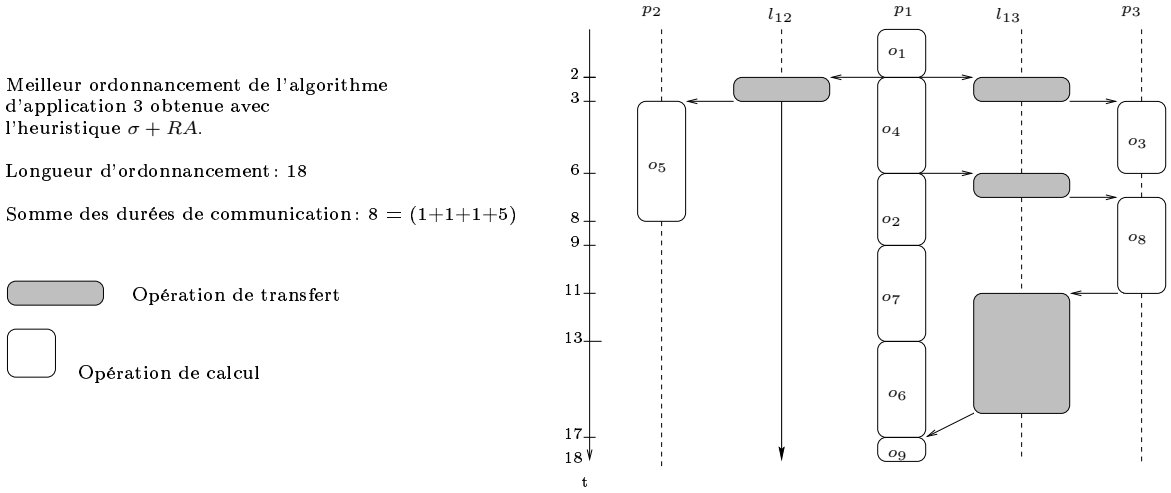


FIG. 3.6.15 – *Algorithme ordonnancé donné par l'heuristique avec retour-arrière*

Troisième partie

Développement logiciel pour SynDEx

1. Présentation du logiciel SynDEx

1.1 Introduction

La première partie de la thèse a décrit les modèles d'algorithme, d'architecture et d'implantation (distribution et ordonnancement) de l'algorithme sur l'architecture. A l'issue de cette partie, l'ensemble des implantations valides d'un algorithme sur une architecture a été construit. Mettre en adéquation de manière efficace l'algorithme avec l'architecture, c'est-à-dire trouver une implantation de durée d'exécution respectant la contrainte temps réel fixée par l'environnement est un problème NP-difficile résolu par une méthode approchée pouvant être gloutonne puis éventuellement améliorée par une méthode de voisinage de recherche locale. La deuxième partie décrit ces méthodes de résolution après avoir fait un état de l'art permettant de se situer dans le contexte. Le logiciel SynDEx [53, 51] –Exécutif Distribué Synchrone– automatise partiellement la résolution de ce type de problème d'ordonnancement et permet de générer [34, 52] un exécutif qui supporte l'exécution de l'algorithme sur l'architecture. Dans la troisième partie, nous allons spécifier les deux heuristiques que nous avons formalisées dans la deuxième partie de la thèse et qui sont implantées dans le cœur de SynDEx V5.

1.2 Présentation de l'environnement logiciel SynDEx

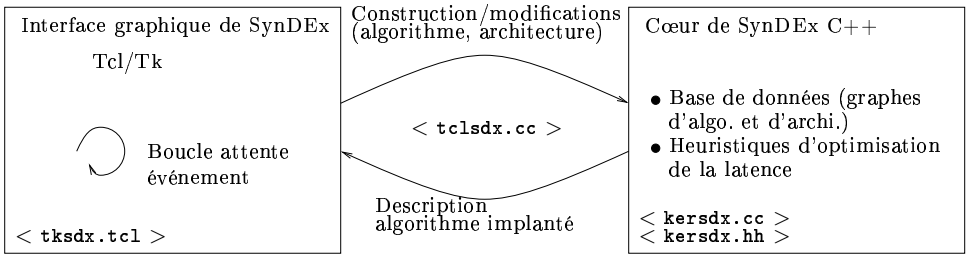


FIG. 1.2.1 – *Fonctionnement logiciel de SynDEx*

SynDEx est un environnement logiciel graphique interactif de développement pour applications temps réel de commande, de traitement du signal et des images, supportant la méthode d'Adéquation Algorithme Architecture. Il permet, à partir de la spécification de

l'algorithme et de l'architecture de conduire à une implantation efficace sur machine multiprocesseur, c'est-à-dire respectant des contraintes temps réel et minimisant les ressources matérielles (nombre de processeurs et de liaisons de communication). Enfin, il génère automatiquement des exécutifs optimisés gérant entre autre un système de communications sans inter-blocage. Ceci décharge l'utilisateur de la programmation bas niveau et supprime les tests en multiprocesseur, dans le cas ou l'on suppose le matériel sans panne.

SynDEx a d'abord été écrit en Smalltalk par C. Lavarenne pour les versions 1 à 4, puis en Tcl/Tk et C++ par T. Grandpierre [32] pour la version 5.

Le logiciel SynDEx est composé de deux parties distinctes, une interface homme machine graphique écrite en Tcl/Tk et un cœur écrit en C++. L'interface permet de saisir les graphes d'algorithme et d'architecture et de visualiser le graphe de l'algorithme implanté. Le cœur de SynDEx contient, d'une part la base de données associée aux graphes d'algorithme et d'architecture saisis dans l'IHM, et d'autre part les heuristiques d'optimisation de la latence qui construisent un graphe d'algorithme implanté dont la description est transmise à l'IHM. Le fonctionnement de SynDEx est illustré dans la figure 1.2.1.

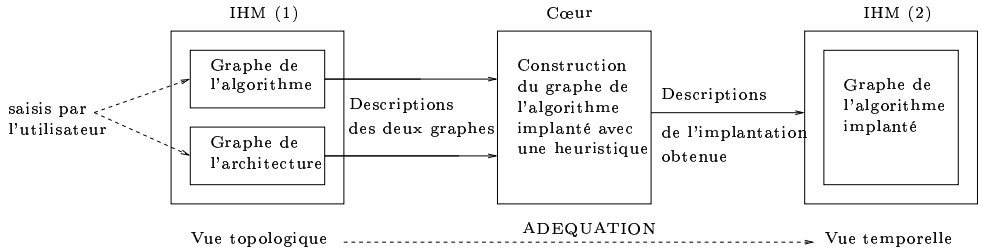


FIG. 1.2.2 – *Les deux vues de l'IHM de SynDEx*

L'IHM possède deux vues, une vue topologique et une vue temporelle. Ces deux vues sont décrites dans la figure 1.2.2. La vue topologique permet à l'utilisateur de saisir les deux graphes algorithme et architecture.

Remarque 62 *Le graphe de l'algorithme peut aussi être importé à partir d'un fichier produit lors de la compilation d'un programme SIGNAL avec l'option de génération de code SynDEx [9], ceci pour la version 4 seulement.*

La vue temporelle permet de visualiser le graphe de l'algorithme implanté construit à l'aide d'une des heuristiques d'optimisation présentes dans le cœur de SynDEx. Un exemple [82] de vue topologique est donné figure 1.2.3 et la vue temporelle associée est donnée figure 1.2.4.

Dans la figure 1.2.4 le temps est lu selon une échelle verticale et chaque colonne contient les opérations liées à une ressource de type calcul ou de type média. Sur chaque processeur (resp. chaque média), les opérations de calcul (resp. de transfert) sont ordonnancées. Une

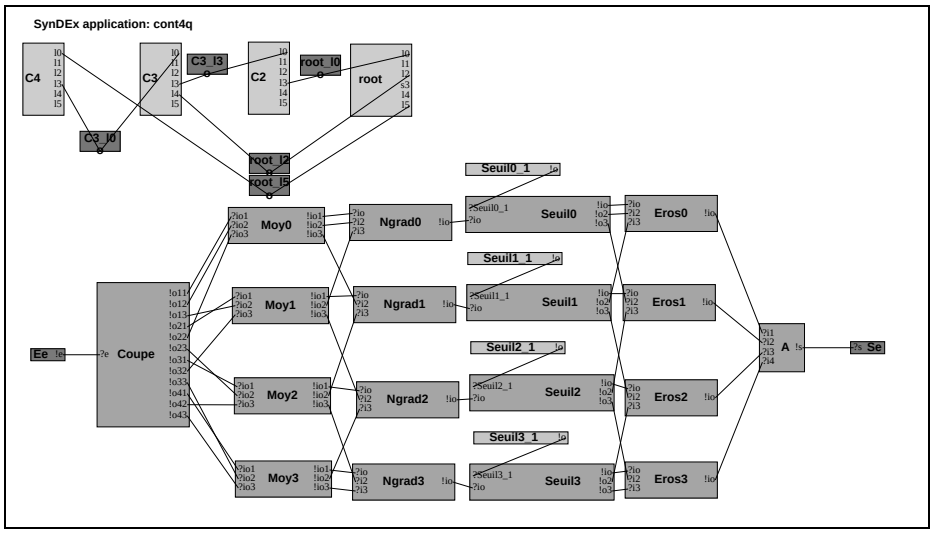


FIG. 1.2.3 – *Environnement SynDEx: vue topologique des graphes de l'algorithme et de l'architecture*

opération est représentée par une boîte de hauteur proportionnelle à sa durée d'exécution. La durée de chaque boîte est mesurée, au préalable, à l'aide d'une horloge temps réelle ou estimée quand ce n'est pas possible. Ces durées sont fournies lors de la spécification des caractéristiques des composants de l'algorithme relativement aux composants de l'architecture. Il est important de noter que la qualité des résultats donnés par l'heuristique d'optimisation dépend de la qualité de la mesure de ces durées.

1.3 Interface graphique de SynDEx

1.3.1 Introduction

L'IHM – Interface Homme Machine – de SynDEx v5 est décrite à l'aide du langage Tcl/Tk [65, 42] spécialement conçu pour les interfaces graphiques. L'interface existante permet de saisir des graphes d'algorithmes non conditionnés. Nous avons intégré à cette interface un nouveau type de dépendances, définies dans la modélisation, *les dépendances de conditionnement*, encore appelées *conditions d'activation* qui permettent de modéliser un modèle d'algorithme conditionné. Nous allons détailler au paragraphe suivant ces modifications.

1.3.2 Modifications de l'interface pour supporter le conditionnement

Le graphe de l'architecture est inchangé, le graphe de l'algorithme est enrichi par les conditions d'activation que nous avons ajoutées. On trouvera en annexes, la procédure

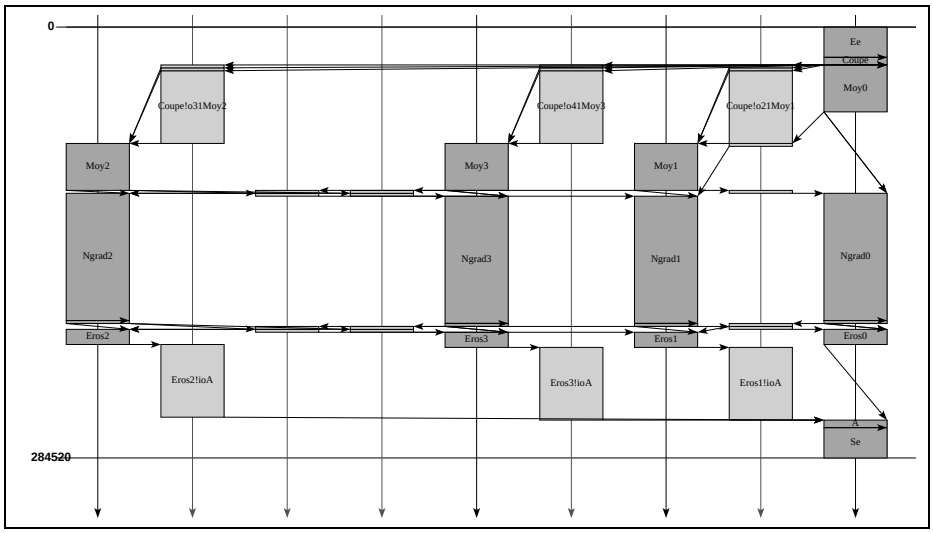


FIG. 1.2.4 – *Environnement SynDEx : vue temporelle du graphe de l'algorithme implanté*

`AjoutActi` écrite en Tcl/Tk qui ajoute les conditions d'activation de manière implicite à chaque sommet du graphe de l'algorithme.

Comme on l'a vu dans la modélisation, nous avons choisi de représenter le conditionnement du graphe de l'algorithme par des conditions d'activation encore appelées arcs de dépendances de conditionnement. Le graphe de l'algorithme possède deux sortes d'arcs de dépendances, les arcs de dépendances de données et les arcs de dépendances de conditionnement. Chaque dépendance de conditionnement est un arc booléen induisant une condition d'activation sur l'opération réceptrice. L'opération réceptrice ne sera exécutée que si la condition d'activation est vraie. Les dépendances de conditionnement doivent se différencier des dépendances de données. La convention graphique existante associée à une opération est que ses dépendances de données sont connectées aux bords latéraux du rectangle représentant l'opération, le bord gauche est utilisé pour les entrées et le bord droit pour les sorties. La convention graphique choisie est que la dépendance de conditionnement associée à chaque opération soit connectée au bord supérieur du rectangle représentant l'opération réceptrice (cf. figure 1.3.5).

Nous allons maintenant définir plus précisément ces conditions d'activation.

Conditions d'activation

Chaque dépendance (de conditionnement comme de donnée) est connectée à une opération par un *port*. Si l'opération est la source (émettrice) de la dépendance, la connexion se fait par l'intermédiaire d'un *port de sortie*, sinon, l'opération est la destination (réceptrice)

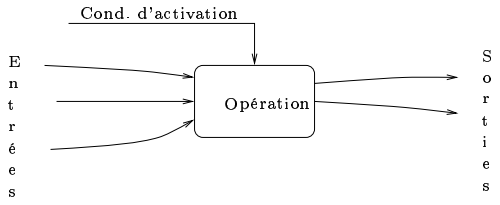


FIG. 1.3.5 – *Convention graphique*

de la dépendance et la connexion se fait par l'intermédiaire d'un *port d'entrée*.

Pour différencier une dépendance de conditionnement d'une dépendance de donnée simple, le port d'entrée de l'opération réceptrice de la dépendance, appelée condition d'activation, est un port d'entrée particulier réservé uniquement pour les dépendances de conditionnement et est appelé *port d'activation* ou *port de conditionnement*. Si ce port est connecté, l'opération est conditionnée. Toute opération du graphe doit donc posséder, au moment de sa création, un tel port en plus de tous ses autres ports décrits par l'utilisateur. Ainsi, au niveau de l'IHM, la création de ce port doit être implicite.

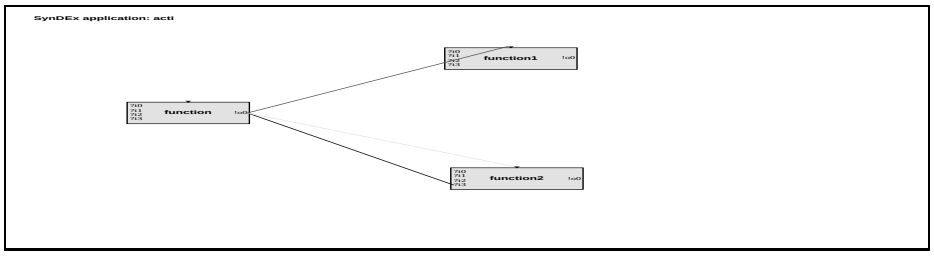


FIG. 1.3.6 – *Environnement SynDEx : conditions d'activation*

Modifications de la base de données C++

La base de données existante, décrite en C++ [77], est modifiée pour pouvoir différencier les dépendances de données simples des dépendances de conditionnement. Pour les graphes d'algorithme et d'architecture, des classes C++ ont été définies. Chaque objet d'une classe est composé d'un ensemble de méthodes qui permettent de modifier ou de donner l'état de l'objet. La convention donnée par T. Grandpierre dans [32] et que nous allons utiliser par la suite est la suivante : les méthodes dont le nom commence par un point d'interrogation **?** sont des méthodes d'accès qui retournent un aspect de l'état de l'objet (sans le modifier) et les méthodes dont le nom commence par un point d'exclamation **!** sont des méthodes de création ou de modification qui peuvent modifier l'état interne de l'objet.

Par exemple, la classe `OPERATION` contient l'ensemble des objets opérations de calcul et de transfert du graphe de l'algorithme. De même, la classe `OPERATEUR` contient l'ensemble des objets ressources de type processeur ou média du graphe de l'architecture. La méthode`?get_duree()` retourne la durée de l'opération et la méthode `!set_duree(delta)` modifie la durée de l'opération et lui donne la valeur delta.

Voyons maintenant comment sont codées les dépendances du graphe de l'algorithme. Les ports des opérations appartiennent à une même classe : la classe PORT-LOGICIEL, cette classe contient deux sous-classes, la sous-classe PORT-IN et la sous-classe PORT-OUT. Chaque instance de la classe PORT-IN est un port d'entrée et chaque instance de la classe PORT-OUT est un port de sortie. Une dépendance de donnée est caractérisée par une connexion entre un port de sortie d'une opération émettrice et au moins un port d'entrée d'une opération réceptrice. Pour différencier les dépendances de données simples des dépendances de conditionnement, il faut parmi les ports d'entrée, distinguer le port d'entrée de conditionnement. Pour ce faire, nous créeons une classe PORT-ACTI qui hérite de la classe PORT-IN (cf. figure 1.3.7). Ainsi, si le port de conditionnement est connecté, l'opération qui le possède est conditionnée. La définition de cette classe en C++ est donnée en annexe à la page 172.

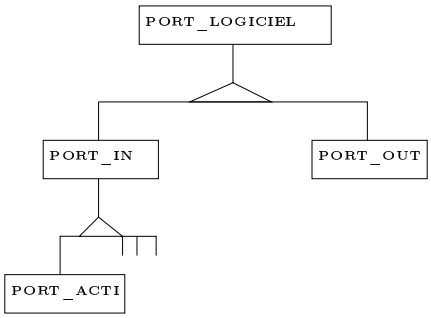


FIG. 1.3.7 – *Nouvelle classe PORT-ACTI*

Chaque instance de la classe PORT-ACTI correspond au port d'activation d'une opération. Le port d'activation possède un champ supplémentaire, appelé *polarité*, correspondant à la polarité du conditionnement 1 (vrai) ou 0 (faux) du booléen qui conditionne l'opération. Cette polarité est mise à jour par la méthode `!set_pol_acti()` et est consultable par `?get_pol_acti()`. Au moment de la création d'un port d'activation, la polarité est par défaut mise à 1. Au niveau Tcl/Tk, en cliquant sur un port d'entrée d'activation, nous avons la possibilité de modifier la polarité (cf. figure 1.3.8). Si la polarité est égale à 1, la dépendance de conditionnement associée est représentée dans la fenêtre édition par un arc vert et si la polarité vaut 0, l'arc de dépendance de conditionnement est dessiné en rouge.

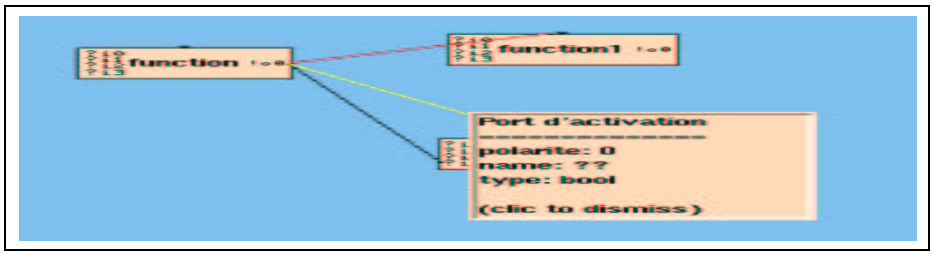


FIG. 1.3.8 – *Environnement SynDEx: conditions d'activation*

Le port d'activation possède également deux autres méthodes `?get_next_port()` et `!set_next_port()` qui permettent, à opération donnée, d'accéder ou de définir l'éventuel port de conditionnement qui succède à son port de conditionnement dans une branche de conditionnement donnée.

2. Heuristiques de conditionnement implantées dans SynDEx

2.1 Codage de l'ordonnancement conditionné

Une des phases de compilation consiste à générer un ordonnancement conditionné. Pour ce faire, on introduit une nouvelle classe C++ d'opérations dans le cœur de SynDEx, appelée *conditionnelle*, pour supporter l'ordonnancement conditionné. Cette classe ne correspond pas, contrairement à la classe OPERATION, à une opération visible par l'utilisateur dans l'interface graphique. Chaque conditionnelle possède une référence à un port p de sortie booléen de conditionnement, et deux branches d'ordonnancement séquentiel exclusives, une pour les opérations conditionnées par p et dont le port d'activation à une polarité égale à 1 (resp. 0), dite *branche1* (booléen de conditionnement vrai), (resp. *branche0* (booléen de conditionnement faux)).

Nous avons créé une nouvelle sous-classe de la classe OPERATION qui contient les opérations de calcul et de transfert du graphe de l'algorithme. Cette nouvelle classe s'appelle la classe OP_COND (cf. figure 2.1.1). Chaque instance de cette classe est une opération dite *conditionnelle*, qui sera souvent notée par la suite de manière abrégée *cond.*.

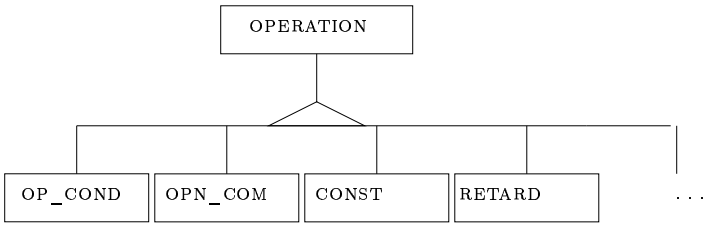


FIG. 2.1.1 – *Nouvelle classe OP_COND*

Toute opération conditionnelle est une opération qui possède deux champs supplémentaires : `last_true`, dernière opération ordonnancée sur la branche1 et `last_false`, dernière opération ordonnancée sur la branche0. L'opération `last_true` (resp. `last_false`) est accessible par la méthode `?get_last(1)` (resp. `?get_last(0)`). Ces opérations sont mises à jour par la méthode `!set_last(polarite, opération)`, polarité étant égal à un (resp. zéro) si c'est `last_true` (resp. `last_false`) qui est mis à jour. Chaque opération répond par

ailleurs aux méthodes `?get_prev_scheduled()` et `!set_prev_scheduled(opération)` qui permettent respectivement d'obtenir et de changer l'opération qui précède dans l'ordonnancement celle à laquelle s'applique la méthode.

La date de fin au-plus-tôt d'une opération *cond.* est égale à la plus grande des dates de fin des opérations `last_true` et `last_false` de cette *cond.*.

2.2 Visualisation de l'ordonnancement conditionné

Pour l'instant on a choisi de ne visualiser pour chaque opération conditionnelle qu'une des deux branches et plus précisément celle dont la date de fin correspond à la date de fin de la conditionnelle, ce qui revient à visualiser "la plus longue des deux branches".

2.3 Exemple

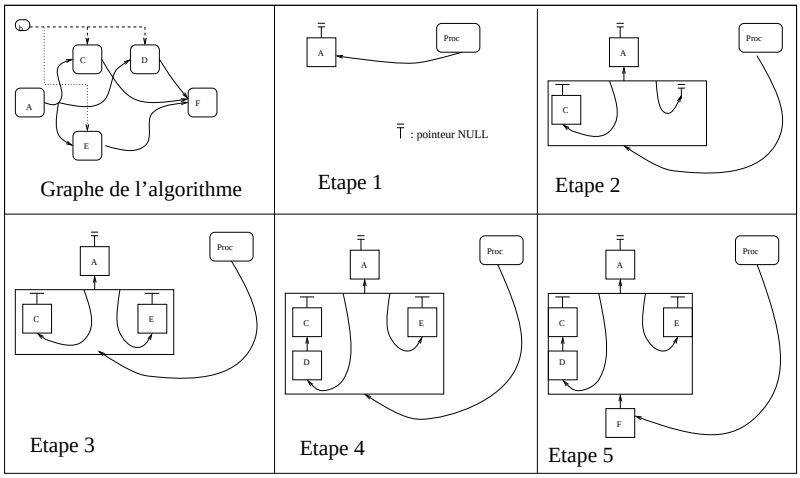


FIG. 2.3.2 – *Exemple*

La liste des opérations ordonnancées sur chaque ressource de type processeur ou média est mémorisée par une conditionnelle accessible par la méthode `?get_owner()`. Soit c0 la conditionnelle du processeur Proc.

- Etape 1 : L'opération A est implantée du côté de la polarité 1 de c0 :

  `c0 ←?get_owner(Proc)`

  `!set_last(c0, 1, A)`

  `!set_prev_scheduled(A, NULL)`

- Etape 2 : Création d'une opération conditionnelle c1 (représentée sur la figure par une boîte). L'opération C est implantée du côté de la polarité 1 de c1 :

```
!set_last(c0, 1, c1)
!set_last(c1, 1, C)
!set_prev_scheduled(C, NULL)
```

– Etape 3 : L'opération E est implantée du côté de la polarité 0 de c1 :

```
!set_last(c1, 0, E)
!set_prev_scheduled(E, NULL)
```

– Etape 4 : L'opération D est implantée du côté de la polarité 1 de c1 après l'opération C :

```
!set_last(c1, 1, D)
!set_prev_scheduled(D, C)
```

– Etape 5 : L'opération F est implantée du côté de la polarité 1 de c0 après $c1$:

```
!set_last(c0, 1, F)
!set_prev_scheduled(F, c1)
```

2.4 Heuristique gloutonne

2.4.1 Algorithme de construction de la branche de conditionnement d'une opération

L'algorithme, appelé algorithme *racine*, consiste à construire la branche de conditionnement d'une opération en remontant l'arborescence de conditionnement tout en mémorisant les ports d'activation par lesquels on pourra redescendre en suivant la même branche, et à retourner le plus haut port d'activation de l'arborescence. La remontée dans l'arborescence à partir d'une opération donnée est unique par construction de l'arborescence, en revanche, la descente ne l'est pas, c'est pourquoi les ports d'activation rencontrés dans la montée sont marqués pour pouvoir redescendre par la même branche et atteindre à nouveau l'opération étudiée. Le principe de l'algorithme est illustré figure 2.4.3.

Si l'opération n'est pas conditionnée, son port d'activation n'est pas connecté, l'algorithme retourne le pointeur NULL. Si l'opération est conditionnée par un unique booléen, l'algorithme racine retourne le port d'activation de l'opération. Le principe de cet algorithme est donné dans la table 2.4.1.

L'objet retourné par l'algorithme racine correspond au plus haut port d'activation, correspondant à une opération conditionnée, de l'arborescence à laquelle appartient l'opération.

Dans la figure 2.4.3, on cherche à construire la branche de conditionnement de l'opération E. On part de E et on initialise son port suivant à NULL (P_S =NULL) et on remonte à son unique prédécesseur de conditionnement D (P_P =D). Une fois arrivé à D, le champ P_S

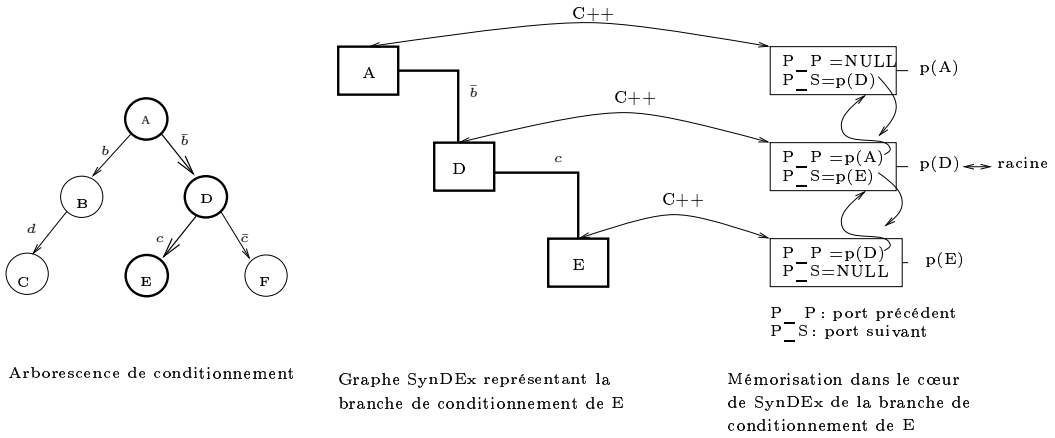


FIG. 2.4.3 – *Construction de la branche de conditionnement d'une opération*

Initialisation du `port_courant` avec le port d'activation de l'opération
Initialisation du champ `port_suivant` du `port_courant` à NULL
`TantQue` le `port_de_sortie` auquel est connecté le `port_courant` est différent de NULL

- Mise à jour du `port_précédent` avec le port d'activation appartenant à l'opération qui possède ce `port_de_sortie`
- Mise à jour du champ `port_suivant` du `port_précédent` avec le `port_courant`
- Mise à jour du `port_courant` avec le `port_précédent`

`FindeTantQue`
Retourner le champ `port_suivant` du `port_courant`.

TAB. 2.4.1 – *Algorithme de construction de la branche de conditionnement d'une opération*

de D est mis à E, ceci pour pouvoir redescendre et ainsi de suite jusqu'à ce que l'on ne puisse plus remonter. A est l'opération la plus haute dans la branche de conditionnement de E, D est l'opération conditionnée la plus haute dans l'arborescence, et donc le port d'activation de D constitue la racine de E.

2.4.2 Algorithme d'ordonnancement conditionné d'une opération sur un opérateur

Plus précisément, l'algorithme d'ordonnancement conditionné est décrit à la table 2.4.2.

Initialisation de la `polarité` à 1
Initialisation de la `conditionnelle_courante` avec la conditionnelle de l'opérateur
Construction de la branche d'activation à laquelle appartient l'opération (cf. table 2.4.1)
Initialisation du `port_acti_courant` avec le plus haut port d'activation de la branche
`TantQue` le `port_acti_courant` est non NULL (c'est-à-dire tant que l'ensemble des ports d'activation composant la branche d'activation n'a pas été examiné)

- Soit `opn` la dernière opération ordonnancée du côté de la `polarité` dans la `conditionnelle_courante`
- `Si opn` est NULL *ou*
  `Si opn` n'est pas une conditionnelle *ou*
  `Si opn` est une conditionnelle mais que son port de conditionnement est différent du port de sortie auquel est connecté le `port_acti_courant`, `alors`
  - Création d'une nouvelle conditionnelle sur l'opérateur avec comme port de conditionnement le port de sortie auquel est connecté le `port_acti_courant`
  - Mise à jour de la `polarité` du port d'activation de cette conditionnelle avec la `polarité` courante
  - Mise à jour du champ opérateur de cette conditionnelle créée
  - Ajout de la conditionnelle créée dans la `conditionnelle_courante` du côté de la `polarité`
  - `opn` devient la conditionnelle créée
- `FinDeSi`
- La `conditionnelle_courante` devient `opn`
- La `polarité` devient celle du `port_acti_courant`
- Le `port_acti_courant` devient le port d'activation suivant dans l'arborescence de conditionnement

`FindeTantQue`
Ajout de l'opération étudiée dans la `conditionnelle_courante` du côté de la `polarité`
Création des opérations de transfert de l'opération étudiée vers ses prédécesseurs
Invalidation des dates des successeurs de l'opération étudiée

TAB. 2.4.2 – *Algorithme d'ordonnancement conditionné d'une opération*

2.4.3 Algorithme de désordonnancement conditionné d'une opération sur un opérateur

L'algorithme de désordonnancement conditionné est décrit à la table 2.4.3.

Initialisation de la `polarité` à 1

`Si` l'`opération_étudiée` est conditionnée `alors`

- ♦ Mise à jour de la `polarité` avec la `polarité` de l'opération

Initialisation de la `cond._courante` avec la cond. à laquelle appartient `opération_étudiée`

Suppression des opérations de transfert arrivant sur `opération_étudiée`

Initialisation de `opération_précédant_opération_étudiée` à NULL

Initialisation de `opération_courante` avec la dernière opération ordonnancée du côté de la `polarité` sur la `cond._courante`

`TantQue` `opération_courante` est différente de `opération_étudiée`

- ♦ Mise à jour de `opération_précédant_opération_étudiée` avec `opération_courante`
- ♦ Mise à jour de `opération_courante` avec l'opération la précédant dans l'ordonnancement

`FindeTantQue`

`Si` `opération_précédant_opération_étudiée` est différente de NULL `alors`

- ♦ Mise à jour du champ opération précédente de cette opération avec le champ opération précédente de `opération_étudiée`

`Sinon`

- ♦ Mise à jour du côté de la `polarité` de la dernière opération de la conditionnelle avec le champ opération précédente de `opération_étudiée`
- ♦ `Si` la conditionnelle n'est pas NULL *et*

  `Si` la dernière opération ordonnancée du côté de la polarité 1 est NULL *et*

  `Si` la dernière opération ordonnancée du côté de la polarité 0 est NULL *et*

  `Si` la conditionnelle propriétaire de la conditionnelle n'est pas NULL

  - ◊ Désordonnancement conditionné de la conditionnelle
  - ◊ Suppression de la conditionnelle

  `FindeSi`

`FindeSi`

Mise à NULL du champ conditionnelle propriétaire de `opération_étudiée`

Mise à NULL du champ opération précédente de `opération_étudiée`

Mise à NULL du champ processeur de `opération_étudiée`

Invalidation des dates de `opération_étudiée`

TAB. 2.4.3 – *Algorithme de désordonnancement conditionné d'une opération*

2.5 Heuristique avec retour-arrière

2.5.1 Algorithme de recherche des candidats équivalents

L'heuristique de voisinage avec retour-arrière consiste, comme on l'a vu au chapitre sur l'optimisation de la latence, à construire une solution initiale en notant à chaque étape de l'heuristique les candidats équivalents, puis à revenir sur chacun de ces candidats équivalents et à construire une nouvelle implantation. La table 2.5.4 détaille l'algorithme qui construit la liste des opérations qui n'ont pas pu être départagées par la pression d'ordonnancement à une étape donnée de l'heuristique gloutonne.

Initialisation de ϵ qui va déterminer la taille du voisinage
Initialisation de la liste `liste_eq` des candidats équivalents avec la liste vide
Initialisation de la pression d'ordonnancement maximale `SP_max` à $-\infty$
Restriction de l'ensemble des candidats en utilisant la règle donnée page 130
`Pour` tous les candidats du sous-ensemble
- ♦ Calcul de la pression d'ordonnancement courante `SP_crt`
- ♦ `Si SP_crt` $>$ `SP_max alors`
  - ◊ `SP_max` devient `SP_crt`
- ♦ `FinDeSi`

`FinDePour`
`Pour` tous les candidats du sous-ensemble
- ♦ `Si` $|$`SP_crt` $-$ `SP_max`$| < \epsilon$ `alors`
  - ◊ Ajout du candidat courant à `liste_eq`
- ♦ `FinDeSi`

`FinDePour`
Retourner `liste_eq`

TAB. 2.5.4 – *Algorithme de recherche des candidats équivalents*

L'heuristique gloutonne aurait simplement retourné le premier candidat rencontré qui ait une pression égale à la pression maximale `SP_max`.

Conclusion

Cette thèse se situe dans le cadre des recherches menées par l'équipe AAA du projet Sosso de l'INRIA-Rocquencourt sur les méthodes d'Adéquation Algorithme Architecture pour les systèmes temps réel distribués embarqués. Notre travail de thèse s'inscrit dans un contexte s'étendant de la spécification des systèmes distribués temps réel à la génération correspondante des exécutifs distribués temps réel. Les objectifs de cette thèse étaient d'une part la formalisation et d'autre part l'optimisation de ces systèmes.

Nous avons tout d'abord modélisé finement à la fois l'algorithme, l'architecture et l'ensemble des implantations valides de l'algorithme sur l'architecture. La caractérisation du graphe de l'algorithme implanté en utilisant les caractéristiques de l'architecture, nous a permis, à l'aide d'heuristiques d'optimisation, de choisir, parmi les implantations valides, une implantation optimisée vérifiant la contrainte temps réel de notre système. Cette implantation optimisée va être ensuite utilisée comme base pour la génération automatique de code. Bien que nous ne traitions pas ces aspects dans cette thèse, il est facile de comprendre que, plus la description de l'implantation est fine, c'est-à-dire proche du code exécutable, plus il sera simple de générer ce code.

Nous allons reprendre les trois parties de la thèse en indiquant les principaux apports de chacune d'elles et les perspectives associées [44, 33].

Modélisation

Principaux apports

Pour utiliser au mieux le parallélisme disponible des architectures multiprocesseur, il est intéressant de bien le décrire. Pour ce faire, nous avons choisi un modèle de graphe non orienté. Nous avons décrit l'architecture comme un hypergraphe non orienté où les sommets sont les unités de calcul et de communication, et les arêtes sont les liaisons entre ces unités. Nous avons modélisé les liaisons entre unités de communication de processeurs différents comme étant des liaisons SAM ayant une capacité finie.

Il est intéressant d'utiliser un modèle de graphe orienté pour décrire le parallélisme potentiel et les dépendances de données entre les opérations sommets du graphe. On définit ainsi un ordre partiel d'exécution sur ces opérations. De plus, nous avons pris plutôt un hypergraphe, ce qui permet de modéliser la diffusion de données. Cet hypergraphe est infiniment

itéré et conditionné pour prendre en compte les aspects réactifs temps réel et l'exécution exclusive de certaines opérations du graphe.

Il est intéressant de modéliser l'implantation également par un graphe orienté à partir des graphes d'algorithme et d'architecture. Nous proposons un modèle relationnel qui décrit l'ensemble des implantations valides associées à un couple (algorithme, architecture). Ce modèle est le résultat de la composition de trois relations sur des graphes de types différents : le routage, la distribution et l'ordonnancement. Les communications inter-processeur sont distribuées et routées sur les liaisons inter-processeur grâce à l'ajout au graphe de l'algorithme d'opérations de transfert qui vont être ordonnancées sur les média (ensemble composé d'une liaison et de ses unités de communication associées) du graphe de l'architecture routé. Chacune des implantations valides est un hypergraphe orienté conditionné, dont l'ordre partiel associé inclut l'ordre partiel initial du graphe de l'algorithme. Pour chaque unité de calcul de chaque processeur, on a une séquence d'opérations de calcul, et pour chaque unité de communication, on a une séquence d'opérations de communication. La génération de code ne consiste plus alors qu'à synchroniser ces séquences à l'aide de sémaphores. Plus précisément, il s'agit d'une part de synchroniser la séquence de calcul d'un processeur avec ses séquences de communication, et d'autre part de synchroniser entre elles les séquences de communication de processeurs différents.

Perspectives

Pour être plus réaliste, le modèle d'architecture devrait être étendu pour supporter en plus des liaisons SAM, des liaisons RAM.

Certaines parties du graphe de l'algorithme peuvent être répétitives (les nids de boucles par exemple), il serait donc intéressant de les factoriser à la fois pour simplifier la description par l'utilisateur, et à la fois pour en déduire les optimisations au niveau de la distribution et de l'ordonnancement.

De plus, l'algorithme est souvent décrit par les automaticiens comme un ensemble d'automates, représenté par un graphe flot de contrôle, activant des lois de commande représentées par des graphes flot de données. Les sommets du graphe de contrôle correspondent aux différents états du système représentant chacun une loi de commande différente, et les arcs indiquent sous quelles conditions un système transite d'un état à un autre. La distribution et l'ordonnancement d'un tel algorithme s'en trouveraient facilités s'il était possible de définir une transformation de la partie flot de contrôle en un graphe flot de données conditionné qui soit connecté aux différents graphes flot de données exécutées dans chaque état. Une étude est en cours dans [44].

Enfin, il serait intéressant de modifier le modèle d'implantation pour prendre en compte

les extensions des modèles précédents.

Optimisation

Principaux apports

La caractérisation qui consiste à étiqueter le graphe de l'algorithme en fonction des caractéristiques de l'architecture a été définie sur le graphe de l'algorithme implanté comprenant les opérations de calcul et les opérations de transfert. Un état de l'art a été effectué sur les méthodes de résolution des problèmes d'ordonnancement et ce chapitre a été partiellement intégré dans un état de l'art du projet national AEE (Architecture Electronique Embarquée) de l'INRIA. Une heuristique de liste, prenant en compte un modèle d'algorithme conditionné, et une heuristique de voisinage de recherche locale avec retour-arrière utilisant comme solution initiale l'implantation construite avec l'heuristique gloutonne, ont été définies. Une comparaison avec d'autres heuristiques du même type a été effectuée sur quelques exemples d'applications, cette comparaison a montré que les résultats fournis par les heuristiques définies sont de bonne qualité par rapport à ceux obtenus par les autres heuristiques.

Perspectives

La caractérisation doit être modifiée pour prendre en compte les interférences entre les calculs et les communications. En effet, dans notre modèle nous avons fait l'hypothèse simplificatrice que l'unité de calcul et les unités de communication d'un même processeur sont totalement indépendantes. En fait, ce n'est pas réellement le cas car les unités partagent le même bus mémoire et le séquenceur d'instruction. Le fait de partager un bus implique la présence d'un arbitre pour gérer les conflits et un seul séquenceur d'instructions par processeur implique que, pour débuter une communication, les calculs doivent être interrompus le temps que la communication débute, le calcul reprendra ensuite normalement jusqu'à l'interruption suivante. Ainsi le calcul est ralenti par les communications : on dit que les calculs et les communications interfèrent. Bien que le coût de l'arbitrage soit faible devant les coûts de communication et l'interruption soit de courte durée, si la modélisation de l'arbitrage et de l'interférence est effectuée, l'implantation associée sera plus précise.

La comparaison des heuristiques que nous avons proposées, avec d'autres doit être étendue. Des études statistiques doivent être menées sur un grand nombre de graphes d'algorithme et d'architecture pour vérifier la qualité des heuristiques gloutonne et de voisinage proposées.

De études sur de nouvelles variantes d'heuristiques optimisant une seule latence et prenant en compte les extensions des modèles d'architecture (liaisons RAM, interférence) et d'algorithme (factorisation partielle, conditionnement des opérations par des entiers) sont en cours [44, 33].

Développement logiciel

Principaux apports

Nous avons modifié l'interface graphique de SynDEx pour saisir des graphes conditionnés et nous avons implanté l'heuristique prenant en compte l'exclusivité des opérations en modifiant l'heuristique existante.

Perspectives

Il serait intéressant de modifier l'interface de SynDEx pour d'une part pouvoir saisir des graphes flot de contrôle et les transformer en graphes flot de données puis les connecter aux graphes de calcul, et d'autre part pouvoir saisir des graphes conditionnés non plus par des booléens mais par des entiers.

Une spécification d'une nouvelle heuristique statique est en cours pour prendre en compte les caractéristiques des nouveaux modèles d'architecture et d'algorithme.

A. Annexes

A.1 Ajout des conditions d'activation dans l'IHM

```
proc AjoutActi {canvas x y y1 nom} {
    global listporte$nom  colselect facteurcourant

    # de maniere implicite dans la base de donnees
    # type : bool
    # value : ??
    # polarite : 1 par defaut

    set value ??
    # Creation du port dans la base C++
    if { [set res [newportlogicielproc  ?  bool $value  $nom]] != ""} {
         .eval.t insert insert $res }

    if {$res == ""} {
    # creation du port d'activation du rectangle
     $canvas create polygon [expr "$x-3"] [expr "$y1-3"] \
    [expr "$x+3"]  [expr "$y1-3"] $x [expr "$y1+1"] \
    -tag $nom-$value -fill black

     # ajout de ce port d'activation a la liste des ports
     lappend listporte$nom $value
     # ajout a la liste de connection
     lappend listporte$nom bool
    }

    #conditions d'activation
    .net scale $nom-$value $x $y $facteurcourant $facteurcourant

     # Binding du port d'activation
```

```
    #-------------------------------
    # Mise en evidence du port de sortie
    $canvas bind $nom-$value <Enter> \
   "$canvas itemconfigure $nom-$value -fill $colselect"
    $canvas bind $nom-$value <Leave>   "LeavePort $nom-$value"
    # Connexion du port
    $canvas bind $nom-$value <ButtonPress-1> " SelectPort $nom $value"
    $canvas bind $nom-$value <Shift-ButtonPress-1> "BeginConnect $nom $value "
    # Deconnexion du port
    $canvas bind $nom-$value <Control-ButtonPress-1> "delcnxportbytag $nom-$value"
    # Polarite du port
    $canvas bind $nom-$value <Shift-ButtonPress-3> \
   "changepol $canvas $nom $value 1"
    $canvas bind $nom-$value <Control-ButtonPress-3> \
   "changepol $canvas $nom $value 0"
    #pour menu contextuel
    $canvas bind $nom-$value <ButtonPress-3>\
   "menucontextport $canvas $nom $value .mbar.acti.menu"
}
```

A.2 Déclarations des nouvelles classes

```
/********************** *********
 Declaration de la classe PORT_ACTI
********************** *********/
class PORT_ACTI : public PORT_IN {

 private :
 int polarite;     // 1 si la polarite  = vrai
                   // 0 si la polarite  = faux
 PORT_ACTI *next_port; // port d'activation suivant dans l'arborescence
                       // de conditionnement

 public :
```

```
 // Methodes d'acces
 int is_port_acti(){return 1;} //retourne 1 car c'est un port d'activation,
 int get_pol_acti(){return polarite;} // retourne la valeur de la polarite
 PORT_ACTI *get_next_port(){return next_port;}

 // Methodes de creation, modification.
 int set_pol_acti(int pol){polarite = pol; return 0;}
 void set_next_port(PORT_ACTI *port){next_port = port;}

 // Constructeur
 //        proprietaire
 PORT_ACTI(OPERATION *);
};

/********************** *********
Declaration de la classe OP_COND
************************ *********/
class OP_COND : public OPERATION {

 private :
 OPERATION *last_true;  // dernier successeur de l'operation sur la branche1
 OPERATION *last_false; // dernier successeur de l'operation sur la branche0

 public :
 // Methodes d'acces
    int is_op_cond(){return 1;} //retourne 1 car c'est une opn conditionnelle
    int is_schedulable() { return 0; } // une operation conditionnelle
      // n'est jamais ordonnancable! tout comme les const et les opns-coms
    OPERATION *get_last(int polarite); // si pol=1  =>last_true
                                       // si pol=0  =>last_false
 // Methodes de creation, modification.
 // Ajoute du cote de la polarite, l'operation donnee en argument
    void add_last(OPERATION *op, int polarite);
    void set_last(OPERATION *op, int polarite);

 // Retourne la plus grande des dates de fin des opns last_true et last_false
```

```
    int EEFS();

 // Retranche du cote de la polarite donnee en argument la derniere opn
 // ordonnancee du cote de la polarite
    OPERATION *rem_last(int polarite);

  // Constructeur      port de sortie, application
     OP_COND(PORT_OUT *port_out, APPLICATION *appli);
};
```

A.3 Nouvelles méthodes

A.3.1 Méthode *racine*

```
//-------------------------------------------------------------
// Methode     : racine
// Description : on remonte l'arborescence de conditionnement
//
// Entree      : rien
// Sortie      : le plus haut port de conditionnement de l'arborescence
// Remarque    : Si l'opération n'est pas conditionnée (=> son port
// d'activation n'est pas connecté), on ne rentre pas dans la boucle et
// la méthode racine retourne le pointeur NULL.
// Si l'opération est conditionnée par un unique booléen, la méthode
// racine retourne le port d'activation de l'opération.
//-------------------------------------------------------------
PORT_ACTI *OPERATION :: racine () {

 PORT_ACTI *port_crt, *port_prev;
 PORT_OUT *port_out;


// Initialisation
 port_crt =  get_port_acti();
 port_crt -> set_next_port (NULL);
```

```
// Boucle
 while ((port_out =  port_crt -> out_connected()) != NULL){
   port_prev = port_out -> operation() -> get_port_acti();
   port_prev ->set_next_port(port_crt);
   port_crt = port_prev;
 }
 return (port_crt ->get_next_port());
}
```

A.3.2 Méthode *schedule*

```
//-------------------------------------------------------------
// Methode     : schedule
// Description : Implante l'opération ayant appelé la méthode
//                sur l'opérateur passé en argument.
//               Crée toutes les opérations de transfert arrivant sur l'opération
// Entrée      : Opérateur où l'on souhaite implanter l'opération
// Sortie      : Rien
//-------------------------------------------------------------

void OPERATION::schedule(OPERATEUR *opr) {
 APPLICATION *applic;
 OPERATION *last_opn;
 OP_COND *opcond, *cond;
 PORT_ACTI *port_a;
 int polarite;


 applic = get_appli(); //utilisé par op_cond
 set_proc_owner(opr);
 polarite = TRUE;
 opcond = opr ->get_op_cond(); // pour descendre dans l'ordonnancement de l'operateur
 port_a = racine(); // pour descendre dans la branche de conditionnement

 while ( port_a != NULL ){
   last_opn = opcond -> get_last(polarite);
```

```
   if((last_opn == NULL) || !(last_opn -> is_op_cond()) || \
     (( last_opn->get_port_acti()->out_connected()) != port_a->out_connected()) {
     // si c'est une operation NULL OU si ce n'est pas une conditionnelle OU
     // si le port_out de cette cond est != du port_out auquel est
     // connecte le port_a)
       // creation d'une nouvelle conditionnelle sur l'opr avec le port_out
       // auquel est connecte le port_a
       cond= new OP_COND (port_a ->out_connected(), applic);
       cond->get_port_acti()->set_pol_acti(polarite);
       cond->set_proc_owner(opr);
       opcond->add_last(cond,polarite);
       last_opn = opcond ->get_last (polarite);
     }
   (opcond)  =  static_cast<OP_COND *>(last_opn);
   polarite = port_a ->get_pol_acti();
   port_a   = port_a ->get_next_port();
 }
 opcond -> add_last (this,polarite);
 make_com();
 invalide_ESFS();
}
```

A.3.3 Méthode *deschedule*

```
//-------------------------------------------------------------
// Méthode     : deschedule
// Description : Désordonnance l'opération de son opérateur
//               Destruction des opérations de transfert arrivant sur l'opération
// Entree      : rien
// Sortie      : rien
//-------------------------------------------------------------
void OPERATION::deschedule() {
OPERATION *prev_opn, *last;
OP_COND *opcond, *sched;;
int pol;

pol = TRUE; //initialisation de la polarite par defaut
```

```
if (is_active())   pol = get_polarite();
opcond  = get_owner(); // conditionnelle a laquelle appartient l'opn
deletecom(); // detruit toutes les communications arrivant sur l'operation

prev_opn=NULL;
last= opcond -> get_last(pol);
while ( last != this ){
  prev_opn = last;
  last = last->get_prev_scheduled();
}

if ( prev_opn != NULL)
  prev_opn ->set_prev_scheduled(this->get_prev_scheduled());
else {
  opcond->set_last(this->get_prev_scheduled(), pol);
    if (opcond !=NULL && opcond -> get_last(TRUE) == NULL && \
        opcond->get_last(FALSE) == NULL && opcond->get_owner() !=NULL) {
          opcond->deschedule();
          delete(opcond);
    }
}


set_owner(NULL);
set_prev_scheduled(NULL);
set_proc_owner(NULL);
invalide_ESFS(); //sa date n'est plus valide
}
```

Bibliographie

[1] T. Adam, K.M. Chandy, and J.R. Dickson. A comparison of list schedules for parallel processing systems. In *CACM*, volume 17, pages 685–690, 1974.

[2] I. Ahmad and Y. K. Kwok. Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors. In *International Symposium on Parallel Architecture, Algorithms and Networks*, pages 207–213, Beijing China, June 1996.

[3] C. Aiglon, C. Lavarenne, Y. Sorel, and A. Vicard. Utilisation de SynDEx pour le traitement d'images temps-réel. Rapport de Recherche 2968, INRIA, Septembre 1996.

[4] A. Arnold. Modèles et logiques pour la vérification. In *Actes de l'école d'été MOVEP -MOdélisation et VErifications des Processus parallèles*. Laboratoire d'Automatique de Nantes, 1996.

[5] Jean-Pierre Beauvais. *Etude d'algorithmes de placement de tâches temps réel périodiques complexes dans un système réparti*. PhD thesis, Université de Nantes, Juin 1996.

[6] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-time Systems. *Proceedings of the IEEE*, 79(9), September 1991.

[7] A. Benveniste, M. LeBorgne, and P. LeGuernic. SIGNAL as a Model for Real-Time and Hybrid Systems. Rapport de recherche 1608, INRIA, Feb 1992.

[8] J.F. Bonnans, C. Pola, and R. Rébaï. Perturbed path following interior point algorithms. *OMS*, 1995.

[9] P. Bournai, C. Lavarenne, P. Le Guernic, O. Maffeïs, and Y. Sorel. Interface signal-syndex. Technical Report 2206, INRIA, Mars 1994.

[10] J. C. Browne. Formulation and Programming of Parallel Computations : A Unified Approach. In *Proc. of Parallel Processing*, pages 624–631, 1985.

[11] J. Carlier and P. Chrétienne. *Problèmes d'ordonnancement*. Masson, 1988. Etudes et Recherches en informatique.

[12] H.H. ten Cate and E.A.H. Vollebregt. On the portability and efficiency of parallel algorithms and software. *Parallel Computing*, 22:1149–1963, 1996.

[13] B. Charron-Bost. *Mesures de la concurrence et du parallélisme des calculs répartis*. Thèse d'informatique, Université Paris VII, Septembre 1989.

[14] V. Chaudhary and J. K. Aggarwal. A generalized scheme for mapping parallel algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):328–346, March 1993.

[15] Ph. Chrétienne and C. Picouleau. *Scheduling with Communication Delays : A Survey*, chapter 4. Wiley, 1995.

[16] E. G. Jr. Coffman and R.L. Graham. Optimal scheduling for two-processor systems. In *Acta Informatica*, volume 1, pages 200–213, 1972.

[17] E.G. Jr. Coffman. *Computers and job-shop scheduling theory.* Wiley, New-York, 1976.

[18] M. Cosnard and A. Ferreira. The real power of loosely coupled parallel architectures. In *Parallel Processing Letters*, pages 103–112, 1991.

[19] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles.* InterEditions, PARIS, 1993.

[20] Michel Cosnard, Emmanuel Jeannot, and Tao Yang. Symbolic partitioning and scheduling of parameterized task graphs. Technical Report 1998-41, ENS Lyon, 1998.

[21] Jean de Rumeur. *Communications dans les réseaux de processeurs.* Etudes et recherches en informatique, masson edition, 1994.

[22] M. Dhodhi, Imtiaz Ahmad, and I. Ahmad. A Multiprocessor Scheduling Scheme Using Problem-Space Genetic Algorithms. In *IEEE International Conference on Evolutionary Computing*, Perth, Western Australia, November 1995.

[23] Coffman E.G. and Denning P.J. Operating systems theory. In NJ Englewood Cliffs, editor, *Series in Automatic Computation.* Prentice-Hall, 1973.

[24] D. Etiemble and J.C. Syre. Parallel architectures and languages europe. In *proceedings PARLE'92, 4th International PARLE conference*, Paris, June 1992.

[25] M.J. Flynn. Some computer organization and their effectiveness. *IEEE Transations on Computer*, pages 948–960, Sept 1972.

[26] U. Furbach. Formal specification methods for reactive systems. *Journal of Systems Software*, 21:129–139, 1993.

[27] Garey and Johnson. *Computers and intractability : a guide to the theory of NP-completeness.* W.H. Freeman and Company, 1979.

[28] A. Gerasoulis and T. Yang. On the Granularity and Clustering of Directed Acyclic Task Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6), 1993.

[29] P.B. Gibbons, Y. Matias, and Ramachandran V. The Queue-Read Queue-Write Asynchronous PRAM Model. In Springer, editor, *Europar'96*, pages 279–292, 1996.

[30] M. Gondran and M. Minoux. *Graphes et algorithmes.* Eyrolles, 1979.

[31] R.L. Graham, E.L. Lawler, Lenstra J.K., and Rinnoy Kan A.H.G. Opimization and approximation in deterministic sequencing and scheduling : a survey. *Ann. Discrete Math*, 5:287–326, 1979.

[32] T. Grandpierre. *Spécification en langage orienté objet pour le portage du cœur de SynDEx*, Université d'Orsay edition, Août 1996. Rapport de DEA.

[33] T. Grandpierre. *Génération automatique d'exécutifs distribués temps réel optimisés.* PhD thesis, Université Paris XI Orsay, à paraître.

[34] T. Grandpierre, C. Lavarenne, and Y. Sorel. Modèle d'exécutif distribué temps réel pour SynDEx. Rapport de recherche, INRIA, Août 1998.

[35] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Pub., 1993.

[36] C. Hanen and A. Munier. *Scheduling Theory and its Applications*, chapter 9. Wiley, 1995.

[37] D. Harel. StateCharts : A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[38] D. Harel and A. Pnueli. On the development of reactive systems. *Logic and Models of Concurrent Systems*, 1985. Advanced Study Institute on Logics and Models for Verification, Springer Verlag.

[39] H. Heping and H. Zedan. A fast prototype tool for parallel reactive systems. *Journal of Systems Architecture*, 42:251–266, 1996.

[40] T.C. Hu. Parallel sequencing and assssembly line problems. *Operational Research*, 9:841–848, 1961.

[41] J.J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Computing*, pages 244–257, Aug. 1989.

[42] E.F. Johnson. *Graphical applications with Tcl and Tk.* M and T books, 1996.

[43] S. J. Kim and J. C. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. In *Proc. of Int'l Conference on Parallel Processing*, volume 3, pages 1–8, Août 1988.

[44] R. Kocik. *Optimisation des systèmes distribués temps réel embarqués pour l'automobile.* PhD thesis, Université de Rouen, à paraître.

[45] R. Kocik and Y. Sorel. A Methodology to Design and Prototype Optimized Embedded Robotic Systems. In *2nd IMACS International Multiconference CESA'98*, Hammamet, Tunisie, 1998.

[46] Y. K. Kwok and I. Ahmad. Bubble Scheduling : An Efficient Algorithm For Compile-Time Scheduling of Parallel Computations on Messages-Passing Architectures. *Journal of Parallel and Distributed Computing.* to appear.

[47] Y. K. Kwok and I. Ahmad. Bubble scheduling : A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In *Proc of 7th IEEE symposium on Parallel and Distributed Processing*, pages 36–43, Octobre 1995.

[48] Y. K. Kwok and I. Ahmad. Dynamic critical-path scheduling : An effective technique for allocating task graphs to multiprocessors. In *IEEE Trans. on Parallel and Distributed Systems*, volume 7, pages 506–521. May 1996.

[49] Y. K. Kwok, I. Ahmad, and J. Gu. FAST : A Low-Complexity Algorithm For Efficient Scheduling of DAGs on Parallel Processors. In *Proceedings of the 25th International Conference on Parallel Processing*, volume 2, pages 150–157, Aout 1996.

[50] J.-C. Laprie et al. *Dependability: Basic Concepts and Terminology.* Dependable Computing and Fault-Tolerant Systems. Springer-Verlag, Wien, New York, 1992.

[51] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The SynDEx software environment for real-time distributed systems design and implementation. In *Proc. of the European Control Conference*, 1991.

[52] C. Lavarenne and Y. Sorel. Optimisation et génération d'exécutifs distribués temps réel pour algorithmes spécifiés avec les langages synchrones. In *RTS'94 – Paris, 11–14 Janvier.*

[53] C. Lavarenne and Y. Sorel. The SynDEx software. http://www-rocq.inria.fr/syndex/welcome.html.

[54] C. Lavarenne and Y. Sorel. Performance Optimization of Multiprocessor Real-Time Applications by Graph Transformations. In *Parallel Computing*, Grenoble, Septembre 1993.

[55] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, vol 14, n.6, 1997.

[56] J.K. Lenstra and A.H.G. Rinnoy Kan. Complexity of scheduling under precedence constraints. In *Operations Research*, pages 22–35. Janvier 1978.

[57] V. Leppänen. Goodness of Time-Processor Optimal PRAM Simulations. In Springer, editor, *Europar'96*, pages 303–306, 1996.

[58] Z. Liu and C. Coroyer. Effectiveness of heuristics and simulated annealing for the scheduling of concurrent task. an empirical comparison. *Proc. of PARLE'93, 5th international PARLE conference, Munich, Germany, June 14-17*, pages 452–463, Nov. 1993.

[59] O. Maffeïs. *Ordonnancements de graphes de flots synchrones; Application à la mise en œuvre de Signal.* PhD thesis, Université de Rennes 1, Jan 1993.

[60] Mehdiratta N. and Ghose K. A Bottom-Up Approach to Task Scheduling on Distributed Memory Multiprocessor. In *Int'l Conf. on Parallel Processing*, volume 2, pages 151–154, Aug 1994.

[61] J.M. Nash, P.M. Dew, J.R. Davy, and M. E. Dyer. Implementations Issues Relating to the WPRAM Model for Scalable Computing. In Springer, editor, *Europar'96*, pages 319–326, 1996.

[62] R. Nikoukhah and S. Steer. Scicos : A dynamic system builder and simulator user's guide, Juin 1997. Version 1.0.

[63] S. Norre. *Problème de placement de tâches : méthodes stochastiques et évaluation de performances.* PhD thesis, Université Blaise Pascal Clermont-Ferrand, 1993.

[64] C.N. Nikolaou N.S. Bowen and A. Ghafoor. On the assignment problem of arbitrary process systems to heterogeneous distributed computer systems. *IEEE Transactions on Computers*, pages 257–273, March 1992.

[65] J.K. Ousterhout. *TCL and the TK toolkit.* Addison-Wesley professional computing series, 1994.

[66] F. Panzieri and R. Davoli. Real Time Systems : A Tutorial. Technical report UBLCS-93-22, University of Bologna, Oct. 1993.

[67] L. Phelippeau-Gelineau. *Etude de problèmes d'ordonnancement multiprocesseur avec communication par diffusion.* PhD thesis, Université Paris VI, 1996.

[68] V.J. Rayward-Smith. Uet scheduling with unit interprocessor communication delays. *Disc. App. Math.*, 18:55–71, 1987.

[69] H. El. Rewini and T. G. Lewis. Scheduling parallel programs onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[70] R. Rockafellar and Tyrrell. *Convex analysis.* Princeton University Press, 1972.

[71] Group Scilab. A free matlab like software available via ftp anonymous on ftp@inria.fr (192.93.2.54). /INRIA/Projects/Meta2/Scilab.

[72] Y. Shin and K. Choi. Thread-Based Software Synthesis for Embedded System Design. In *European Design and Test Conference*, Paris, Mars 96.

[73] G.C. Sih and E.A. Lee. A Compile-Time Scheduling Heuristic for Interconnection Constrained Heterogeneous Processor Architectures. *IEEE Trans. on Parallel and Distributed Systems*, 4(2), 1993.

[74] Y. Sorel. Massively parallel systems with real time constraints. the "Algorithm Architecture Adequation Methodology". In *Proc. Massively Parallel Computing Systems*, Italy, May 1994.

[75] Y. Sorel. Real-Time Embedded Image Processing Applications using the A^3 Methodology. In *Proc.IEEE International Conference on Image Processing*, Switzerland, Sep. 1996.

[76] J. A. Stankovic. Distributed Real-Time Computing : The Next Generation. *Journal of the Society of Instrument and Control Engineers of Japan*, 1992. également publié à l'Université du Massachusetts, rapport UM-CS-1992-001.

[77] B. Stroustrup. *The C++ programming language.* Addison-Wesley, third edition, 1998.

[78] SYNCHRON. The common format of synchronous languages -the declarative code DC version 1.0. Technical report, C2A-SYNCHRON project, October 1995.

[79] A. Tiskin. The Bulk-Synchronous Parallel Random Access Machine. In Springer, editor, *Europar'96*, pages 327–338, 1996.

[80] TMS320C4x User's Guide. Texas Instruments.

[81] T. A. Varvarigou, V. P. Roychowdhury, T. Kailath, and E. Lawler. Scheduling In and Out Forests in the Presence of Communication Delays . *IEEE Transactions on Parallel and Distributed Systems*, 7(10), 1996.

[82] A. Vicard. *Evaluation du logiciel SynDEx et de la méthodologie Adéquation Algorithme Architecture*, Juillet 95. Rapport ETCA 95 R 103.

[83] A Vicard and Y. Sorel. Formalisation et Optimisation statique d'implantations parallèles. In *Deuxièmes Journées francophones de recherche opérationnelle*, 1998.

[84] A. Vicard and Y. Sorel. Formalization and static optimization for parallel implementations. In *DAPSYS'98 Workshop on Distributed and Parallel Systems*, 1998.

[85] M. Y. Wu and D. D. Gajski. Hypertool : A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, July 1990.

[86] T. Yang, C. Fu, A. Gerasoulis, and V. Sarkar. Mapping Iterative Task Graphs on Distributed Memory Machines. In *Proc of 24th International Conf. on Parallel Processing*, volume 2, pages 151–158, August 1995.

[87] T. Yang and A. Gerasoulis. DSC : Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–957, 1994.

[88] T. Yang and A. Gerasoulis. List Scheduling with and without Communication Delays. Rutgers University, August 92.

[89] J. Zwiers and W. Janssen. Partial order based design of concurrency systems. In Springer Verlag, editor, *REX School/Symposium*, pages 622–684. In a Decade of Concurrency, reflexions and perspectives, June 1993.